An Agent-Based Modeling Framework and Application for the Generic Nuclear Fuel Cycle

by

Matthew J. Gidden

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Nuclear Engineering & Engineering Physics)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 03/26/2015

This work is dedicated to my parents, Howard and Melanie, who always believed that I could do anything I set my mind to, and to Ms. Margaret Maes, who has been my solid foundation through the highs and lows of the past few years.

Acknowledgments

This work would never have materialized without the dedicated students and staff who comprise and have comprised the Fuel Cycle Group of the UW Computational Nuclear Engineering Research Group (CNERG). There would be no Cyclus without the dedication of Dr. Katy Huff, under whose watchful eye Cyclus work was initiated. I honed my programming chops through Cyclus development with Robert Carlsen, who has reviewed many a pull request of mine. Cyclus would likely have no users without the hard work of Dr. Anthony Scopatz, who also provided valuable advice through many parts of the testing and analysis work herein described. Dr. Paul Wilson has served as a thoughtful advisor and mentor throughout my tenure at UW.

I would also like to thank the good folks at the Department of Energy's Nuclear Energy University Program (NEUP) office. The majority of my time at UW was funding by an NEUP fellowship, which provided the freedom necessary to chase after a wacky idea like agent-based modeling of the nuclear fuel cycle. Getting it right (enough) took a lot of work and effort that is only now starting to bear fruit. The NEUP's support was crucial for this work.

Contents

List of Tables

List of Figures

Abstract

Key components of a novel methodology and implementation of an agent-based, dynamic nuclear fuel cycle simulator, Cyclus , are presented. The nuclear fuel cycle is a complex, physics-dependent supply chain. To date, existing dynamic simulators have not treated constrained fuel supply, time-dependent, isotopic-quality based demand, or fuel fungibility particularly well. Utilizing an agent-based methodology that incorporates sophisticated graph theory and operations research techniques can overcome these deficiencies. This work describes a simulation kernel and agents that interact with it, highlighting the Dynamic Resource Exchange (DRE), the supply-demand framework at the heart of the kernel. The key agent-DRE interaction mechanisms are described, which enable complex entity interaction through the use of physics and socio-economic models. The translation of an exchange instance to a variant of the Multicommodity Transportation Problem, which can be solved feasibly or optimally, follows. An extensive investigation of solution performance and fidelity is then presented. Finally, recommendations for future users of Cyclus and the DRE are provided.

1 Introduction

1.1 The Nuclear Fuel Cycle

The nuclear fuel cycle can be described as a set of facilities that interact with one another to either provide or consume fuel services. Facilities in the fuel cycle act together to provide fuel to nuclear power plants which, in turn, generate energy. The used fuel produced by the power plants is then returned to servicing facilities to either be recycled or disposed. The overall goal of the system is to produce power at a competitive price while managing externalities of the process, the chief of which is spent nuclear fuel. Myriad strategies exist to achieve this aim which can be classified along a spectrum of the degree to which fuel is recycled. In general, fuel cycles that do not recycle fuel are concerned most with cost, whereas fuel cycles that fully recycle fuel are concerned most with issues of sustainability and inter-generational equity. It is the goal of fuel cycle simulation to rigorously explore this option space.

1.1.1 The Open Fuel Cycle

The open, or once-through, fuel cycle is relatively simple and is in place in most nation states that currently utilize nuclear power. In practice, the primary fuel element used in this type of cycle is uranium; however, processed fertile material, such as thorium, can also be used. The fuel cycle is considered open because fuel that is used in a reactor is stored indefinitely once its reactivity has dropped below useful levels.

Beginning the fuel cycle process, uranium ore is initially extracted from the ground using one of a variety of techniques including open pit mining, underground mining, and *in situ* leaching. The uranium ore is then milled to form yellowcake, U_3O_8. The tailings, or byproducts, of this process are slightly radioactive and are therefore considered to be low-level waste (LLW) by the Nuclear Regulatory Commission (NRC) (see [7]).

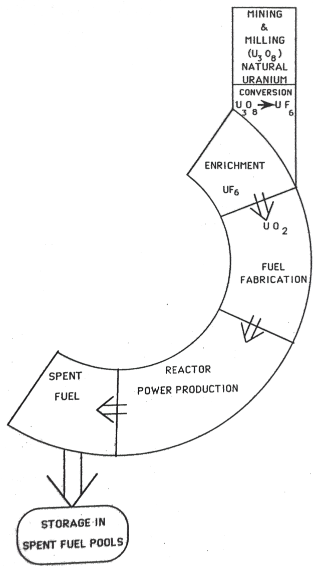
MINING & MILLING (U_3O_8) NATURAL URANIUM
CONVERSION $U_3O_8 \rightarrow UF_6$
ENRICHMENT UF_6
UO_2
FUEL FABRICATION
REACTOR POWER PRODUCTION
SPENT FUEL
STORAGE IN SPENT FUEL POOLS

Figure 1.1: The once-through fuel cycle as shown in [19]. Fuel in spent fuel pools eventually will be sent to a geologic repository, although none are yet operating in the world.

Certain reactors are designed to use naturally enriched uranium. For these reactors, yellowcake can be directly converted to naturally enriched uranium oxide, UO_2. For the majority of power reactors, however, the uranium must be enriched with higher-than-natural levels of uranium-235. In order to do so, yellowcake is sent to a conversion facility, which converts it from U_3O_8 to UF_6. The uranium hexafluoride is then enriched to the required level in an enrichment facility, of which three classes exist: gaseous diffusion, the original enrichment technology; centrifugal diffusion, the current enrichment technology; and Atomic Vapor Laser Isotope Separation (AVLIS), a newer technology not currently in commercial production. The enriched uranium hexafluoride is then sent to a fuel fabrication facility where it is returned to yellowcake form before being reduced to uranium oxide. The uranium oxide is then sintered into pellets and loaded into fuel assemblies to be placed in a reactor. This process, in conjunction with uranium mining, is termed the *front end* of the nuclear fuel cycle.

Once fuel has been processed in a reactor, it is cooled off in pools for a number of years, and then stored in dry casks before eventually being sent to a final geologic repository. The physical location of the fuel may vary during dry cask storage between the reactor site or some other interim storage site. Graphically, the open fuel cycle is shown in Figure 1.1, where fuel in spent fuel pools eventually will be transferred to a geologic repository.

1.1.2 The Closed Fuel Cycle

The closed fuel cycle is one that includes the recycling of used, or spent, fuel to be reused in a reactor. Recycling used nuclear fuel is expensive due to the costs associated with handling highly radioactive material (e.g., capital costs of hot cells, etc.). However, there are at least two overarching benefits that contribute to lowering the overall cost of the fuel cycle: increasing repository capacity and increasing fuel utilization.

Spent fuel that exits the average light water reactor (LWR) has an approximate composition as shown in Table 1.1. Of the elements that comprise used fuel, uranium, plutonium, and the mixed actinides (MA) are all capable of producing power through the fission process. The fission products, however, contain isotopes with high neutron capture cross sections, which therefore act as poisons to the nuclear chain reaction. Achieving theoretical 100% fuel utilization would thus require storing indefinitely only the fission products and any other byproducts of the fuel cycle, rather than additionally having to store other elemental groups produced by nuclear fission, e.g., minor actinides. Furthermore, repository capacity is determined not only by total mass or volume, but also by heat load and radiotoxicity, making the concentration of high-activity isotopes one of the limiting factors in a repository's capacity. Fission products are generally short-lived (in comparison to transuranic elements, i.e., uranium, plutonium, and the MAs). Accordingly, for repositories with long-term heat load limited capacities, minimizing the amount of transuranics increases the amount of material that can be stored in a given repository.

Element Group	wt %
Uranium	$\sim$95
Plutonium	$\sim$1
Mixed Actinides	$\sim$0.1
Fission Products	$\sim$4

Table 1.1: Elemental Breakdown of Spent Fuel Exiting a Typical LWR

The act of reprocessing spent fuel is comprised of a number of subprocesses. Once fuel has left the reactor core, it is stored in a spent fuel pool for a some number of years, typically around five, in order to provide enough time to lower decay heat to acceptable levels for handling of the fuel. It can then be directly sent to a reprocessing facility or be sent for some period of time to dry-cask storage. Reprocessing nuclear fuel is a chemical extraction process and therefore is limited by chemical extraction techniques. In general, there are two types of such processes: low-temperature methods using organic solvents (e.g., PUREX),

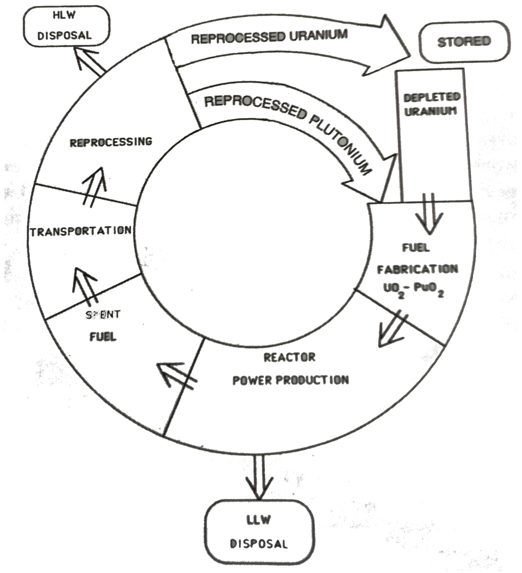

Figure 1.2: The closed fuel cycle as shown in [19].

and high-temperature methods using molten salts and metals, called pyroprocessing. The extraction techniques separate the spent fuel into chemically-similar groups which can vary based on the technique used, but generally align with those shown in Table 1.1. The separated streams are then sent either to a repository as high-level waste (HLW) or to an appropriate fuel fabrication facility. Graphically, the closed fuel cycle is shown in Figure 1.2.

The elemental groups used in fuel fabrication will depend on the fuel cycle that is developed. Current large-scale industrial reprocessing plants, i.e., La Hague in France, THORP in the U.K., Mayak in Russia, and Rokkasho in Japan (still technically under construction [4]), utilize the PUREX process to extract uranium and plutonium. The plutonium is then oxidized and mixed with depleted uranium from the enrichment process to produce mixed-oxide fuel (MOX). Other sources of uranium can be used to fill MOX fuel, such as recycled uranium from reprocessing, as neutronics-related reactivity and safety constraints allow. Other fuel cycles utilize the mixed actinides elemental group as well. Generally, plutonium is included with the mixed actinides, which results in a elemental category called the transuranic (TRU) elements. These fuel cycles generally include fast reactors that convert their TRU inventory into either more TRU (i.e., they have a conversion ratio (CR) of greater than 1), less TRU (CR < 1), or they maintain the amount of TRU entering and exiting their system (CR = 1). Fast reactors with CR > 1 are termed breeder reactors.

It should be noted that with any reprocessing capability, nonproliferation issues arise. Nuclear weapons have historically been produced using either enriched uranium or reprocessed plutonium; however it is possible to produce one with any mix of appropriate materials. Accordingly, any fuel cycle that exposes bare plutonium streams has an inherently higher nonproliferation risk than one that does not, and such risks must be weighed accordingly. However, the primary nuclide that drive such plutonium-based weapons is ^{239}Pu . Additional isotopes, e.g., ^{240}Pu , are considered "impurities" that dilute the efficacy and reliability of plutonium-based weapons. In general, fuel exiting a full LWR cycle have very poor plutonium profiles for the purpose of weapon utilization.

1.1.3 The Modified-Open Fuel Cycle

The modified open fuel cycle is effectively a hybrid of the open and closed fuel cycles. The Blue Ribbon Commission's Reactor and Fuel Cycle Technology Subcommittee tackled a definition as follows:

> We have defined this category to encompass a very wide range of possible fuel cycles with multiple possible combinations of different reactor, separations, and fuel fabrication technologies. Our definition includes any fuel cycle in which some of the spent fuel is processed rather than being directly disposed of after a single pass through a reactor. [13]

1.2 Nuclear Fuel Cycle Simulation

Fuel cycle simulation is a field with a variety of actors, including governments, universities, and international governance organizations. Accordingly, a variety of modeling strategies have been applied to the nuclear fuel cycle. Such strategies span a wide range of fidelity, both at the facility level and the material level. For instance, some simulators describe reactors by fleet (or types) and solve material balances for the entire fleet in aggregate [54, 63] while others instantiate individual (or discrete) facilities [50]. Similarly, some simulators make detailed calculations of fuel depletion due to reactor fluence [14] whereas others simply use pre-tabulated values that depend (generally) on burnup values for thermal reactors and conversion ratios for fast reactors.

There are, broadly, three decision categories that are of concern to fuel cycle simulation. The first is facility deployment, i.e., how, why, and when certain facilities are deployed. In the current simulation development environment, the most common reactor deployment mechanism is allowing a user to define an energy growth curve and, for each type of reactor in the simulation, a percentage of that total energy demand to be met by the reactor type. However, the nuclear fuel cycle is a special case of supply-demand modeling where certain facilities (e.g., fast reactors) require fuel that has been processed by other facilities (e.g., thermal reactors). Accordingly, simulation developers must make a choice regarding the ability for facilities to be built if fuel may not be available for their use. Certain simulators explicitly disallow this behavior by determining reactor build decisions based on look-ahead algorithms [51], others explicitly allow it, while still others offer a hybrid approach that allow a look-ahead function based on a certain amount of fuel that will eventually be needed over a reactor's lifetime [22]. The eventual choice of this decision making process greatly affects simulation outcomes in any scenario in which there is competition for recycled fuel. Because these simulation tools are built to analyze the dynamic symbiotic relationship between different reactors in a cyclical process (e.g., thermal and fast reactors), among other scenarios, this simulation development decision is arguably very important to simulation outcomes.

The second simulation design decision category is the level of fidelity with which to model the physical and chemical processes involved in the nuclear fuel cycle. Broadly, physical fidelity includes two processes, isotopic decay and isotopic transmutation due to residency in a reactor. Physical fidelity is an important concern because fuel cycle simulation measures individual isotopic masses at each point in the fuel cycle, and the isotopic profiles of those mass streams change due to physical processes.

Isotopic decay is important to consider because some isotopes decay on time scales on the order of or smaller than the simulation time. ^{241}Pu , for instance, has a half life of $\sim$14 years. Simulators fall into two camps, those that include decay and those that do not. Interestingly, the MIT development team claims that the lack of modeling decay does not affect the simulation as long as all transuranic isotopes are lumped together [30]. Other codes include isotopic decay in order to inform output metrics such as repository heat capacity.

Reactor physics, i.e., the process by which the transmuted isotopic profile of fuel due to reactor residency is determined, is also an important physical consideration. The rigorous solution of reactor physics equations is an entire field in nuclear science unto itself, and is thus not normally treated by fuel cycle simulators. In most cases for the current suite of simulators, some amount of calculation is performed before a simulation is run, and isotopic profiles are determined via look-up tables. Some simulators, however, choose to perform transmutation calculations *in situ*, during the simulation.

The third simulation-level design decision concerns the connections between facilities and the type of material that flows along those connections. In general, connections between facilities can either be static or dynamic, and can either be fleet-based or facility-based. A static connection implies that material will always flow between two types of facilities, whereas a dynamic connection implies that a facility's input or output connection may change. For those simulators that model fleets of facilities with static connections, the modeling technique is relatively trivial: a fleet of servicing facilities are directly connected to their serviced facilities. If more than one type of facility is being serviced (e.g., TRU-based fuels going to thermal and fast reactors), then either a user or the simulation engine must define the percentage of capacity going towards each type of serviced facility [54].

Most simulators to date have taken the fleet-based, static-connection approach to modeling fuel cycles which lacks the ability to be easily extended and improved upon, a key feature of a research code. This work enables the dynamic connection approach in the Cyclus nuclear fuel cycle simulator . Dynamic connections between facilities allow for more complicated scenarios, e.g., scenarios with regional influences or scenarios in which competition for resources exists, to be modeled. The dynamic exchange of resources, however, introduces two complications. The first is that a given need, e.g., for fuel, can be met by multiple commodities. As an example, consider fuel for thermal reactors. Thermal reactors can be fueled by either uranium oxide (UOX) or mixed uranium-plutonium oxide (MOX). Furthermore, MOX fuel is composed of plutonium (and some minor actinides, such as americium) from spent thermal fuel as well as uranium (the source of which can be depleted enrichment tails, depleted recycled uranium or natural uranium). The second is that the isotopics comprising fuel orders are *fungible*. A nuclear reactor generates power by fissioning nuclei. Whether the fissile nuclei involved is ^{235}U , ^{239}Pu , or ^{233}U makes little difference from a power-generation standpoint – each generates power. However, each is involved with a different nuclear fuel cycle.

Supporting economic and social models is rare among simulators. Only one simulator purports to have any *in situ* economic decision making [22]. A single other simulator reports including any socio-geographic concerns [9]. Any supply-demand framework that enables< economic, geographic, or other behavior models is a novel step forward in the realm of computational fuel cycle simulation and analysis.

The core concept that connects both the design decisions regarding fidelity and facility connections is the notion of material *quality*. Because of the nature of nuclear reactors, the simulation of their fuel usage must consider the isotopic profile of material being produced and consumed. This additional concern greatly complicates the modeling of fuel cycles and must be taken into account. Each of the issues addressed by the fidelity decision, decay, transmutation, and fuel fabrication, are all operations on the isotopic profile of material.

1.3 Tools Used

1.3.1 Agent-Based Supply Chain Models

Supply and demand in nuclear fuel cycle simulation drives the flow of resources between entities. The exchange of resources is the primary entity interaction mechanism during a simulation. The supply and demand of a given fuel cycle becomes highly complex when including the recycling of material. Even with such a complication, the notion of a generic fuel cycle, i.e., from the perspective of facilities that supply and demand material, quickly begins to look like a supply chain model. There is a growing literature of agent-based supply chain modeling [18, 34, 37, 57, 64]. The general premise of these types of models is that individual facilities have a notion of their needs (i.e., their demands) and can express to the system these needs at the required time. There is heavy use of inventory policy to determine the correct amount of material inventory that is needed and the correct time to request a resupply. Such an approach has not heretofore been attempted for the nuclear fuel cycle and would support a variety of use cases outlined in section 1.2. For example, reactor facilities could be allowed to be fueled by multiple fuel types (e.g., UOX or MOX), and decide which type to choose based on the simulation environment.

1.3.2 Cardinal Preferences and Game Theory

The notion of social modeling in fuel cycle simulation, e.g., employing regional bias, has to date been a secondary concern. Some semblance of this capability is needed if one is to incorporate outside effects on a domestic fuel cycle model. Furthermore, a robust capability is required if one wishes to actually investigate dynamic interactions between regional entities. Again, a full treatment of this sort of regional interaction would require international relations models, most of which can be found in the cross-cutting disciplines of economics, political science, and game theory. The primary solution technique in game theory is Nash Equilibrium. It describes an optimal solution as follows: given a set of players, states, preferences, and actions, all players choose an action such that any single player's deviation from that actions results in a state of lower preference for that player (thus no player has an incentive to deviate) [44]. There also exists a body of literature that examine Nash Equilibria in the context of optimal flow models [43, 47, 55]. However, the complexity of such models quickly brings them out of the scope of our needs, i.e., dynamic modeling of multi-lateral scenarios ranging 100+ years in a "reasonable" amount of computation time.

The game theoretic notion of preferences can be quite useful in fuel cycle simulation, however. Specifically, cardinal utility, or cardinal preferences [56] provides a relative measure of preference such that any two preferences can be directly compared, provided an arbitrary scaling, similar to the comparison of costs in a system. The notion of preference also nicely extends the work of Oliver's affinity metric [48]. Costs in a nuclear fuel cycle simulation have reasonably large uncertainty [53] and are generally applied to the output of a simulator as a post-processing step. Furthermore, for many fuel cycle simulation cases, a cost proxy is directly applicable, because actual cost values may be very hard to compute or determine *within* a simulation. For example, an analyst may know intuitively that reactors prefer recycled fuel to fresh fuel in order to maximize resource utilization. Especially for initial analyses, informing potential resource flows via preferences, rather than costs, is simpler, quicker, and more intuitive. Further, the notion of preference cleanly maps onto geopolitical models, such as preferential trading, whereas cost usage can require significant additional work to be meaningful. Enabling both preference and cost-based models can provide sufficient simulation fidelity to appropriate analysts.

1.3.3 Transportation Problems & Mathematical Programming

The previous sections have outlined a specific need for nuclear fuel cycle simulation: determining the flow of resources in a system of supply and demand, given a variety of possible capacities and informed by economic and social models. Constrained network flow determination is a canonical problem in computer science and operations research. Further, there is a rich history and capability of modeling such problems using mathematical programming.

A network flow model is represented by a graph, $G(N, A)$, comprised of nodes N and arcs A. If flow can occur between some node i and some other node j, then it flows along arc (i, j). An example of a network-flow graph is shown in Figure 1.3.

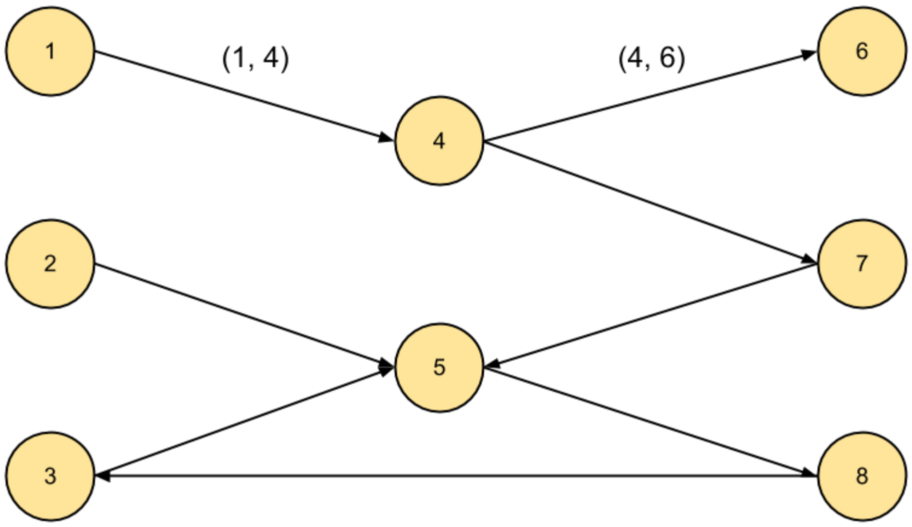

Figure 1.3: An example node-arc network configuration. Arrows denote possible flow directions. Arc notation examples are provided for arcs (1,4) and (4,6).

Given a graph instance, an optimal flow between nodes can be found provided *objective coefficients* and *constraints*. *Decision variables* for this optimization problem comprise the optimal *flow assignment*. If all decision variables are linear, then the resulting formulation is termed a Linear Program (LP), and can be solved using related techniques. A full discussion of LPs and their solution techniques is provided in Appendix A. If any decision variables are integer, then the resulting formulation is termed a Mixed-Integer Linear Program (MILP). For instance, binary variables can be used in network-flow problems to denote whether an arc has flow or not. A full discussion of MILPs and their solution techniques is provided in Appendix B. Many different types of problems can be solved using this structure. This work utilizes *transportation* problems, a specialization of the network-flow problems.

Transportation problems model the flow of a commodity between source nodes and sink nodes. In other words, source nodes and sink nodes comprise two distinct subsets, N_1, N_2, the union of which comprises all nodes in the transportation graph, N. These properties can be described in set notation.

$$N_1 \subset N \tag{1.1}$$

$$N_2 \subset N \tag{1.2}$$

$$N_1 \cup N_2 = N \tag{1.3}$$

From the node-arc graph point of view, this strict subset division allows for the transportation problem to be modeled as a *bipartite* graph, an example of which is shown in Figure 1.4.

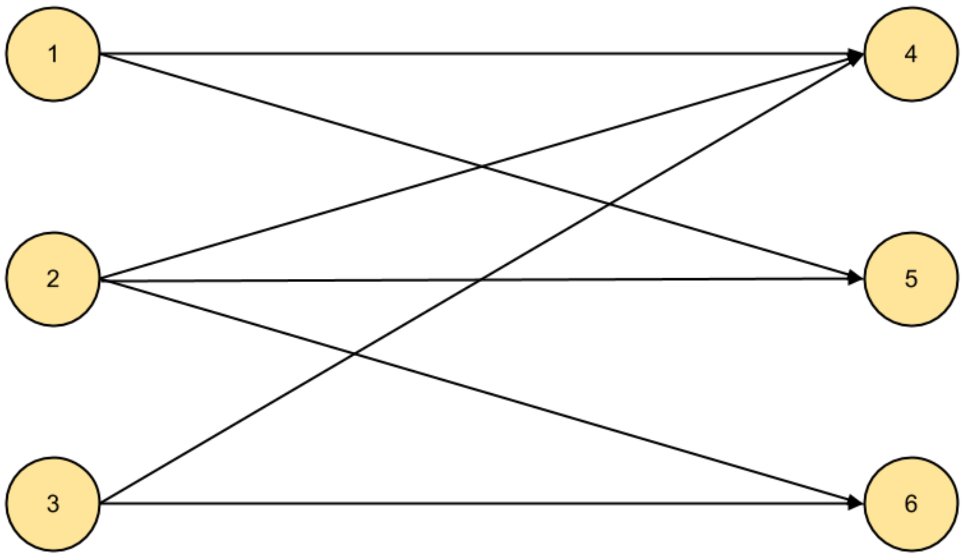

Figure 1.4: An example node-arc transportation network configuration. Arrows denote possible flow directions. Note that all nodes either belong to the set of sources (left) or set of sinks (right).

Many variations of the transportation problem exist. The minimum-cost transportation problem is a useful example. In such a formulation, each arc has an associated *unit cost* associated with the cost of transporting a unit of a commodity along it, $c_{i,j}$. Additionally, supplier and consumer nodes have an associated supply, s_i, or demand, d_i, which provide a notion of *node capacity*. The minimum-cost transportation problem can be formulated as a linear program as shown in Equation 1.4.

$$\min_{x} \sum_{(i,j)\in A} c_{i,j} x_{i,j} \tag{1.4a}$$

$$\text{s.t.} \sum_{j\in N_2} x_{i,j} \le s_i \qquad \forall i \in N_1 \tag{1.4b}$$

$$\sum_{i\in N_1} x_{i,j} \ge d_j \qquad \forall j \in N_2 \tag{1.4c}$$

$$x_{i,j} \ge 0 \qquad \forall (i,j) \in A \tag{1.4d}$$

An intuitive constraint on the problem to guarantee a feasible solution is that the total demand in the system must be no greater than the total supply in the system, shown in Equation 1.5.

$$\sum_{j\in N_2} d_j \le \sum_{i\in N_1} s_i \tag{1.5}$$

A given problem instance may violate Equation 1.5 and thus be infeasible. Feasibility in this sense can be guaranteed by adding an artificial supply node. Such a node can have infinite supply capacity but at (effectively) infinite cost. The problem can then be solved, and any flow leaving the artificial node in the optimal solution can be dealt with accordingly, e.g., it can be ignored.

A more complex transportation-problem formulation can support systems in which supply or demand can be met by multiple commodities. Variables and constants in the multi-commodity formulation are generally analogs of their counterparts in the single-commodity problem. There is a unit cost $c^h_{i,j}$ for commodity h to traverse arc (i,j). A supplier of commodity h has a certain supply capacity s^h_i which cannot be surpassed and consumers of commodity h have a certain demand level which must be met, d^h_i.

In the simplest extension from the single-commodity to multi-commodity transportation problem, arc constraints for all commodities are combined, i.e., there is a single capacity $u_{i,j}$ for a given arc (i,j). A classic application of this enhanced complexity deals with data networks. Multiple classifications of data exist, but they all must traverse the same network infrastructure. Accordingly, the infrastructure can only accommodate a certain quantity of total flow among all communication types. The formulation of the multi-commodity flow problem is shown in Equation 1.6. Note the commodity coupling in Equation 1.6d.

$$\min_{x} \sum_{i\in I}\sum_{j\in J}\sum_{h\in H} c_{i,j}^{h} x_{i,j}^{h} \tag{1.6a}$$

$$\text{s.t.} \sum_{j\in J} x_{i,j}^{h} \leq s_{i}^{h} \qquad \forall\, i\in I, \forall\, h\in H \tag{1.6b}$$

$$\sum_{i\in I} x_{i,j}^{h} \geq d_{j}^{h} \qquad \forall\, j\in J, \forall\, h\in H \tag{1.6c}$$

$$\sum_{h\in H} x_{i,j}^{h} \leq u_{i,j} \qquad \forall\, (i,j)\in A \tag{1.6d}$$

$$x_{i,j}^{k} \geq 0 \qquad \forall\, (i,j)\in A, \forall\, h\in H \tag{1.6e}$$

It can be possible to reduce instances of the multi-commodity transportation problem. In the case in which is no arc that shares multiple commodities, the multicommodity connection constraints disappear, and the single multi-commodity problem can be broken into m different single-commodity transportation problems, where m is the cardinality of the set of commodities, H. Such reductions are important because optimization problems will generally scale poorly with problem size.

Optimization problems are solved by solving a number of decision problems. Decision problems are generally computationally *hard*. Decision problems ask yes or no questions, e.g., “is there a flow path with a flow larger than x?”. An optimization problem asks instead “what is the flow path with the largest flow?”. Any problem can be associated with one of four categories of computational complexity: Polynomial-time ($\mathcal{P}$), Non-deterministic Polynomial-time ($\mathcal{NP}$), Non-deterministic Polynomial-time Complete ($\mathcal{NP}$-C), and Non-deterministic Polynomial-time Hard ($\mathcal{NP}$-hard).

A classic example of a polynomial-time algorithm is naive matrix inversion, known to be of order n^3 (i.e., $\mathcal{O}(n^3)$) for a given $n \times n$ matrix. A decision problem, on the other hand, is considered to be in ($\mathcal{NP}$) if for any proposed solution, there is a *short certificate*.

Definition 1.1. *A **certificate** is a method to verify that a solutions provides a positive or negative response to the question at hand. A certificate is considered **short** if it is polynomial in size and can be verified in polynomial time.*

A decision problem, Q, is considered to be in $\mathcal{NP}$-C, if $Q \in \mathcal{NP}$ and *any* problem, $P \in \mathcal{NP}$ is polynomial-time reducible to Q. That is, instances of P can be reformulated as instances of Q in polynomial time. The most popular candidate of this polynomial reduction is the Satisfiability Problem, known to be in

$\mathcal{NP}$-C [20, 41]. Finally, a problem Q, is in $\mathcal{NP}$-hard, if any problem $P \in \mathcal{NP}$ is polynomial-time reducible to Q, but $Q \notin \mathcal{NP}$. If a decision problem is in $\mathcal{NP}$-C, then the corresponding optimization problem is $\mathcal{NP}$-hard. The relationship between these set of problem complexities, reflecting current understanding, is shown graphically in Figure 1.5.

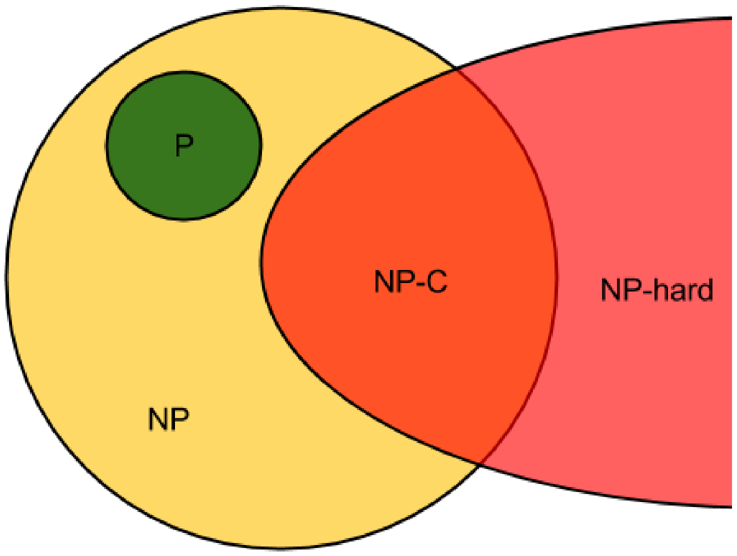

Figure 1.5: The relationship between the various types of computational complexities.

Certain optimization problems can be solved with specialty algorithms that greatly decrease solution times. However, because optimization problems are $\mathcal{NP}$-hard, no guarantee can be made *in general* regarding their scalability. Worst case scenarios result in exponential scaling with problem size. Further, in practice, MILPs experience much worse solution time behavior than do LPs. In short, reducing problem size is an important strategy for solving optimization problems more quickly.

1.4 Statement of Work

Deciding how a simulation is structured from an interactions standpoint is a delicate balance of known necessity and perceived future needs. There are basic decisions to make, such as modeling material transfer as either discrete or continuous. Discrete transfers more closely match reality and may provide insights in that regard, however they require more of their modeling apparatus due to messaging needs and other structures. More complex decisions include how one wants to determine connections between facilities, and whether such connections are assigned statically and incorporated into the simulation architecture or determined dynamically.

In his conclusions of a MIT benchmarking exercise, Guerin states that "operation of a fuel cycle model is as much art as science" [31], an opinion that likely stems from this "freedom". These simulation-engine decisions comprise the art-related portion of fuel cycle simulation, but developers have a goal of making these decisions in as informed a way as possible using domain-level knowledge with respect to our known and perceived requirements. In general, this work tries to minimize the sheer number of choices one makes in this regard, instead relying on well known and well documented practices of computer scientists and systems engineers. In short, the goal of this work focuses on extending the current state of the art in nuclear fuel cycle simulation and associated analysis.

To date, no nuclear fuel cycle simulator has been implemented using agent-based simulation design principles. Cyclus , the fuel cycle simulator in which this work is being implemented, was initially developed without a solid simulation infrastructure design principles. The initial thrust of this work comprises the development of Cyclus as an agent-based simulator. In order to do so, agent-to-agent interaction mechanisms must be defined and designed. Furthermore, a clear time-stepping procedure must be identified that provides a sufficient amount of entity-interaction opportunities to agents in a simulation.

Perhaps the least well-treated aspect in current nuclear fuel cycle simulation is resource allocation decision making. As stated previously, the vast majority of current simulators treat this process very simply: statically connect facility types *a priori*. The primary thrust of this work is the extension of the current state of the art by designing and implementing a general framework that dynamically determines the flow of resources in an arbitrary nuclear fuel cycle.

Any such mechanism must meet a number of design criteria. First, it must be fuel cycle agnostic: any possible facility connections must be supported. The mechanism must take into account the isotopic profiles of the commodities produced and consumed by agents in a simulation. Any system in which fuel recycling exists will, by definition, have some supply constraints. Therefore, the framework must support both the existence capacitated supply and demand as well as its communication between agents and with the framework. Further, constraints must be able to be influenced by sophisticated physical, chemical, and supply chain models. Further extending the state of the art, the framework must also allow for economic, social, and geographic models to inform the exchange of resources between simulation entities. Finally, the framework must support quantized transfers of resources. Nuclear reactors cores in practice are comprised of individual fuel assemblies. Any framework must support the modeling of individual fuel assemblies, enabling a high level of simulation detail as well as nonproliferation analyses.

It is the goal of this corpus of effort to design and implement such a mechanism using agent-based supply chain simulation techniques and mathematical programming methods. Once a supply-demand framework is developed, its performance must be analyzed. Fuel cycle simulation can require varying levels of computational fidelity. Some scoping studies may wish to sample a large option space with low fidelity, while others may wish to sample a small option space with high fidelity. The performance trade-off between feasible and optimal solutions to resource flows must be understood. Because the Cyclus ecosystem is still nascent, sophisticated agent models have yet to be developed. Accordingly, a methodology for generating instances of nuclear fuel cycles is required. A large collection of instances must then be executed with all available supply-demand solution techniques, those that find optimal solutions and those that report some best-guess feasible solution.

Upon completion of this work, fuel cycle simulation modelers and analysts will be provided a robust tool that greatly increases the fidelity and flexibility with which arbitrary fuel cycles can be modeled. The state of the art of NFC simulation will be furthered, and novel scenarios that involve sophisticated interactions such as the competition for resources, dynamic commodity consumption, and geopolitical relationships can finally be supported.

2 Simulation and Agent Based Modeling in Cyclus

Developing a simulator of any complex system is an involved process requiring a solid methodological base. A reasonable approach is to define precisely the simulation framework, including a definition of how time moves forward and what events can occur. This chapter lays the foundation for the simulation of nuclear fuel cycles in Cyclus. Section 2.1 begins the discussion by broadly describing methodological principles on which Cyclus has been developed. The use of agent-based modeling techniques is presented in section 2.2, and an agent deployment methodology with a proof-of-principle benchmark is presented. Finally, section 2.3 treats the most complex simulation interaction in Cyclus, Dynamic Resource Exchange (DRE), describing its methodology, discussing its implementation, and presenting a set proof-of-principle results.

2.1 Simulation Principles

Cyclus is designed to dynamically model the flow of resources and deployment of facilities in the Nuclear Fuel Cycle (NFC). As such, Cyclus is a *simulator* which models the NFC as a *system*. System simulation is a rich field of study, spanning a variety of disciplines, as described in section 1.3.1.

By Law's definition [40], Cyclus is a dynamic, discrete-event simulation that uses a fixed-increment time advance mechanism. In general, fixed-increment time advance simulations assume a time step (Δt). Further they assume that all events that would happen during a time occur simultaneously at the end of the time step. This situation can be thought of as an event-based time advance mechanism, i.e., one that steps from event to event, that executes all events simultaneously that were supposed to have occurred in the time step.

A Cyclus simulation models a collection of *entities* which either trade resources, manage other entities, or perform both actions. The most basic *entity* in a Cyclus simulation is a Facility. Facilities can be used to model processes with arbitrary levels of physical fidelity, and can interact with the simulator and other entities with arbitrary levels of behavioral fidelity. As such, Cyclus can also be described as an *agent-based model* (ABM). Accordingly, the *entities* in a given simulation can be interchangeably referred to as *agents*. Cyclus has an additional notion of an *archetype*. An archetype is the implementation of an entity, whereas an agent is the *in situ* instantiation of a entity. The remainder of this document will use the term archetype when referring to the implementation of an entity and will use the term agent when referring to an entity acting in a simulation.

2.1.1 Events

Two key types of events occur in every Cyclus simulation:

- agent entry into and exit from the simulation
- the exchange of resources between agents

Agent entry and exit events are scheduled by another managing agent, or are scheduled as an initial condition to the simulation. The managing agent and managed agent form a parent-child relationship. Upon entering the simulation, the child entity is constructed and notified of its entry; the parent is then notified. Upon exiting the simulation, the parent is notified; the child entity is then notified and deconstructed. Unlike many of the simulators described in section 1.2, the Cyclus simulation kernel naturally treats each agent individually, rather than grouping agents by an attribute and treating like-facilities in an aggregate manner.

While the determination of supply and demand is complex and described further in section 2.3, the execution of resource exchange is rather straightforward and a primary event in a Cyclus simulation. When an agent's demand for a resource is matched with another agent's supply of a resource by the Cyclus kernel, a transfer is initiated. Each transfer is treated as discrete, individual trade between two agents.

2.1.2 Timesteps

Simulation entities can have arbitrarily complex state which is dependent on the results of resource exchange and the present discernible status of other agents in the simulation at a given time step. Furthermore, resource exchange necessarily must involve all existing agents in the simulation. Therefore, a well-defined timestep, incorporating agent entry, exit, resource exchange, and agent response to system state must be defined. Cyclus implements a timestep mechanism that deviates slightly from Law's description of fixed-increment time advance by preserving a specific ordering of *event triggers*. Importantly, the following invariant is preserved: *any agent that exists in a given time step experiences the entire time step execution stack*.

This leads to the following *phases* of time step execution:

- agents enter simulation (Building Phase)
- agents respond to current simulation state (Tick Phase)
- resource exchange execution (Exchange Phase)
- agents respond to current simulation state (Tock Phase)
- agents leave simulation (Decommissioning Phase)

The Building, Exchange, and Decommissioning phases each include critical, core-based events, and are called *Kernel* phases. The Tick and Tock phases do not include core-based events, and instead let agents react to previous core-based events and inspect core simulation state. Furthermore, they are periods in which agents can update their own state and are accordingly considered *Agent* phases.

Technically, whether agent entry occurs simultaneously with agent exit or not does not matter from a simulation-mechanics point of view, because the two phases have a direct ordering. It will, however, from the point of view of module development. It is simpler to think of an agent entering the simulation and acting in that time step, rather than entering a simulation at a given time and taking its first action in the subsequent time step.

In the spirit of Law's definition of a fixed-increment time advance mechanism, there is an additional important invariant: *there is no guaranteed agent ordering of within-phase execution*. This invariant allows for:

- a more cognitively simple process

- paralellized implementation

2.2 Agents and Agent Based Modeling in Cyclus

Cyclus has worked to formally move from a modeling paradigm that does not differentiate between individual facilities, as has been the case historically in FCS, to one that does. Modeling individual facilities in the NFC requires a nuanced approach to determine facility behavior, because such behavior can depend on intricate physical parameters of resources in the simulation as well as complex social-behavioral models of facility interaction.

Agent-based models are defined primarily by two concepts: agents and the simulation environment. Agents in Cyclus are designed to be able to incorporate arbitrary complexity in both physical process models as well as behavioral models. A three-tiered taxonomy has been developed to achieve this aim, specializing agents as either Facilities, Institutions, or Regions. Section 2.2.1 fleshes out a discussion of this design.

The simulation environment in Cyclus is defined by supply and demand. There is a notion of supply and demand for facility capacity. For example, there can be a demand for power production which drives the deployment of power producing facilities. There is also a notion of supply and demand for resources.

Sufficiently treating resource supply and demand is the primary argument for implementing Cyclus as an ABM simulator. In the NFC, resource supply and demand is a function of both resource quantity and quality, that is, the isotopic composition of material resources. In the extreme in which a high level of detail is required in the notion of resource quality, e.g. tracking an arbitrary number of isotopes, adopting techniques that allow decision-making based on that level of detail is desirable. Modeling the nuclear fuel cycle represents such a level of detail. For example, even in the case of a once-through fuel cycle, many reactors of the same type (e.g., PWRs), may require different resource qualities (i.e., Uranium enrichment). As the complexity of a quality metric increases, an aggregate approach becomes less desirable as it loses such detail through aggregation. Furthermore, by disassociating simulation logic from entity logic, agents of arbitrary fidelity levels can be used in the same simulation. For instance, a reactor agent that tracks a small subset of isotopes can be used in tandem with a reactor agent that tracks a large set of isotopes.

In summary, the arbitrary levels of complexity that can be required for a flexible NFC simulator suggests that ABM is a reasonable tool to use. The remainder of this section describes how agents are provided agency in Cyclus and specifically how agents interact with respect to supply and demand of facility capacity. A proof-of-principle benchmark comparison to a systems dynamics simulator is shown in section 2.2.3. Agent interaction with respect to supply and demand of resources is more complicated and therefore treated separately in section 2.3.

2.2.1 Agent Taxonomy

The Cyclus kernel implements a basic `Agent` class that provides the minimal interface for agents to be instantiated within a simulation. A `Trader` interface provides a communication layer required for agents to be included in the exchange of resources. Three useful derived classes are provided to be used as basic abstractions of entities in the NFC. `Facility` agents in Cyclus implement both interfaces, while `Institution` and `Region` agents implement only the `Agent` interface. A summary of the conceptual placing of each archetype in a Cyclus simulation is provided below.

2.2.1.1 Facilities

Facilities in Cyclus are either consumers or suppliers of commodities, and some may be both. Supplier agents are provided agency by being able to communicate to the market-resolution mechanism a variety of production capacity constraints in second phase of the information gathering methodology. Consumer agents are provided agency by being able to assign preferences among possible suppliers based on the supplier's quality of product. Because this agency is encapsulated for each agent, it is possible to define strategies that can be attached or detached to the agents at run-time. Such strategies are an example of the Strategy design pattern [60].

2.2.1.2 Institutions

Institutions in Cyclus manage a set of facilities. Facility management is nominally split into two main categories: the commissioning and decommissioning of facilities and supply-demand association. The goal of including a notion of institutions is to allow an increased level of detail when investigating regional-specific scenarios. For example, a consumer facility may prefer to be supplied by a supplier facility in its institution rather than one associated with a different institution. Furthermore, there are international governmental organizations, such as the IAEA, that have proposed managing large fuel cycle facilities that service many countries in a given global region. A fuel bank is an example of such a facility. Accordingly, institutions in Cyclus are able to augment the preferences of supplier-consumer pairs that have been established in order to simulate a mutual preference to trade material within an institution. Of course, situations arise in real life where an institution has the capability to service its own facilities, but choose to use an outside provider because of either cost or time constraints. Such a situation is allowed in this framework as well. It is not clear how such a relationship should be instantiated and to what degree institutions should be allowed to affect their managed facilities' preferences. This issue lies squarely in the realm of simulation design decisions, part of the *art* of simulation. Accordingly, through the course of research, the possible design space will be analyzed in order to determine best practices for this type of design.

2.2.1.3 Regions

Regions in Cyclus provide the forcing function for simulations by requiring that certain parameters be met, e.g., power capacity, fuel cycle service capacity, etc. For example, in the case of nuclear power capacity, a region knows that it needs additional reactors to be built, but leaves the building of those reactors to the institutions that operate in the region. It is important to note here that this abstraction allows for different deployment algorithms to be tested and exchanged in the Cyclus framework without necessitating changes to the simulation engine, as is the case with other simulators described in section 1.2.

Regions, like Institutions, are able to affect preferences between supplier-consumer facility pairs in the market information gathering process. The ability to perturb arc preferences between a given supplier and a given consumer allows fuel cycle simulation developers to model relatively complex interactions at a regional level such as tariffs and sanctions.

2.2.2 Methods of Agency

Agency is provided in two primary modes: determining facility deployment and informing resource exchange mechanisms.

Facility deployment involves some combination of an `Institution` agent, a `Facility` agent, and a `Region` agent. `Institution` agents represent a simulation entity abstraction that can deploy `Facility` agents. `Region` agents represent a simulation entity abstraction that have a demand for certain commodities that `Facility` agents provide, for example, reactor-like `Facility` agents provide electrical power.

`Facility` agents are further provided agency by informing market mechanisms of the supply and demand of resource quantity and quality. Cyclus initially used a simple interface and algorithm for determining resource transactions. Individual markets were defined as agents themselves much like `Facility`, `Institution`, and `Region` agents. Many limitations were identified at the time, however, and the market-as-agent approach was eventually abandoned. An enumeration of the observed limitations is described further in section 2.2.4.

The primary source of agency is provided to `Facility` agents through the `Trader` interface in order to negotiate the quantity and quality of potential resource transactions. `Region`, `Institution`, and `Facility` agents are then provided agency in the negotiation of preferences of potential transactions, where preference is a proxy for price.

2.2.3 Proof of Principle

Agents were developed to show an initial proof of principle that fuel cycle simulation can be implemented using an agent-based modeling methodology. By definition, dynamic simulators model the deployment of facilities and measure the flow of resources between facilities in the system over time. In the extreme case of unconstrained supply and no competition for resources, resource exchange decisions can be made arbitrarily. In such cases, therefore, only facility deployment agency is required. An initial benchmark case was performed to confirm expected deployment behavior and basic resource routing.

2.2.3.1 Benchmark Cases

The INPRO Business As Usual (BAU) benchmark [6] for the once-through fuel cycle was chosen for three reasons. First, it was the simplest benchmark that demonstrated deployment behavior. Second, no supply or demand constraints were present, so a basic supply-demand framework would suffice. Finally, results from another fuel cycle simulation code, VISION [36], was available for comparison. The INPRO BAU benchmark identified two cases, high electricity demand and moderate electricity demand, as shown in Fig. 2.1. Both cases require that demand met by a composition of 94% Light Water Reactors (LWRs) and 6% Heavy Water Reactors (HWRs). LWRs are fueled with 4% by weight UO_2 while HWRs use natural Uranium fuel.

The goal of this proof-of-principle study was to showcase the capability for a developer to generate the required Facility, Institution, and Region archetypes, and that such archetypes could be deployed in

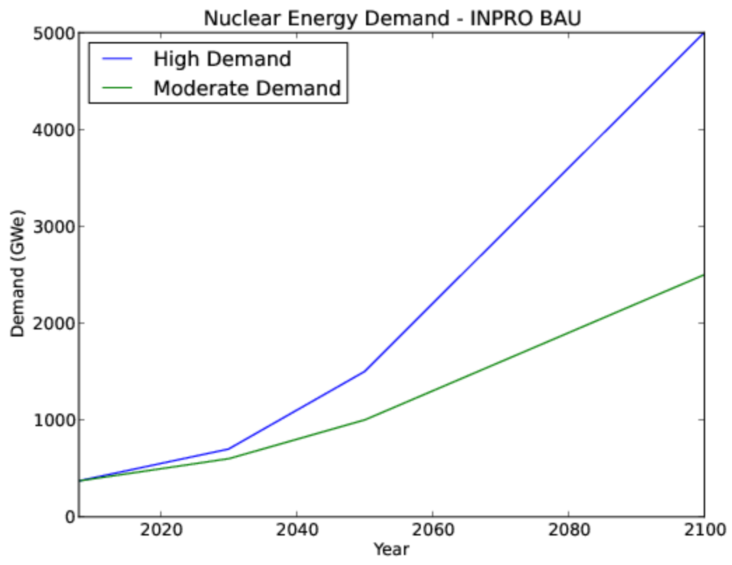

Figure 2.1: The energy demand specification for the INPRO BAU scenarios.

the Cyclus simulation framework and generate satisfactory results. Comparison metrics are based on similar metrics used in the origin INPRO benchmarking exercise, including deployment patterns, natural uranium consumed, and used fuel produced by all reactors.

2.2.3.2 Agent Archetypes Developed

Each implemented agent is available in the Cycamore repository [61].

GrowthRegion

The `GrowthRegion` is a Region archetype developed to assist in facility deployment logic. The `GrowthRegion` takes as input a listing of commodities for which it has a demand. For example, the `GrowthRegion` agents in this benchmark demand electrical power. The demand curves for commodities is defined by symbolic functions. Currently, linear functions, exponential functions, and piece-wise combinations of both are supported.

At any time step in which there exists a demand gap, i.e., there exists more demand than supply, a build decision is made. This decision is modeled as the following minimum cost facility deployment integer program:

$$\min_{n} \quad \sum_{i \in I} c_i * n_i \tag{2.1a}$$

$$\text{s.t.} \quad \sum_{i \in I} \phi_i * n_i \geq \Phi \tag{2.1b}$$

$$n_i \in [0, \infty) \quad \forall \; i \in I \tag{2.1c}$$

$$n_i \quad integer \quad \forall \; i \in I \tag{2.1d}$$

where Φ is the unmet demand, I is the set of facilities capable of meeting the demand, and, for each facility in I, c_i is the cost of building, and ϕ_i is the nameplate capacity. Finally, n_i is the optimized number of facilities to build of type i.

ManagerInst

The `ManagerInst` is an Institution archetype also developed to assist in facility deployment. While the `GrowthRegion` places a build order, the `ManagerInst` fulfills the order. Further, the `ManagerInst` determines the set of facilities, I, shown in in Eqn. 2.1, which can be built. Note that the set I can change over time. Once a deployment decision is made, the `GrowthRegion` makes a facility deployment request of the `ManagerInst` which then deploys the chosen facility.

BatchReactor

While a reactor model existed prior to this work, it did not provide the functionality to interchange *batches* of fuel, as required by the INPRO benchmark. A batch of fuel is a fraction of a full reactor core that is extracted and replaced when a reactor is refueled. In general, LWRs replace between a third and a quarter of their assemblies during refueling based on the fuel management scheme used.

The `BatchReactor` used in this work had configurable properties as displayed in Table 2.1. The values used based on the defined INPRO benchmark are described in Table 2.2.

EnrichmentFacility

The `EnrichmentFacility` archetype was developed to provide enrichment-related output for the simulation, namely the amount of separative work units (SWU) and natural uranium used during a given

Parameter	Description
Process Time	Active fuel time in the reactor
Refuel Time	Time to refuel the reactor
N Batches	Number of batches in the reactor
Batch Size	Quantity of a batch
Power Capacity	Nameplate Capacity for Power
Power Cost	Cost to build a new reactor

Table 2.1: Configurable input for the `BatchReactor` archetype.

Parameter	LWR Value	HWR Value
Process Time	10	10
Refuel Time	2	2
N Batches	4	4
Batch Size	7.87E4	1.39e5
Power Capacity	1000	600
Power Cost	1000*	600*

(*) Note that the Cost used is arbitrary and set equal to the capacity so that a minimum capacity is built per Eqn. 2.1.

Table 2.2: Configurable input values for reactors used in the INPRO once-through benchmark.

time step. For the INPRO cases, it can be defined quite simply using the values shown in Table 2.3. The feed assay and product assay, both required for determining output metrics, are defined by the isotopic compositions of resource input, i.e., natural uranium, and resource output, i.e., the isotopic composition of requested fuel.

Parameter	Description	Values
Input Recipe	A description of input isotopics	Natural Uranium
Tails Assay	The U-235 assay of tails.	0.003

Table 2.3: Configurable input values for the `EnrichmentFacility` used in the INPRO once-through benchmark.

2.2.3.3 Results

In aggregate, Cyclus performed well relative to the other benchmarks. Fig. 2.2 shows the reactor deployment curves for each simulator for the moderate growth scenario while Fig. 2.3 shows reactor deployment for the high scenario. The slight differences are attributed to VISION's look-ahead functionality which builds the required facilities one time step after they are needed, whereas Cyclus builds facilities on the timestep in which they are needed. One can observe that a simple one timestep translation will result in

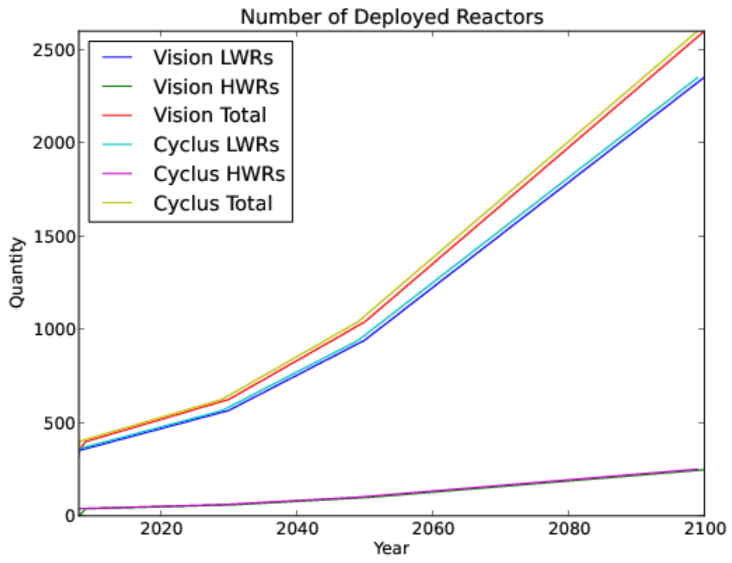

Figure 2.2: The reactor deployment schedule by reactor type for the moderate demand scenario.

identical output.

Cumulative natural uranium utilization curves for the moderate and high cases are shown in Figures 2.4 and 2.5, and cumulative used fuel inventory curves are shown in Figures 2.6 and 2.7. Slight discrepancies are noted between Cyclus and VISION. These discrepancies are attributed to the implementation of core batch recycling in each of the respective codes. The differences between Cyclus and VISION are magnified because the curves show cumulative metrics. In other words, results at time t_1 are added the results at time t_2 and so on. Therefore, a series of small discrepancies appears to be compounded by using this metric. It is not the metric of choice for general comparisons, but has been used because it was the metric of choice of the benchmark exercise. It is not immediately obvious why there is a greater discrepancy regarding output fuel quantities than natural uranium utilization. Further benchmarking exercises with support from a VISION developer would be required to fully investigate the issue.

2.2.4 Multiple Market Limitations

The proof of principle benchmark described in section 2.2.3 utilized the agency provided for facility deployment rather than the agency provided for both deployment and resource exchange. In general informing resource exchange regarding quantity and quality of resources as well as socioeconomic effects

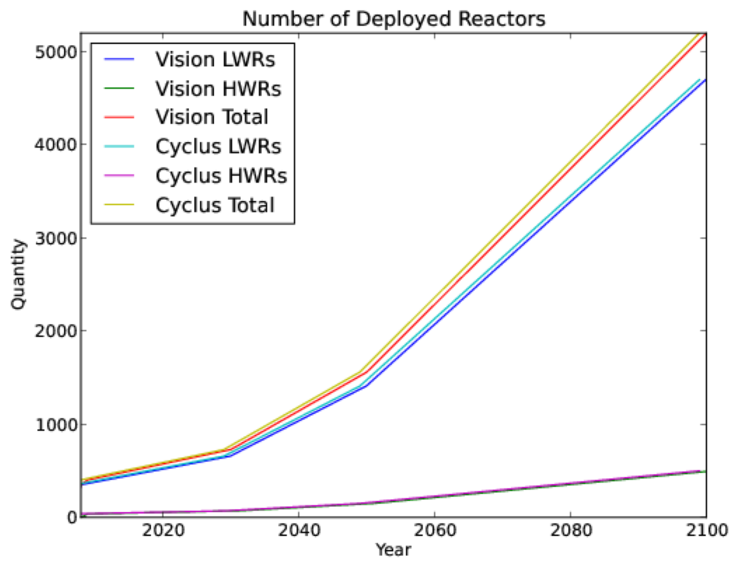

Figure 2.3: The reactor deployment schedule by reactor type for the high demand scenario.

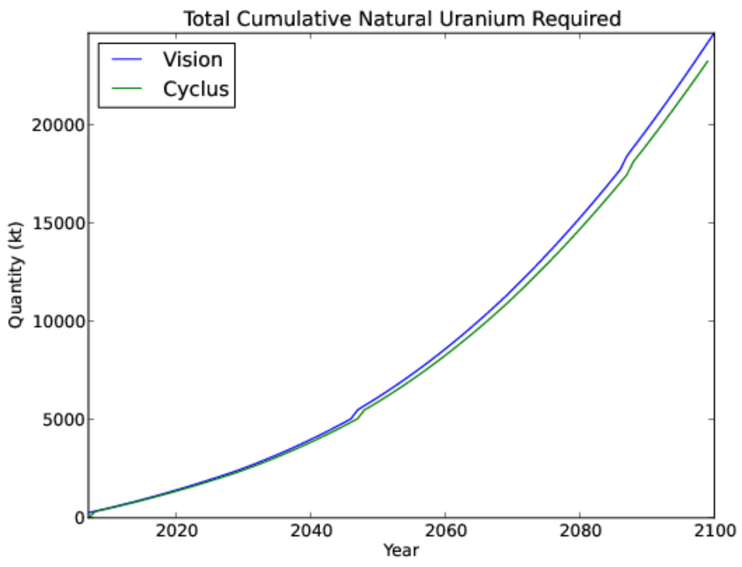

Figure 2.4: The total natural uranium used for the moderate demand scenario.

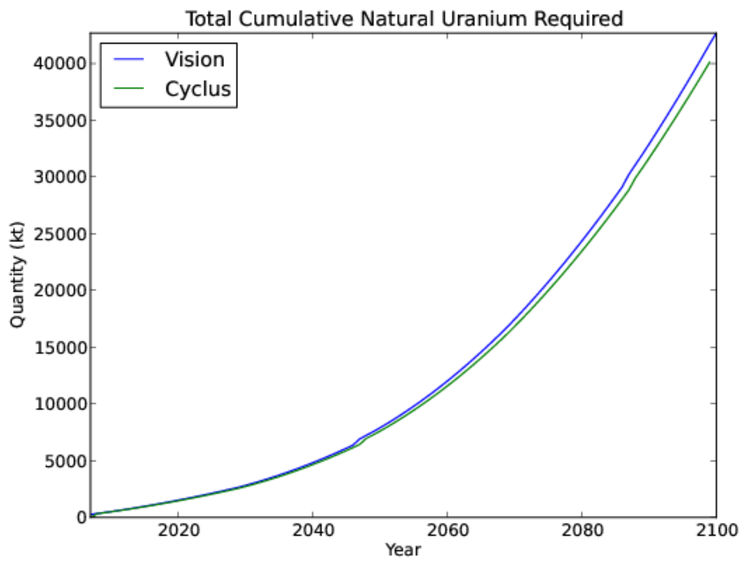

Figure 2.5: The total natural uranium used for the high demand scenario.

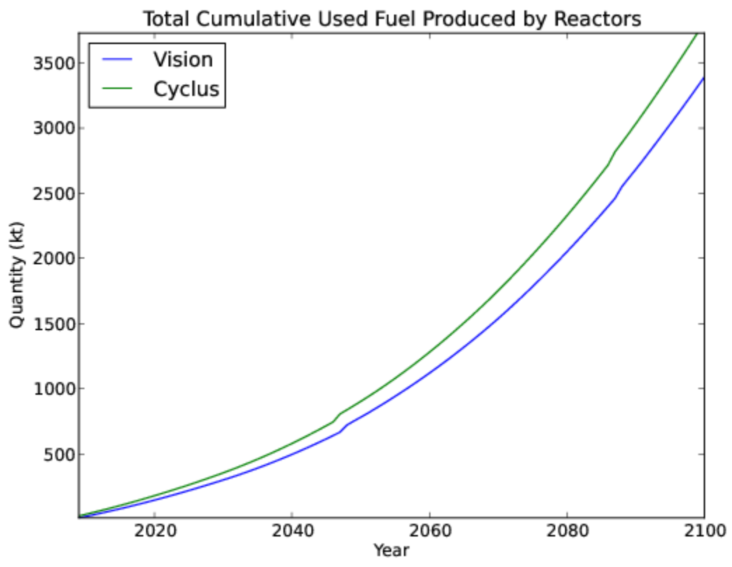

Figure 2.6: The amount of used fuel produced for the moderate demand scenario.

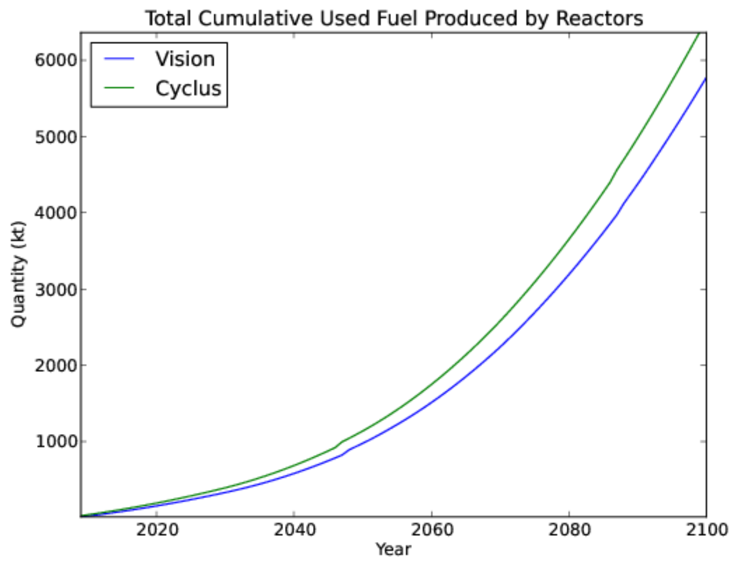

Figure 2.7: The amount of used fuel produced for the high demand scenario.

is a hard problem.

Cyclus was originally designed to use an additional agent archetype called a `Market`. `Markets` were envisioned to represent markets for specific commodities. For example, the simulation described in section 2.2.3 used three commodity markets: natural uranium, enriched uranium fuel, and used fuel. This approach is valid in the absence agent-specified supply or demand constraints and competition for resources in multiple markets (e.g., for fungible resources). However, the inclusion of either of these features requires a much more involved process.

If supply or demand constraints are to be modeled, each associated `Market` agent must have both a corresponding communication interface and an implementation that accounts for such constraints. While quantity constraints are not unreasonable to implement and support, quality constraints are much more difficult. Furthermore, communicating such constraints is difficult. Whereas the `Market` agent can implement a solver algorithm, constraints are more naturally defined by the trader interacting with the `Market` agent. For example, consider the enriched uranium market used in section 2.2.3. While the simulation used an agent abstraction for an enrichment facility and fuel fabrication plant, another simulation may wish to model facilities that downblend HEU, rather than enrich LEU. Such a process will have different constraints. Importantly, those constraints are a function of the `Facility` archetype, not of a `Market` archetype.

Assuming that supply or demand is constrained by either resource quantity or quality, competition for the resource in question can arise. When competition for resources exist, there must be some mechanism that determines which transactions are to be executed, i.e., which agents should trade which resources. Determining supply and demand under competition is a well studied problem with many possible formulations and solution frameworks.

Fungibility is the property of a good or commodity to be *capable of being substituted in place of one another* [45]. For example, a light water reactor generates power by fissioning nuclei in the thermal energy spectrum. Whether those nuclei are ^{239}Pu , ^{235}U , or ^{233}U makes little difference from a power generation perspective. In other words, those nuclei are *fungible* for light water reactors, given some safety and cycle length considerations. A similar issue arises from a supplier's perspective. Consider a MOX fuel supplier and two requesters: a fast reactor and a thermal reactor. Given the isotopic makeup of Plutonium in the MOX fuel, the supplier's fuel could be potentially be used in either reactor type. Again, Plutonium in this example is a fungible resource. Importantly, the notion of fungibility in a NFC context can refer to both individual isotopes, collections of isotopes, or complete fuel forms. Accordingly, a facility may demand multiple fungible commodities, which must be accounted for by a given market clearing mechanism.

The one-market-per-commodity approach does not treat competition, constrained supply and demand, and fungibility particularly well. Constraints are handled poorly because constraints are best determined by the supplying and demanding agents rather than the market. Separated markets must, of course, be solved separately. Therefore, competition and fungibility are treated poorly, because information involving multiple commodities is not taken into account during the solution of a single market. Accordingly, a solution framework and methodology that incorporates agent querying of supply, demand, and constraints and resolves markets in parallel is required to properly treat resource exchange in the nuclear fuel cycle.

2.3 Dynamic Resource Exchange

Dynamic Resource Exchange (DRE) is the functional bedrock on which Cyclus simulations are built. It defines the interaction mechanisms and methodologies for agents, specifically agents whose archetypes have implemented the `Trader` interface. This section begins by providing a motivating problem statement in section 2.3.1. It then details the methodology for querying supply and demand during the information gathering phase of the DRE in section 2.3.2. The solution phase, in which the defined DRE is translated into a form of the Multicommodity Transportation Problem (MCTP) and solved, is then described in section 2.3.3. Finally, two proof of principle simulations with novel fuel cycle DREs are presented in section 2.3.5.

This section represents the culmination of significant previous effort [25, 27, 28]. What follows constitutes the refinement of previous descriptions of the DRE methodology with lessons learned from initial implementation and usage.

2.3.1 Problem Statement

As a next-generation nuclear fuel cycle simulation framework, Cyclus maintains a primary goal of modeling flexibility. As facility, institutional, and regional archetypes are proposed, they should be relatively easily implemented and utilized in the Cyclus simulation framework. Furthermore, the level of modeling abstraction for different facilities in a fuel cycle will be different based on the needs of archetype developer. Any supply-demand resolution framework, therefore, must be able to support arbitrary facilities.

As stated previously in section 2.2.4, a number of considerations must be taken into account in such a framework. Supply and demand must be able to be solved globally at any given time step. Therefore, the framework must support an arbitrary number of facilities. Further, resources must be able to be treated in a fungible manner. The framework must be able to handle arbitrary resource definitions and incorporate arbitrary, agent-defined constraints.

In order to address each of these concerns, the concept of a Dynamic Resource Exchange (DRE) was developed and implemented. That process was motivated by the following problem statement:

> If facilities are treated as individual black boxes and connections between facilities are determined dynamically, how does one match suppliers with consumers considering quantity and quality-based supply constraints, quantity and quality-based demand constraints, supply response to quality-based demands, and issues of fungibility?

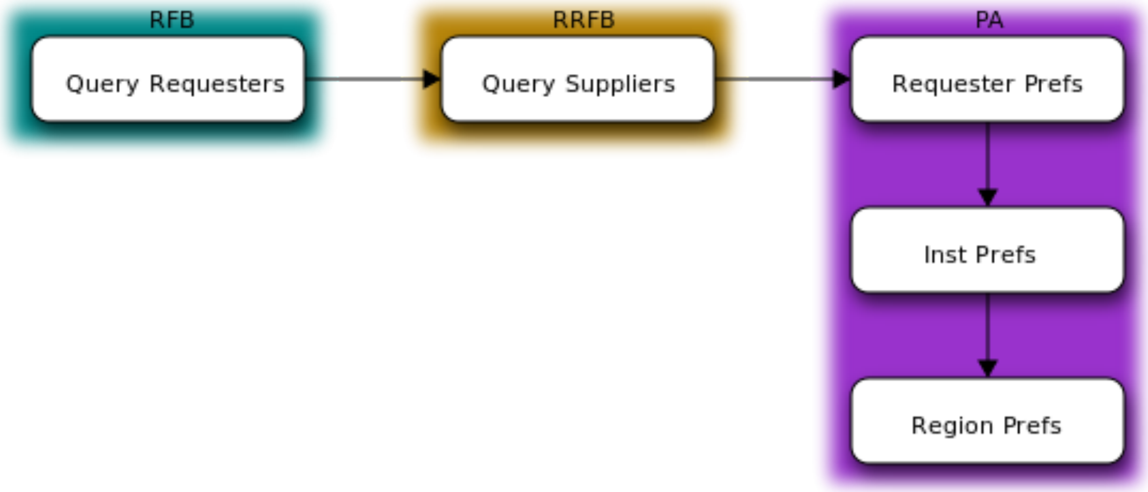

Figure 2.8: Schematic illustrating the DRE's information gathering procedure.

2.3.2 Information Gathering

The DRE begins at any given time step with three *phases*, the terminology of which is influenced from previous supply chain agent-based modeling work [37]. Importantly, this information-gathering step is agnostic as to the supply-demand matching algorithm used, it is concerned only with querying the current status of supply and demand in the simulation. The collective information gathering procedure is shown in Figure 2.8.

The first phase allows consumers of commodities to denote both the quantity of a commodity they need to consume as well as the target isotopics, or quality, by *posting* their demand to the market exchange. This posting informs producers of commodities what is needed by consumers, and is termed the *Request for Bids* (RFB) phase. Consumers are allowed to over-post, i.e., request more quantity than they can actually consume, as long as a corresponding capacity constraint accompanies this posting. Requests can be denoted as *exclusive*. An exclusive request is one that must either be met in full or not at all. Exclusive requests allow the modeling of quantized, packaged transfers, e.g., fuel assemblies.

Consumers are allowed to post demand for multiple commodities that may serve to meet the same combine capacity. For example, consider an LWR that can be filled with MOX or UOX. It can post a demand for both, but must define a preference over the set of possible commodities that can be consumed. Such requests are termed *mutual requests*. Another example is that of an advanced fuel fabrication facility, i.e., one that fabricates fuel partially from separated material that has already passed through a reactor. Such a facility can choose to fill the remaining space in a certain assembly with various types of fertile material, including depleted uranium from enrichment or reprocessed uranium from separations. Accordingly, it could demand both commodities as long as it provides a corresponding constraint with respect to total consumption. A set of exclusive requests may also be grouped as mutual requests, in which case the set is termed *mutually exclusive*.

At the completion of the RFB phase, the market exchange will have a set of request portfolios. Each each portfolio consists of a set requests. Arbitrary constraints over the set of requests can be provided that are functions of quantity or quality. Each request may have an associated preference. For requests that mutually satisfy a given demand, a preference distribution over those requests informs the solver as to which should be satisfied first, given constraints. Finally, each request portfolio has a specific quantity associated with it.

The second phase allows suppliers to *respond* to the set of request portfolios, and is termed the *Response to Request for Bids* (RRFB) phase (analogous to Julka's Reply to Request for Quote phase [37]). Each request portfolio is comprised of requests for some set of commodities. Accordingly, for each request, suppliers of that commodity denote production capacities and an isotopic profile of the commodity they can provide. Suppliers are allowed to offer the null set of isotopics as their profile, effectively providing no information. Suppliers are also allowed to denote responses as exclusive, as is done in the RFB phase. Supply responses can also be grouped into mutual responses, and sets of responses may be mutually exclusive. This functionality again supports the notion of quantized orders, e.g., in the case of fuel assemblies.

A supplier may have its production constrained by more than one parameter. For example, a processing facility may have both a throughput constraint (i.e., it can only process material at a certain rate) and an inventory constraint (i.e., it can only hold some total material). Further, the facility could have a constraint on the quality of material to be processed, e.g., it may be able to handle a maximum radiotoxicity for any given time step which is a function of both the quantity of material in processes and the isotopic content of that material. Multiple of such constraints are allowed. At the completion of the RRFB phase the possible connections between supplier and producer facilities, i.e., the arcs in the graph of the transportation problem, have been established with specific capacity constraints defined both by the quantity and quality of commodities that will traverse the arcs.

The final phase of the information gathering procedure allows consumer facilities to adjust their set of preferences and for managers of consumer facilities to affect the consumer's set of preferences. Accordingly, the last phase is termed the *Preference Adjustment* (PA) phase. By allowing facility managers, i.e., a facility's institution and region, to also adjust preferences, socio-economic models are allowed to inform the exchange of resources. For example, a region can detect a trans-regional trade between one of its facilities and a facility in another region. If a tariff model is employed, the trade preference and be diminished or even removed.

For facilities, preference adjustments occurs in response to the set of responses provided by producer facilities. Consider the example of a reactor facility that requests two fuel types, MOX and UOX. It may get two responses to its request for MOX, each with different isotopic profiles of the MOX that can be provided. It can then assign preference values over this set of potential MOX providers. Another prime example is in the case of repositories. A repository may have a defined preference of material to accept based upon its heat load or radiotoxicity, both of which are functions of the quality, or isotopics, of a material. In certain simulators, limits on fuel entering a repository are imposed based upon the amount of time that has elapsed since the fuel has exited a reactor, which can be assessed during this phase. The time constraint is, in actuality, a constraint on heat load or radiotoxicity (one must let enough of the fission products decay). A repository could analyze possible input fuel isotopics and set the arc preference of any that violate a given rule to 0, effectively eliminating that arc.

2.3.3 The Nuclear Fuel Cycle Transportation Problem

Supply and demand in a nuclear fuel cycle context is inherently a multicommodity problem. A light water reactor can be fueled by both UOX and MOX fuel, for instance. How it is fueled is a result both of fuel availability and associated preferences. Allowing for complex physical and chemical constraints on both processes and inventories, as well as including economics-based approaches for determining exchange preferences is a complicated affair. Determining the optimum solution to such a system is even more complicated. Accordingly, sophisticated tools in both the operations research and agent based modeling realms have been leveraged to accomplish the task.

An instance of supply and demand defined by the DRE information gathering step can be solved in a variety of ways. It can be cast to a constrained, bipartite network, and any heuristic that provides a feasible solution to such networks are valid. The system can be solved optimally, however, by formulating the system as a mathematical program. This section describes a Multicommodity Transportation Problem variant used for this approach, entitled the *Nuclear Fuel Cycle Transportation Problem* (NFCTP). A linear program (LP) formulation and a mixed-integer linear program (MILP) formulation are provided. A greedy heuristic is also designed and implemented.

The LP formulation can be solved quickly, but allows split orders. In other words, the LP formulation solves a relaxation of the defined instance that does not take into account *exclusive* requests or bids. The nuclear fuel cycle deals with bundled orders, such as nuclear fuel assemblies, thus this modeling paradigm is only an approximation. The MILP provides a more realistic exchange, but can take much longer to solve.

2.3.3.1 Terminology

Objects and data structures generated in the information gathering procedure are used in the formal definition of the NFCTP. Each portfolio can be considered separately. The set of supply portfolios is denoted as S and the set of request portfolios is denoted as R, and each agent may have multiple portfolios in a given exchange. Each supply portfolio is comprised of s_M supply nodes, and each request portfolio is comprised of r_N nodes. The set of supply nodes is denoted I, and the set of request nodes is denoted J. The total number of supply and request nodes is then

$$|I| = \sum_{s \in S} s_M \tag{2.2}$$

and

$$|J| = \sum_{r \in R} r_N . \tag{2.3}$$

Each portfolio has a set of commodities, H, associated with it. These are denoted H_s for supply portfolios and H_r for request portfolios. Furthermore, each portfolio has a set of constraints, K, associated with it. Each constraint has a constraining value, b_s^k and b_r^k, respectively. Additionally, each unique combination of portfolio and constraint has an associated *constraint coefficient conversion function*, denoted β_s^k for supply portfolios and β_r^k for request portfolios. Each constraint coefficient conversion function takes as an argument a proposed resource $q_{i,j}$. Request portfolios are provided a quantity constraint by default for which coefficients are unity. For a set of *mutual requests*, M, where each request has a request quantity, x_m, the coefficient is defined by the ratio between the the average request quantity over all mutual requests and x_m

$$\beta_{r,m} = \frac{\bar{x_M}}{x_m}. \tag{2.4}$$

The constraint conversion functions are utilized in the NFCTP by applying them to the proposed resource transfers, creating constraint coefficients. Coefficients for supply constraints are defined as

$$a_{i,j}^k = \beta_s^k(q_{i_j}). \tag{2.5}$$

Coefficients for request constraints are defined as

$$a_{j,i}^k = \beta_r^k(q_{i_j}). \tag{2.6}$$

Finally, for each supply-request node pair, there is an associated preference, $p_{i,j}$. The set of all preferences is denoted P. Similarly, flow between a node pair is denoted $x_{i,j}$, and the set of all flows is denoted X. The possible flow on an arc is provided an upper bound by the request node quantity, $\tilde{x_j}$.

2.3.3.2 Exchange Graph

Upon completion of the information gathering phase, a *bipartite* network is formed. This network is called the *exchange graph*. The network consists of sending (bid) nodes, I, and receiving (request) nodes, J. For each request node, j, there may be many bid nodes; however, there is a one-to-one mapping between bid nodes and request nodes. In other words, a given bid node, i, is a unique response to a request node, j. An example of a bare exchange graph graph is shown in Figure 2.9.

In the bipartite graph, portfolios act as partitions that group nodes together. Node groups share common constraints, and request node groups share a common notion of satisfiable quantity, i.e., a default mass-based constraint. An example of a partitioned exchange graph is shown in Figure 2.10.

Because of defined constraints, there may not be sufficient supply in the simulated exchange. To ensure a feasible solution, an unconstrained false supply node is added to the exchange graph. Additionally, false nodes are added to each request portfolio and are connected to the false supply source. These arcs

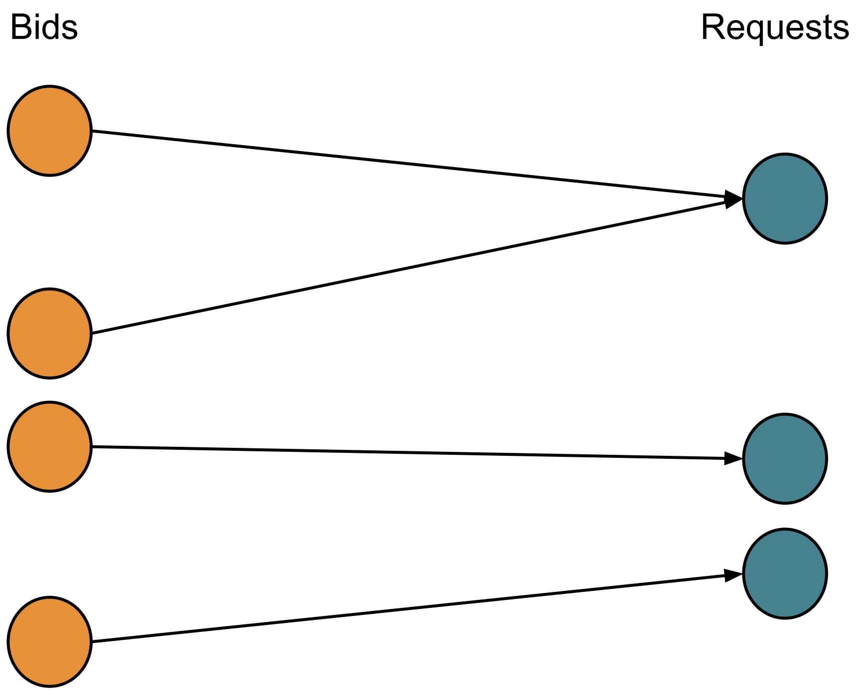

Figure 2.9: A bare example exchange with supply nodes colored orange on left and request nodes colored blue on right. As shown, there can be multiple supply nodes connected to a request node, but each supply node corresponds uniquely to one request node. It is a specific response to that request, as outlined in the RRFB phase.

are denoted as *false arcs*. The preferences given to each false arc, p_f, is defined to be lower than the lowest preference in the system, P.

$$p_f < \min P \tag{2.7}$$

The total number of arcs in the system, $|A_t|$, is then increased by the number of request portfolios, i.e.,

$$|A_t| = |A| + |R| \tag{2.8}$$

Because preferences are defined as in Equation 2.7, any false arc will only be engaged if no other possible arc can be engage, due to capacity constraints. If any flow is assigned to false arcs after the exchange graph is solved, that flow is ignored when initiating transactions. Figure 2.11 shows a fully

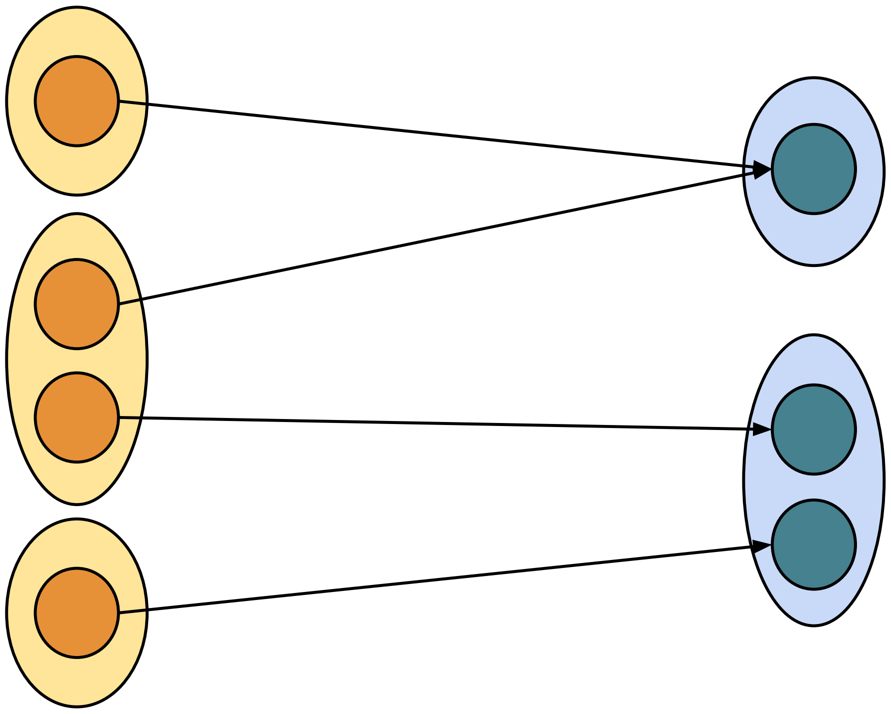

Figure 2.10: The same exchange shown in Figure 2.9 with the inclusion of portfolio partitions. In this example, there are three supplier agents and two consumer agents. The second consumer has two requests (for different commodities) which may satisfy its demand. The second supplier can supply the commodities requested by both consumers and has provided two bids accordingly.

defined exchange graph.

2.3.3.3 Arc Properties

The result of the DRE is flow determined along arcs, where arcs connect supply nodes to request nodes. A number of properties are defined on arcs, namely commodities, constraint coefficients, and preferences.

Commodities

During the information gathering step in section 2.3.2, consumers and suppliers are queried based on *commodities*. A consumer is allowed to request multiple commodities, and a supplier is allowed to supply multiple commodities. However, each possible resource transfer, i.e., each arc, is based on a single commodity. Accordingly, it is possible to color each arc, given a commodity-to-color mapping.

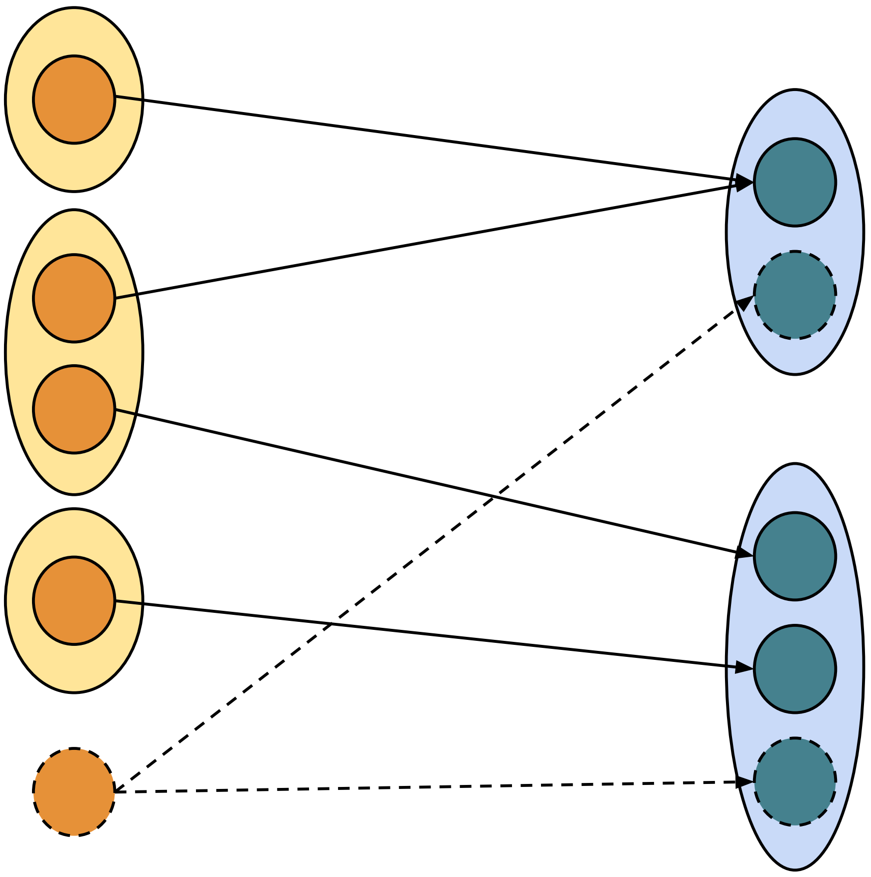

Figure 2.11: The same exchange shown in Figure 2.10 with the inclusion of false arcs. The false supplier and consumer nodes are shown with a dashed outline. Similarly, false arcs are dashed. Note that the false nodes have no associated portfolio structure – there are no constraints associated with false nodes and arcs. The inclusion of a false supplier and consumer guarantees a feasible solution.

For example, consider an exchange similar to that shown in Figure 2.10 with two fuel commodities (A, B), two requesters (R_1, R_2), and two suppliers (S_1, S_2, S_3) in the configuration described by Tables 2.4 and 2.5.

Supplier	Commodities
S_1	A
S_2	A, B
S_3	B

Table 2.4: A mapping from suppliers to commodities supplied.

Given the color map A: green, B: brown, the resulting exchange graph can be colored as shown in

Consumer	Commodities
R_1	A
R_2	B

Table 2.5: A mapping from requesters to commodities requested.

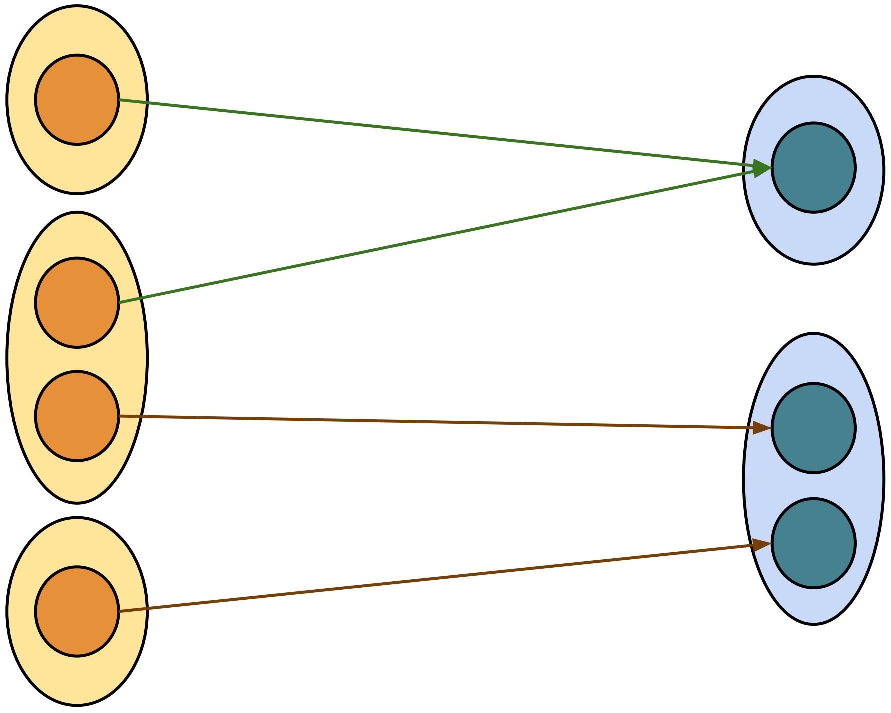

Figure 2.12: The same exchange shown in Figure 2.10 arcs colored by commodity based on Tables 2.4 and 2.5. A green arc corresponds to commodity A; a brown arc corresponds to commodity B.

Figure 2.12.

The notion of commodities is critical during the information gathering step as it is the basic classification used in communicating supply and demand. It is also useful when an exchange graph is formed, because the graph may be able to be partitioned by collections of commodities. However, once minimally connected exchange graphs are established, solution mechanisms do not employ the notion of commodities. Rather, quantities, constraints, and preferences are used.

Constraint Coefficients

Constraint coefficients are determined for an arc based on the proposed resource to be transferred along that arc, the requester's constraint conversion functions, and the suppliers constraint conversion function.

An example of supply-based constraints is provided to help clarify its purpose.

Consider a supplier enrichment facility, s, which produces the commodity enriched uranium (EU). This facility has two constraints on its operation for any given time period: the amount of Separative Work Units (SWU) that it can process, b_s^{SWU}, and the total natural uranium (NU) feed it has on hand., b_s^{NU}. The constraint set for s is then

$$K_s = \{\text{SWU}, \text{NU}\}. \tag{2.9}$$

Note that neither of these capacities are measure directly in the units of the commodity it produces, i.e., kilograms of EU.

Consider a set of requests for enriched uranium that this facility can possibly meet. Such requests have, in general, two parameters: P_j, the total product quantity (in kilograms), and ε_j, the product enrichment (in w/o ^{235}U).[1] For the purposes of this constraint set, the quality of material in question is its enrichment, i.e.,

$$q_j \equiv \varepsilon_j. \tag{2.10}$$

These values are set during a prior phase of the overall matching algorithm, and can therefore be considered constant. Further, note that, in general, an enrichment facility's operation, or rather its capacity, is governed by two parameters: ε_f, the fraction of ^{235}U in its feed material, and ε_t, the fraction of ^{235}U in its tails material. These parameters determine the amount of SWU required to produce some amount of enriched uranium, shown in Equation 2.11 as well as the amount of natural uranium, or feed, required, as shown in Equation 2.12.

$$\begin{aligned} SWU = \; & P(V(\varepsilon_j) + \frac{\varepsilon_j - \varepsilon_f}{\varepsilon_f - \varepsilon_t} V(\varepsilon_t) \\ & - \frac{\varepsilon_j - \varepsilon_t}{\varepsilon_f - \varepsilon_t} V(\varepsilon_f)) \end{aligned} \tag{2.11}$$

$$F = P \frac{\varepsilon_j - \varepsilon_t}{\varepsilon_f - \varepsilon_t} \tag{2.12}$$

[1]The notation for enrichment, ε_j, is chosen over its normal form, x_p, to limit confusion with the notation of material flow, $x_{i,j}^h$.

P in Equations 2.11 and 2.12 is the amount of produced enriched uranium, F is the amount of feed, or natural uranium, and $V(x)$ is the value function,

$$V(x) = (1 - 2x)\ln\left(\frac{1-x}{x}\right) \tag{2.13}$$

Utilizing the above equations, one can denote the functional forms of the arguments of this facility's two capacity constraints.

$$\beta_s^{NU}(\varepsilon_j) = \frac{\varepsilon_j - \varepsilon_t}{\varepsilon_f - \varepsilon_t} \tag{2.14}$$

$$\begin{aligned}\beta_s^{SWU}(\varepsilon_j) = \; & V(\varepsilon_j) \\ & + \frac{\varepsilon_j - \varepsilon_f}{\varepsilon_f - \varepsilon_t} V(\varepsilon_t) \\ & - \frac{\varepsilon_j - \varepsilon_t}{\varepsilon_f - \varepsilon_t} V(\varepsilon_f)\end{aligned} \tag{2.15}$$

These constraints correspond to the per-unit requirements for enriched uranium of natural uranium feed and SWU. Finally, we can form the set of constraint equations for the enrichment facility by combining Equations 2.10, 2.14, and 2.15.

$$\sum_{j \in J} \beta_s^{NU}(\varepsilon_j)\, x_{s,j} \le b_s^{NU} \tag{2.16}$$

$$\sum_{j \in J} \beta_s^{SWU}(\varepsilon_j)\, x_{s,j} \le b_s^{SWU} \tag{2.17}$$

Preferences & Costs

In any network flow problem, of which transportation problems are a subset, the objective coefficients associated with transporting commodities is what drives the solution. Given the nature of supply and demand constraints, the transportation problem naturally lends itself to a minimum cost formulation. A preference-based formulation has been presented thus far due to the difficulties of employing reasonable cost coefficients, as was discussed in section 1.3.2. While directly using costs should be available to users, in practice using a more abstract notion of preferences is simpler.

Formally, a preference function, $p_{i,j}(h)$, is defined which is a cardinal preference ordering over a consumer's satisfying commodity set.

$$p_{i,j}(h) \ \forall i \in I \ \forall h \in H_r \tag{2.18}$$

A preference is assigned to each arc in the NFCTP, and are a function both of the consumer, j, and producer, i, and the proposed resource transfer from consumer to producer. The dependence on producer encapsulates the relationship effects due to managerial preferences. The preference set used in the NFCTP formulation follows directly from the Preference Adjustment phase described in section 2.3.2.

The notion of a preference is a positive one, that is, an optimal solution maximizes the product of preference and flow in the system. However, the transportation problem requires a cost-based objective function. Because preferences are a proxy for cost and there is a desire to support cost-based DREs in the future, a preference-to-cost translation function is utilized. A cost translation function, f, is defined that operates on the commodity preference function to produce an appropriate cost for the NFCTP.

$$f : p_{i,j}(h) \to c_{i,j} \tag{2.19}$$

For the purposes of this work, any operator that preserves the preference monotonicity and cardinal ordering is suitable. The inversion operator has been chosen because it preserves required features and also allows for easy translation from preference to cost as well as translation from cost to preference.

$$f(x) = \frac{1}{x} \tag{2.20}$$

If cost data and a valid cost assignment methodology is developed in the future, costs may be used directly, and the preference-to-cost translation may be ignored.

2.3.3.4 Linear Programming Formulation

Combining the previous discussions, the LP Formulation of the NFCTP, denoted the NFCTP-LP, can be constructed. In general, the NFCTP is a minimum cost transportation problem that includes custom constraints as described in previous sections. Including all of the discussion in the previous sections, the formulation is straightforward and shown in Equation 2.21.

$$\min_{x} \; z = \sum_{i \in I} \sum_{j \in J} c_{i,j} x_{i,j} \tag{2.21a}$$

$$\text{s.t.} \; \sum_{i \in I_s} \sum_{j \in J} a^k_{i,j} x_{i,j} \leq b^k_s \qquad \forall\, k \in K_s, \forall\, s \in S \tag{2.21b}$$

$$\sum_{j \in J_r} \sum_{i \in I} a^k_{i,j} x_{i,j} \geq b^k_r \qquad \forall\, k \in K_r, \forall\, r \in R \tag{2.21c}$$

$$x_{i,j} \in [0, \tilde{x_j}] \qquad \forall\, i \in I, \forall\, j \in J \tag{2.21d}$$

The variables and sets used to define Equation 2.21 have been described in detail in previous sections. A short synopsis of the sets used is provided in Table 2.6, and a corresponding synopsis of the variables used is provided in Table 2.7.

Set	Description
S	suppliers
R	requesters
I	all supply nodes
I_s	nodes for a supplier s
J	all request nodes
J_r	nodes for a requester r
K_s	constraints for a supplier s
K_r	constraints for a requester r
X	the feasible set of flows between producers and consumers

Table 2.6: Sets Appearing in the NFCTP-LP Formulation

Variable	Description
$c_{i,j}$	the unit cost of flow from producer node i to consumer node j
$x_{i,j}$	a decision variable, the flow from producer node i to consumer node j
$a^k_{i,j}$	the constraint coefficient for constraint k on flow between nodes i and j
b^k_s	the constraining value for constraint k of supplier s
b^k_r	the constraining value for constraint k of requester r
$\tilde{x_j}$	the requested quantity associated with request node j

Table 2.7: Variables Appearing in the NFCTP-LP Formulation

Notably, a feasible solution to the formulation provided in Equation 2.21 is guaranteed due to the presence of false arcs. Accordingly, the DRE using this formulation will never fail within a simulation.

2.3.3.5 Mixed Integer Linear Programming Formulation

The previous linear program (LP) formulation of the Generic Fuel Cycle Transportation Problem fully describes many of the types of transactions that arise at any given time step. However, it does not allow the critical case of reactor fuel orders, which comprise a large amount of material orders within the simulation context. Specifically, it allows reactor fuel orders to be met by more than one supplier with an arbitrary amount of the order met by each supplier. Put another way, the LP formulation does not contain the discrete material information required to model the transaction of fuel assemblies.

In order to provide this capability of quantizing orders, binary decision variables must be introduced. The addition of integer variables changes both the complexity of the formulation and the complexity of the solution technique. Such a change requires a Mixed Integer-Linear Program (MILP) formulation and solution via the branch-and-bound method which solves NP-Hard combinatorial optimization problems.

Binary Variables

The primary difference between the LP and MILP formulations is the inclusion binary decision variables $y_{i,j}$. A variable $y_{i,j}$ has a value of 1 if flow occurs between producer node i and consumer node j. If flow occurs, its quantity will be equal to the equivalent flow upper bound along that arc, $\tilde{x}_j$, which denote the quantity of a quantized order.

Binary variables, representing quantized flow, are directly related to the notion of *exclusive* bids and requests discussed in section 2.3.2. In the MILP formulation, an arc (i, j) is considered exclusive if either node i or node j was defined as exclusive in the information gathering phase of the DRE. Accordingly, it is useful to partition all arcs based on this characteristic. Given the set of arcs A, a partition exists such that A can be separated into exclusive arcs, A_e, and non-exclusive arcs, or arcs that allow partial flow, A_p.

$$A = A_p \cup A_e \tag{2.22}$$

Similarly, each partition can be further subdivided into partitions based on supplier and requester.

$$A = \bigcup_{r \in R} A_{p_r} \cup A_{e_r} \tag{2.23}$$

$$A = \bigcup_{s \in S} A_{p_s} \cup A_{e_s} \tag{2.24}$$

Mutually Exclusive Constraints

Mutual requests and responses were described in section 2.3.2. These are defined as a set of requests or responses, of which only one may be satisfied. This is represented in the formulation as a constraint on the associated variables. Again, if a variable $y_{i,j}$ is set to 1, flow is sent along arc (i, j). If it is 0, no flow occurs. A *mutually exclusive* constraint simply says that only one arc in a mutual set may have a value of 1.

The set of mutually satisfying arcs is denoted M_s and M_r for suppliers and requesters, respectively. The associated constraints are then defined by Equations 2.25 and 2.26.

$$\sum_{(i,j) \in M_s} y_{i,j} \leq 1 \; \forall \; s \in S \tag{2.25}$$

$$\sum_{(i,j) \in M_r} y_{i,j} \leq 1 \; \forall \; r \in R \tag{2.26}$$

Formulation

Using the above arc partition notation allows for a much simpler written formulation of the MILP that looks quite close to the related LP formulation shown in Equation 2.21. The full formulation of the NFCTP is shown in Equation 2.27. The sets and variables involved in Equation 2.27 are described in Tables 2.8 and 2.9.

$$\min_{x,y} \; z = \sum_{(i,j)\in A_p} c_{i,j} x_{i,j} + \sum_{(i,j)\in A_e} c_{i,j} \tilde{x_j} y_{i,j} \tag{2.27a}$$

$$\text{s.t.} \sum_{(i,j)\in A_{p_s}} a^k_{i,j} x_{i,j} + \sum_{(i,j)\in A_{e_s}} a^k_{i,j} \tilde{x_j} y_{i,j} \le b^k_s \qquad \forall\, k \in K_s, \forall\, s \in S \tag{2.27b}$$

$$\sum_{(i,j)\in M_s} y_{i,j} \le 1 \qquad \forall\, s \in S \tag{2.27c}$$

$$\sum_{(i,j)\in A_{p_r}} a^k_{i,j} x_{i,j} + \sum_{(i,j)\in A_{e_r}} a^k_{i,j} \tilde{x_j} y_{i,j} \ge b^k_r \qquad \forall\, k \in K_r, \forall\, r \in R \tag{2.27d}$$

$$\sum_{(i,j)\in M_r} y_{i,j} \le 1 \qquad \forall\, r \in R \tag{2.27e}$$

$$x_{i,j} \in [0, \tilde{x_j}] \qquad \forall\, (i,j) \in A_p \tag{2.27f}$$

$$y_{i,j} \in \{0,1\} \qquad \forall\, (i,j) \in A_e \tag{2.27g}$$

Set	Description
S	suppliers
R	requesters
A_p	arcs that allow *partial* flows
A_e	*exclusive* flow arcs
A_{p_s}	arcs that allow *partial* flows for supplier s
A_{e_s}	*exclusive* flow arcs for supplier s
A_{p_p}	arcs that allow *partial* flows for requester r
A_{e_p}	*exclusive* flow arcs for requester r
M_s	arcs (i,j) associated with *mutually exclusive* supply for supplier s
M_r	arcs (i,j) associated with *mutually exclusive* requests for requester r
X	the feasible set of flows between producers and consumers
Y	the binary variable set of flows between producers and consumers

Table 2.8: Sets Appearing in the NFCTP Formulation

Variable	Description
$c_{i,j}$	the unit cost of flow from producer node i to consumer node j
$x_{i,j}$	a decision variable, the flow from producer node i to consumer node j
$y_{i,j}$	a decision variable, whether flow exists from producer node i to consumer node j
$a^k_{i,j}$	the constraint coefficient for constraint k on flow between nodes i and j
b^k_s	the constraining value for constraint k of supplier s
b^k_r	the constraining value for constraint k of requester r
$\tilde{x_j}$	the requested quantity associated with request node j

Table 2.9: Variables Appearing in the NFCTP Formulation

The examples of the various constraints from the previous section also apply here. The only difference is the notion of the binary variables, $y_{i,j}$, which act as on/off switch as to whether a consumer's entire requested amount of a resource is met by a supplier or not.

Using this advanced formulation adds significant complexity to the resolution method at every time step. However, should a user wish to find a feasible solution in a shorter amount of time, simple heuristics exist. Such a heuristic used in Cyclus is provided in section 2.3.3.6, and further heuristic development is a fruitful area of future work.

Note that each constraint coefficient for binary variables can be rewritten as Equation 2.28 and each objective coefficient can be rewritten as Equation 2.29.

$$a_{i,j}^{k\prime} = a_{i,j}^{k} \tilde{x_j} \tag{2.28}$$

$$c_{i,j}^{\prime} = c_{i,j} \tilde{x_j} \tag{2.29}$$

Using both updated definitions, a simpler formulation can be written and is shown in Equation 2.30.

$$\min_{x,y} \ z = \sum_{(i,j)\in A_p} c_{i,j} x_{i,j} + \sum_{(i,j)\in A_e} c_{i,j}^{\prime} y_{i,j} \tag{2.30a}$$

$$\text{s.t.} \sum_{(i,j)\in A_{p_s}} a_{i,j}^{k} x_{i,j} + \sum_{(i,j)\in A_{e_s}} a_{i,j}^{k\prime} y_{i,j} \leq b_s^k \qquad \forall\, k \in K_s, \forall\, s \in S \tag{2.30b}$$

$$\sum_{(i,j)\in M_s} y_{i,j} \leq 1 \qquad \forall\, s \in S \tag{2.30c}$$

$$\sum_{(i,j)\in A_{p_r}} a_{i,j}^{k} x_{i,j} + \sum_{(i,j)\in A_{e_r}} a_{i,j}^{k\prime} y_{i,j} \geq b_r^k \qquad \forall\, k \in K_r, \forall\, r \in R \tag{2.30d}$$

$$\sum_{(i,j)\in M_r} y_{i,j} \leq 1 \qquad \forall\, r \in R \tag{2.30e}$$

$$x_{i,j} \in [0, \tilde{x_j}] \qquad \forall\, (i,j) \in A_p \tag{2.30f}$$

$$y_{i,j} \in \{0, 1\} \qquad \forall\, (i,j) \in A_e \tag{2.30g}$$

2.3.3.6 A Heuristic Solution

With full simulation domain knowledge of supply and demand, including false arcs, a feasible solution can be found. By definition a feasible solution is a *solution* to the possible flow of resources, but not necessarily an *optimal* solution. Many heuristics may be applied to bipartite graphs with constrained flows. A simple *greedy* heuristic is presented here and implemented.

The maximum flow along an arc, x_{max}, depends on the constraints associated with each node on the arc. For nodes i and j belonging to portfolios s and r, respectively, the maximum allowable flow is defined as

$$x_{max} = \min\{\min\{\frac{b_s^k}{a_{i,j}^k}\ \forall k \in K_s\},\ \min\{\frac{b_r^k}{a_{i,j}^k}\ \forall k \in K_r\}\}. \tag{2.31}$$

The Greedy Exchange Heuristic matches maximum flow along arcs, up to the requested amount defined by each request portfolio, q_r, after having sorted all arcs. The constraining values of each arc, b_k, are updated upon declaration of a match (via an `AddMatch` function) in Algorithm 1.

```
Data: A resource exchange graph with constraints and preferences.
Result: A valid set of resource flows.
sort request partitions by average preference;
forall the $r \in R$ do
    sort requests by average preference;
    matched $\leftarrow$ 0;
    while matched $\leq q_r$ and $\exists$ a request do
        get next request;
        sort incoming arcs by preference;
        while matched $\leq q_r$ and $\exists$ an arc do
            get next arc;
            remaining $\leftarrow q_r$ - matched;
            to_match $\leftarrow \min\{$remaining, $x_{max}\}$;
            AddMatch(arc, to_match);
            matched $\leftarrow$ matched + to_match;
        end
    end
end
```

Algorithm 1: Greedy Exchange Heuristic

2.3.3.7 Departure from the MCTP

The classic MCTP includes the coloring of flows based on commodity type. For example, for a commodity, h, the unit cost of flow would be $c^h_{i,j}$ rather than $c_{i,j}$. This is included because multiple commodities can flow along the same arc in the MCTP. In other words, the node-arc incidence matrix includes an extra commodity dimension.

The multicommodity nature of the NFCTP is included in constraints, rather than arcs. Because each node pairing, (i, j), corresponds to a specific, proposed resource transfer, it can only have one commodity associated with it. Instead, the constraint set, K, is applied over multiple arcs, where each arc is assigned its own commodity.

Take the enrichment facility example, expanding on the previous discussion. Note that an enrichment facility takes feed uranium and then enriches its ^{235}U content. This feed uranium can come from different sources which have different feed enrichments. In practice, the most likely sources of feed uranium are natural uranium (NU) or recycled uranium (RU), a product of reprocessing light water reactor fuel. Recycled uranium may be advantageous to use if it has a higher weight percent of ^{235}U than does natural uranium. We can now state the set the values for H_r for this facility:

$$H_r = \{\text{NU}, \text{RU}\} \tag{2.32}$$

One or more constraints would then accompany any requests. For example, one could constraint total ^{235}U content needed, which would include both NU and RU flows.

2.3.4 Implementation

The DRE and its solution framework are implemented in three layers. The first layer includes information for specific `Resource` types. For example, a `Material`-based exchange is used for agents to communicate supply and demand information regarding `Material` objects. The *resource layer* is the point of entry and exit of the DRE framework. It is the agent-facing interface of the DRE: supply and demand is provided to the DRE as input during the information gathering step, and trades to be executed are provided to agents as output.

The second layer, called the *exchange layer*, is a `Resource`-agnostic implementation of a specialized bipartite graph. Supply/demand constructs in the first layer are translated into stateful objects repre-

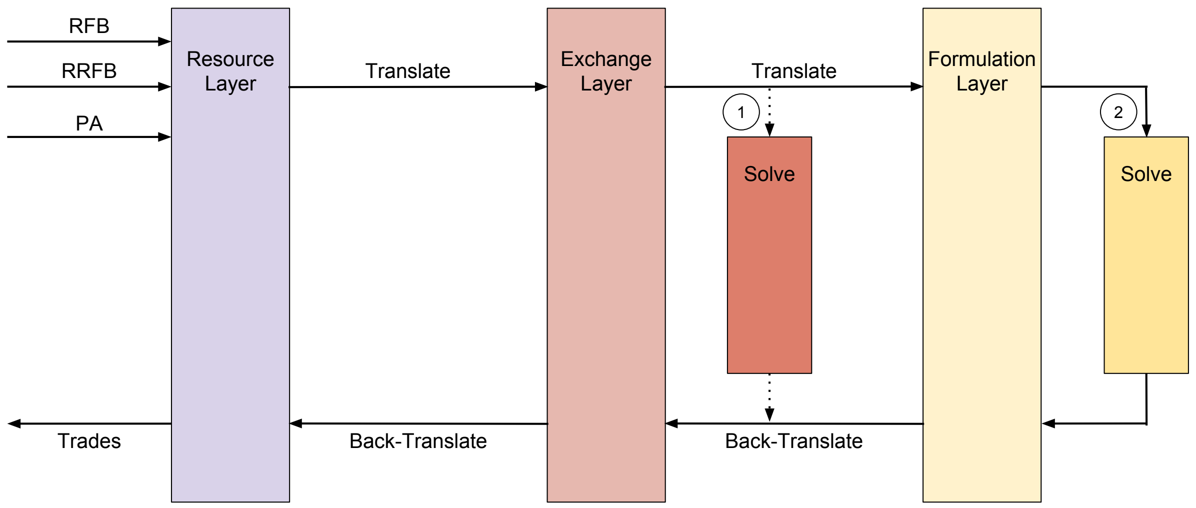

Figure 2.13: The full DRE workflow is shown. The information gathering phase results in the resource layer. The resource layer is translated to the exchange layer; a decision is made whether to continue translation or to directly solve, marked by the number 1. If the exchange is not solved, it is translated into an instance of the NFCTP resulting in the formulation layer. A choice of solver is made, marked by the number 2, and the instance is solved. The solution is back-translated through the exchange and resource layers. The result is a series of resource trades to be executed in the simulation.

senting nodes, arcs, constructs that carry constraint information, *et cetera*. The collection of objects and structures combine to create an `ExchangeGraph`. Any custom, Cyclus-aware solver can be applied to an `ExchangeGraph` to determine a feasible solution to the DRE.

In order to use sophisticated, 3rd party LP and MILP solving libraries, the `ExchangeGraph` must be translated into an appropriate data structure representing an instance of the NFCTP, resulting in the *formulation layer*. The Open Solver Interface (OSI) [24] is used to create the necessary formulation structures, including a constraint matrix and objective coefficient vector. The NFCTP instance is then solved.

After a feasible, perhaps optimal, solution to the NFCTP is found, whether in the exchange or formulation layer, the solution is back-translated to the resource layer. The agents associated with successful supply-demand connections are informed, and trades of resources between agents are executed. A graphic of the entire workflow is shown in Figure 2.13.

2.3.4.1 Resource Layer

The resource layer utilizes *templated* classes in order to reduce the amount of code required for implementation. Each object is templated on the concrete `Resource` type, e.g., the `Material` and `Product` classes. The fundamental data structures in the resource layer reflect the constructs of the information gathering procedure described in section 2.3.2.

In the RFB phase of the DRE, agents populate `RequestPortfolios` with `Requests` and `CapacityConstraints`. A `Request` defines a desired `Resource`, communicating quantity, quality, and preference. Any number of `CapacityConstraints` may be added to a `RequestPortfolio`. A `CapacityConstraint` defines a capacitating value and a conversion function that takes as an argument a `Resource` and returns a value in units of the conversion function. For `RequestPortfolios`, constraints are assumed to be demand constraints, i.e., take the form of a greater-than constraint. In the RRFB phase of the DRE, agents populate `BidPortfolios` with `Bids` and `CapacityConstraints`. Agents can inspect the population of `Requests` and associated `Resources`. A `Bid` targets a specific `Request`, responding with a proposed `Resource` to transfer to the requester. `CapacityConstraints` are applied to all `Bids` in a portfolio. For bidders, constraints are assumed to be less-than constraints. Before continuing, requesting agents and their managers are allowed to alter the preference associated with each `Request-Bid` pair in the PA phase of the DRE. When a solution to the DRE is found, bidders associated with successful `Request-Bid` pairs are informed, and a trade of the bidder's `Resource` is initiated.

Future work can be focused on providing more features to the DRE implementation. A natural extension of the present work is to support both kinds of constraints, greater and less-than, in `Portfolio` data structures. Additionally, the PA procedure could use a negotiation model that involves both suppliers and requesters in order to define a final preference for an arc. Such an extension would allow for more seamless and natural usage of arc costs in addition to preferences.

2.3.4.2 Exchange Layer

The exchange layer is constructed by an `ExchangeTranslator` object that translates the resource layer objects into an instance of an `ExchangeGraph`. Request and bid objects are translated to `ExchangeNodes`, and portfolio objects are translated to `ExchangeGroups`. Constraint coefficient and preference information is recorded on `ExchangeArcs`, which store a reference to a supply `ExchangeNode` and a demand

ExchangeNode. Finally, constraint values are stored on the appropriate ExchangeGroup object.

An ExchangeContext object is tasked with storing a mapping from Request and Bid objects to their associated ExchangeNode. Importantly, the exchange layer does *not* depend on resource type, i.e., the resource type is abstracted away during translation. Finally, a general solver can be implemented that operates on the ExchangeGraph. A solution to the ExchangeGraph instance is a mapping from ExchangeArcs to flow quantities that does not violate the provided constraints. After a solution is found, it is back-translated to the resource layer.

2.3.4.3 Formulation Layer

While a solver may operate on the exchange layer, an instance of an ExchangeGraph can be translated fully into the NFCTP. Once in an LP or MILP form, the DRE instance can be solved by sophisticated 3rd party libraries. In order to interface with a large number of the possible solvers, including COIN-OR and CPLEX, the COIN-OR OSI API [24] is utilized.

The translation from the exchange layer to formulation layer is managed by the ProgTranslator class. A variable in the NFCTP is associated with each ExchangeArc, with variable bounds set by request values on ExchangeNodes; a binary variable is used if the arc is exclusive, otherwise a linear variable is used. Capacity coefficients and preference values defined for ExchangeArcs are translated into an objective coefficient vector and constraint matrix. The right-hand-side b constraint vector is determined by ExchangeGroup constraining values.

A solution to the NFCTP instance is determined by the identified solver, assigning values to linear and integer variables. Linear variable values map directly to assigned resource flow quantity. If a binary variable is set to unity in a solution, the maximum possible flow value is assigned, analogous to $\tilde{x_j}$ in the NFCTP formulation. The variable-flow value assignment is then back-translated into an equivalent ExchangeArc-flow value assignment by the ProgTranslator.

2.3.5 Proof of Principle

In order to demonstrate the correctness of the methodology and implementation, two test cases were developed and analyzed. These test cases are entire Cyclus simulations in which the full DRE procedure is executed at each time step. Both scenarios validate the ability of the DRE to model preferences, preference adjustment, and unresolved markets. The first scenario is a simulation including quantity-based constraints

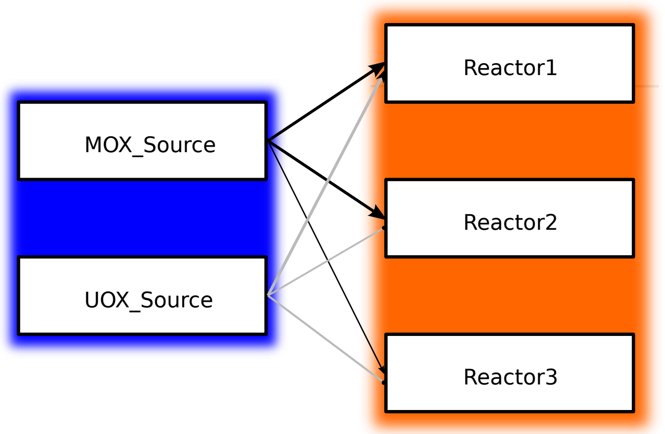

Figure 2.14: Schematic illustrating the first fuel cycle scenario. The thickness of the arrows represents the preference value and the grey color indicates that a material transfer is possible.

and dynamic commodity-based preferences. The second scenario illustrates a simulation that involves quality constraints and dynamic quality-based requests. In each scenario, fuel quantities are treated using arbitrary units without loss of generality. For the purposes of the enrichment example, a unit of fuel is equivalent to a kilogram.

Each scenario is comprised of archetypes defined in Cycamore [61]. The minimal Institution and Region archetypes are used because no complicated facility deployment logic is needed. The facility archetypes used include the *SourceFacility*, *BatchReactor*, and *EnrichmentFacility*.

Finally, each instance of the DRE is solved using the greedy heuristic described in section 2.3.3.6. In each case, requests and supplies are *not* exclusive, and thus multiple sources of supply may be matched to a request. In general, these exchanges are very small and have a unique objective solution which corresponds to the solution determined by the greedy heuristic.

2.3.5.1 Test Cases

2 Sources, 3 Reactors

As shown in Figure 2.14, this scenario includes three *BatchReactors* and two *SourceFacilities*. The *BatchReactors*, denoted as `Reactor1`, `Reactor2`, and `Reactor3`, each have a unique fuel preference. One *SourceFacility* supplies MOX while the other supplies UOX; these are denoted as `MOX_Source` and `UOX_Source`, respectively. Any reactor may be fueled from MOX or UOX fuel; both fuel types are *fungible* in this scenario.

In this example case Reactor1, Reactor2, and Reactor3 are deployed sequentially over 3 time steps. Each of these has a full core when built and requires 1 unit of fresh fuel at each subsequent time step. Both source facilities have a capacity of 2.5 units each time step.

The simulation begins with the following facilities: MOX_Source, UOX_Source, and Reactor1. At time step 2, Reactor2 is deployed, followed by Reactor3 at time step 3. Reactor1 and Reactor2 both are given a stronger preference for MOX requests than Reactor3. At time step 4, Reactor1, Reactor2, and Reactor3 all request to refuel with MOX. At time step 5, Reactor1 changes its preference to UOX. Table 2.10 summarizes the reactor preferences as a function of time.

Table 2.10: Time sequence of reactor preferences and the total MOX requested. The MOX capacity for each time step is 2.5 units.

Time step	Reactor1		Reactor2		Reactor3		Total MOX Requested [units]
	Fuel	Preference	Fuel	Preference	Fuel	Preference	
1	MOX	1.0	none		none		0.0
2	MOX	1.0	MOX	1.0	none		1.0
3	MOX	1.0	MOX	1.0	MOX	0.5	2.0
4	MOX	1.0	MOX	1.0	MOX	0.5	3.0
5	UOX	2.0	MOX	1.0	MOX	0.5	2.0

Enrichment, 2 Reactors

As pictured in Figure 2.15, this scenario includes one *EnrichmentFacility* and two *BatchReactors*. The *EnrichmentFacility*, denoted as `Enrichment`, has a designated capacity at each time step. The two *BatchReactors*, denoted as `Reactor1` and `Reactor2`, request a given amount and quality of enriched uranium upon refueling.

The `Enrichment` facility is constrained by a constant capacity of 10 SWU per time step. For this entire simulation, trade between `Enrichment` and `Reactor1` is preferred over trade between `Enrichment` and `Reactor2` with preference values of 1.0 and 0.5, respectively. Each reactor requests 1 unit of enriched uranium at each time step.

Initially, both reactors are present in the simulation and have a full core of 3% enriched uranium. On time step 1, `Reactor1` requests uranium enriched to 5% U-235 while `Reactor2` requests uranium at a 3% enrichment level. At time step 2, `Reactor1` reduces its enrichment request to 3%. Table 2.11 summarizes the reactor requests as a function of time.

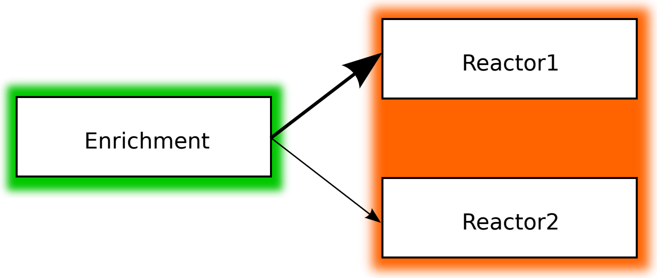

Figure 2.15: Schematic illustrating the second fuel cycle scenario. The thickness of the arrows represents the preference value.

Table 2.11: Time sequence of reactor preferences and the total SWU requested. The SWU capacity for each time step is 10.

Time step	`Reactor1`		`Reactor2`		Total SWU Requested
	Recipe	Preference	Recipe	Preference	
0	3% U-235		3% U-235		0
1	5% U-235	1.0	3% U-235	0.5	10.6
2	3% U-235	1.0	3% U-235	0.5	6.8

2.3.5.2 Results

The cases outlined in section 2.3.5.1 have been designed to provide different conditions at each point in time. The results for these cases are discussed below. In each of these cases, the total Cyclus run time was ∼0.1 seconds and the output database size was ∼68 kB.

Note that the resource flows in Figures 2.16-2.22 have been generated automatically from Cyclus output using Cyan [17]. Due to this, these figures only show facility agents which participated in a resource exchange. For instance, Figure 2.16 does not show the `UOX_Source` facility, even though it is present in the simulation.

2 Sources, 3 Reactors

Initially present are the source facilities and `Reactor1`. `Reactor1` has a preference for accepting MOX fuel over UOX. The `MOX_Source` capacity of 2.5 units is more than enough to handle the 1 unit of MOX requested by `Reactor1`. This matching may be seen in Figure 2.16. The `MOX_Source` only provides the 1 unit of material requested by `Reactor1`. It, correctly, does not oversupply.

At time step 2 in this simulation, `Reactor2` is deployed and also requests fuel with the preference for MOX. Figure 2.17 displays that the `MOX_Source` indeed has the required capacity to meet the requests of both of the reactors. This may seem trivial at first glance but it is important to emphasize that the resource exchange solver was not altered in any way to handle both time steps 1 and 2. Furthermore, the solver on time step 1 had no future knowledge that `Reactor2` would be deployed on time step 2. This is significantly different than the traditional system dynamics approach.

On time step 3, `Reactor3` is deployed. This facility still prefers to accept MOX fuel over UOX fuel. At this point, 3 units of MOX are requested (1 unit from each facility) but the `MOX_Source` may only provide

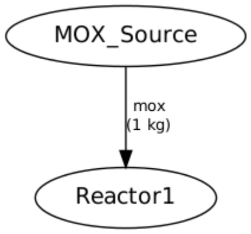

Figure 2.16: Time step 1 for the 2 Sources, 3 Reactors case.

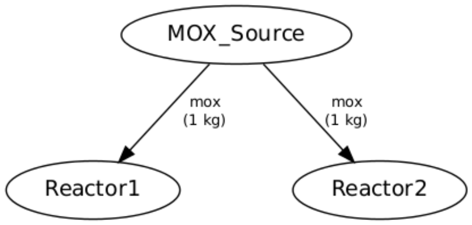

Figure 2.17: Time step 2 for the 2 Sources, 3 Reactors case.

Table 2.12: Resource exchange preferences for agents on time steps 3 and 4 for reactors in the 2 sources, 3 reactors case.

	$t = 3$		$t = 4$	
Agent	UOX	MOX	UOX	MOX
`Reactor1`	0.0	1.0	2.0	1.0
`Reactor2`	0.0	1.0	0.0	1.0
`Reactor3`	0.0	0.5	0.0	0.5

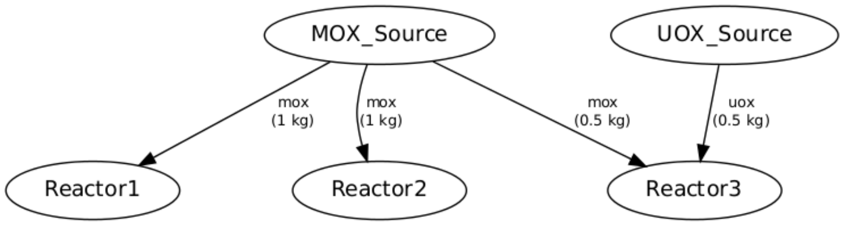

Figure 2.18: Time step 3 for the 2 Sources, 3 Reactors case.

2.5 units. Because of this, the `UOX_Source`, which has been present in the simulation since the beginning, now enters the exchange to make up for the missing 0.5 units of fuel not obtainable from the `MOX_Source`. `Reactor3` is selected to receive the UOX fuel rather than `Reactor1` and `Reactor2`. These preferences are detailed in Table 2.12. In time step, 3 because all agents tie for UOX, `Reactor1` and `Reactor2` tie for MOX, and the `Reactor3` preference for MOX is less than the others, `Reactor3`'s full request for MOX is not met and it must top-up with UOX. This situation is displayed in Figure 2.18.

Finally, on time step 4 the preference of `Reactor1` for UOX changes from 0.0 to 2.0. This alteration causes the tie previously present for UOX to be broken. Furthermore, the value of 2.0 makes this the most preferred arc in the system so it is attempted to be satisfied first. As may be seen in Figure 2.19, the `UOX_Source` capacity of 2.5 units is more than enough to satisfy the request from `Reactor1` for 1 unit of

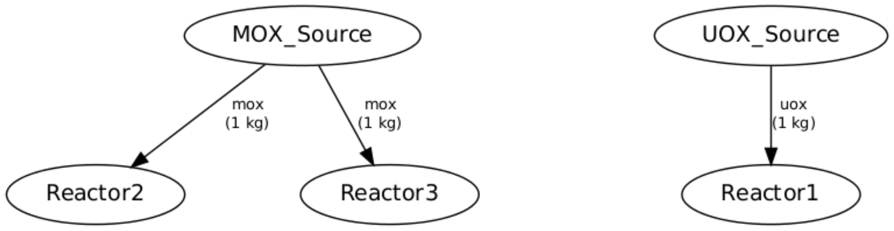

Figure 2.19: Time step 4 for the 2 Sources, 3 Reactors case.

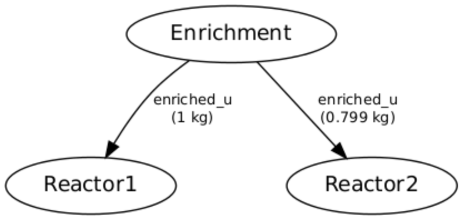

Figure 2.20: Time step 1 for the Enrichment and 2 Reactors case.

UOX. Since `Reactor1` does not diminish the capacity of the `MOX_Source` both `Reactor2` and `Reactor3` are able to obtain their first choice fuel.

This simple 2 source, 3 reactor simulation shows how the resource exchange can dynamically and correctly adapt to the both facility deployments and the preferences that these agents have for requested resources.

Enrichment and 2 Reactors

Initially, `Enrichment`, `Reactor1`, and `Reactor2` are all present. The reactors both begin with cores composed of 3% enriched uranium.

Figure 2.20 shows the result of the resource exchange for time step 1. Here the SWU capacity of `Enrichment` is not sufficient to meet the requests for 5% and 3% enriched fuel simultaneously but is enough to meet either of them individually. Therefore, the bid for at least one of the reactors must be partially unmet. Due to the preferences, the requests of `Reactor1` will be met first. Thus in Figure 2.20 `Reactor1` receives 1 unit of 5% enriched fuel. However, `Reactor2` only receives ∼80% of its request for 3% enriched uranium. This is not enough to run on and so this material is saved for the future.

On time step 2, `Reactor1` now switches from requesting 5% enriched fuel to requesting 3% enriched

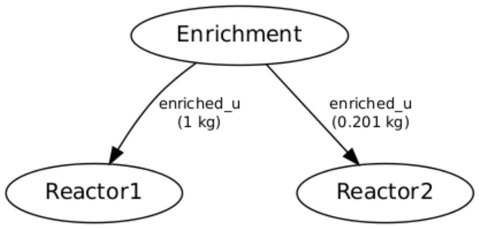

Figure 2.21: Time step 2 for the Enrichment and 2 Reactors case.

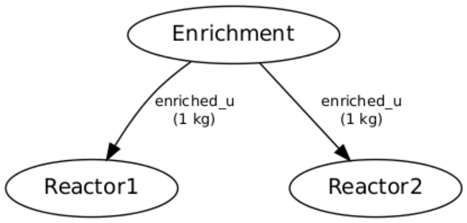

Figure 2.22: Time step 3 for the Enrichment and 2 Reactors case.

fuel. However, the reactor archetypes are implemented such that if they have stored fuel from a previous time step, they will instead only request a mass needed to create 1 unit of fuel. Therefore, `Reactor2` only requests ∼0.2 units from `Enrichment`. `Reactor1` continues to request 1 unit of material and this is met because the SWU capacity constraint is not exceeded. These resource flows may be seen in Figure 2.21.

No further adjustments of requests were made on time step 3. Thus the results displayed in Figure 2.22 represent the same dynamic resource exchange procedure in time step 2. The key differences here however are that now the system has returned to a steady state and - unlike in time step 1 - the SWU capacity of `Enrichment` is enough to meet the 2 units of 3% enriched fuel coming from both reactors. Thus `Reactor1` and `Reactor2` each receive the kilogram of fuel that they request. Without further adjustments this system will continue *ad infinitum*.

2.4 Summary

This chapter introduced a novel way to employ agent-based modeling techniques in the nuclear fuel cycle. Section 2.1 first described how a simulation is structured, focusing on where agent interactions occur in a given time step. A discussion of how the notion agency is applied to fuel cycle entities and a proof-of-principle simulation was shown in section 2.2. Finally, a detailed description of a novel supply-demand, agent-based framework, the DRE, was presented in section 2.3. The DRE is a critical advancement in the realm of nuclear fuel cycle simulation, enabling arbitrary facility-based constraints, competition for fungible resources, and the application of socio-economic models.

3 Experimentation Methodology

The NFCTP is the first attempt at solving the supply and demand of the nuclear fuel cycle in a dynamic manner within a NFC simulation. Accordingly, there is no precedent for investigating the performance and efficacy of a given approach. This chapter describes an experimental methodology to assess the NFCTP in which rules for generating instances of exchanges are defined, exchanges are generated, exchanges are executed, and results are analyzed. The goal of this work is twofold: demonstrate the generation of a large number of exchange instances under reasonable assumptions, and analyze the effects and performance of different solvers applied to those exchange instances.

The chapter begins with a discussion of the generation of exchanges, in section 3.1. Two species of exchanges are included: one in which reactors are requesting fuel, and one in which reactors are supplying used fuel. In NFC parlance, these are called the *front end* of the fuel cycle and *back end* of the fuel cycle. Notably, both of these exchanges occur in the same time step.

Generating and solving instances of exchanges at a large scale is a difficult problem. The Cyclopts (Cyclus Optimization Studies) framework was implemented for this purpose, consisting of both a Python and C++ layer. The Python layer is largely responsible for generating exchanges and interfacing with an associated persistence mechanism. The C++ layer is compiled and linked against the Cyclus kernel shared object library, `libcyclus`, and is responsible for calling directly into the kernel's resource exchange API. section 3.2 describes the implementation of Cyclopts and its varied modes of operation.

3.1 Generating Exchanges

Instances of resource exchanges are required to analyze the effects and performance of the NFCTP formulation and its solvers. In the absence of large Cyclus simulations with interesting facility and relationship models, instances must be generated given some set of rules and parameters. Two distinct *species* of exchanges are generated, those related to the *front end* of the nuclear fuel cycle and those related to the *back end* of the nuclear fuel cycle. Broadly, the front end of the fuel cycle is concerned with fueling reactors, and the back end is concerned with either recycling or disposing of used fuel exiting reactors. Common features to both types of exchange generation are described in section 3.1.1.

Previously, section 2.3, described the methodology for solving a single exchange. If a large exchange is separable, i.e., it can be completely separated into two or more smaller exchanges, sub-exchanges can be solved independently. Section 3.1.2 provides an argument for why it is valid to split exchange instances into, at minimum, the front and back ends of the NFC.

Exchange generation, absent full-scale simulation, is a naturally parameterized process. Some generation parameters are common to any NFCTP instance, and are described in section 3.1.3. Specific species may additionally define their own set of parameters. Section 3.1.4 describes the parameters generation methodology associated with front-end exchanges. Section 3.1.5 follows with a similar discussion for back-end exchanges.

3.1.1 Common Features

3.1.1.1 Fuel Cycles and Commodities

Three types of fuel cycles are generated: a once-through fuel cycle, labeled OT; a plutonium-recycle fuel cycle, labeled MOX; and a plutonium and thorium-recycle fuel cycle, labeled MOX-ThOX. As fuel cycles increase in complexity, the number of commodities that exist increases, as shown in Table 3.1. The commodities are referred to by abbreviation: Uranium Oxide (UOX), Mixed Plutonium Oxide for Thermal Reactors (TMOX), Mixed Plutonium Oxide for Fast Reactors (FMOX), Thorium Oxide for Fast Reactors (FThOX).

Table 3.1: A mapping between fuel cycles to the commodities that exist in each one.

Fuel Cycle	Commodities
OT	UOX
MOX	UOX
	TMOX
	FMOX
ThOX	UOX
	TMOX
	FMOX
	FThOX

3.1.1.2 Reactors

Reactors are modeled as either thermal or fast reactors. It is necessary to estimate the amount of fuel exchanged by reactors each time step. Accordingly, thermal reactors are simplified models of AP-1000 reactors [1], and fast reactors are simplified models of BN-600 reactors [35]. Using the dimensions in Table 3.2, one can estimate that the AP-1000 core volume is approximately 12.5 times larger than the BN-600 core.

Table 3.2: Primary Reactor Parameters

Reactor	Core Height (m)	Core Diameter (m)	Number of Assemblies
AP1000	4.27	3.04	157
BN600	0.75	2.05	369

Reactors operate in a batch mode, where each batch is approximately one quarter of the reactor core, an assumption which similar to other analyses [49]. Additionally, a single AP-1000 fuel assembly is assumed to contain 450 kg of material [39]. Therefore, a single batch of thermal reactor fuel is assumed to be

$$450\frac{kg}{assembly} * \frac{t}{1000\,kg} * 157\frac{assemblies}{core} * \frac{1}{4}core = \sim 17.6t. \tag{3.1}$$

The amount of fuel required and number of assemblies by each reactor type is shown in Table 3.3. The number of assemblies is taken as the ratio of total number of assemblies and number of batches per core rounded to the nearest integer. The batch size for the BN600 reactor is estimated by dividing the AP1000 batch size by the relative core volume.

Table 3.3: Reactor Batch Size

Reactor Type	Quantity (t)	Number of Assemblies
AP1000	17.6	39
BN600	1.41	92

The reactors that operate in a given exchange is also a function of the fuel cycle being modeled. In a OT fuel cycle, only thermal reactors exist. In the MOX case, fast reactors that prefer MOX-based fuel are added and denoted as FMOX reactors. Finally, in the MOX-ThOX case, an additional class of fast reactor is added that prefers ThOX-based fuels and is denoted as FThOX. A summary of available types of reactors as a function of the fuel cycle being modeled is shown in Table 3.4, and a summary of the reactor models used for each reactor type is shown in Table 3.5.

Table 3.4: A mapping between fuel cycles to the reactor types that exist in exchange instances.

Fuel Cycle	**Reactor Types**
OT	Thermal
MOX	Thermal FMOX
ThOX	Thermal FMOX FThOX

Table 3.5: A mapping between surrogate reactor models and reactor types.

Reactor Model	**Reactor Types**
AP1000	Thermal
BN600	FMOX FThOX

Reactors may be fueled by different fuel types, i.e., fuel commodities. The set of commodities that reactors can use is a modeling assumption and a proxy for how reactors may behave in simulations; this particular reactor-to-commodity mapping may not be true for other analysts' fuel cycle models. However, it is appropriate to make certain broad assumptions for such an exploratory study. A mapping of reactors to acceptable commodities is provided in Table 3.6. Note that there is still a preference distribution associated with each reactor-commodity pair as well as constraint coefficient effects. Accordingly, each reactor-commodity pair provides a unique effect on an exchange instance.

Table 3.6: A mapping between reactor types and the commodities allowed to fuel each reactor type.

Reactor Types	**Fuel Commodities**
Thermal	UOX TMOX FMOX
FMOX	UOX TMOX TMOX FThOX
FThOX	UOX TMOX TMOX FThOX

Finally, each reactors in the system is representing an individual agent in a given simulation. Agents are assumed to have some differentiating behavior. In order to model this effect, each reactor is assigned a random variable, $x \in [0, 1)$, that represents a unique measure of quality of fuel requested by reactors.

This stochastic property is expounded upon further in section 3.1.4 and 3.1.5.

3.1.1.3 Support Facilities

In a front-end exchange, fuel suppliers exchange material with reactors. In a back-end exchange, reprocessing and storage facilities exchange material with reactors. In either case, facilities that are not reactors are referred to as *support* facilities, as they support the reactors which generate power. Support facilities for front-end exchanges are described in section 3.1.4.1, and support facilities for back-end exchanges are described in section 3.1.5.1.

3.1.1.4 Preferences

Preferences for all transactions have a default value, $p_c(i, j)$, based on the proposed commodity to be transferred between a supplier, i, and consumer, j. However, a large exchange with a small preference distribution results in problem degeneracy. Further, a primary application for Cyclus is the modeling of regional and location effects on fuel cycles. Accordingly, a location proxy is provided for preferences, as shown in Equation 3.2, in order to simulate both location-based preferences and non-degenerate exchange instances.

Preferences can also be a function of facility location. Each facility is assigned a location value, $\text{loc}_i \in [0, 1)$. The domain is then divided evenly into ten regions, where the first region comprises all location values in $[0, 0.1)$, *et cetera*. For example, a facility at location of 4.6 is in the fifth region. δ_{reg} and δ_{loc} are binary variables which are activated based on the parameters described in section 3.1.3. If δ_{reg} is zero, no location-based preferences are used. If δ_{loc} is zero, only coarse, region-based preferences are used. In both cases, preferences are a function of the Euclidean distance between regional and location values. The inverse exponential functional form was chosen in order to model a preference gradient that decays as distance increases.

$$p_l(i, j) = \delta_{\text{reg}} \frac{\exp(-|\text{reg}_i - \text{reg}_j|) + \delta_{\text{loc}} \exp(-|\text{loc}_i - \text{loc}_j|)}{1 + \delta_{\text{loc}}} \tag{3.2}$$

The preference for a given arc is then a weighted, linear combination of location and commodity preferences as shown in Equation 3.3. The weighting factor, $r_{l,c}$, is a parameter of exchange generation and described further in section 3.1.3.

$$p(i,j) = p_c(i,j) + r_{l,c}p_l(i,j) \tag{3.3}$$

3.1.2 Splitting Exchanges

A well known simplification of the Multicommodity Transportation Problem occurs when supply and demand is separate for separate commodities. The large multicommodity problem can then be decomposed into n single commodity subproblems, where n is the number of commodities. Each subproblem can be solved separately from the others.

An analog exists in the NFCTP when the Exchange Graph is *separable*. A bipartite graph with directed arcs, A, consisting of sending nodes, U, and receiving nodes, V, is separable if there a partition

$$A = A_1 \cup A_2 \tag{3.4}$$

$$U = U_1 \cup U_2 \tag{3.5}$$

$$V = V_1 \cup V_2 \tag{3.6}$$

such that no node in U_1 is connected to a node in V_2 and no node in U_2 is connected to a node in V_1. The graph shown in Figure 3.1 is an example of a separable bipartite graph.

The Exchange Graph of the NFCTP, however, has additional structure in the form of portfolios and thus has a stricter notion of separability. Specifically, the partition must also separate the set of supplier portfolios, S, and requester portfolios, R, as in Equations 3.7 and 3.8, respectively.

$$S = S_1 \cup S_2 \tag{3.7}$$

$$R = R_1 \cup R_2 \tag{3.8}$$

Figure 3.2 depicts a separable Exchange Graph, for example, while Figure 3.3 shows an Exchange Graph where the underlying bipartite graph is separable, but full separability is broken by the overlaid

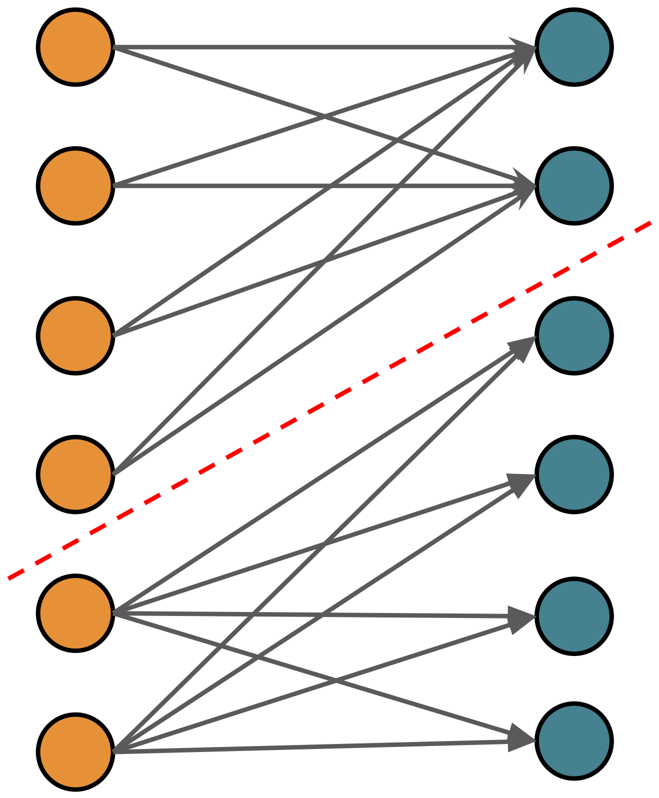

Figure 3.1: A separable bipartite graph with the partition shown as a red dashed line.

portfolio structure.

The Exchange Graph resulting from the information gathering phase of the DRE will be minimally separable into front-end and back-end exchanges if two conditions are true:

1. Reactor output commodities can *not* be sent to both other reactors and supporting facilities.
2. Supporting facility output commodities can *not* be sent to both other supporting facilities and reactors.

In the first case, separability is broken by a supplier providing bids across a separating partition. A minimal example is shown in Figure 3.4. This case can arise if reactors can somehow directly refuel other reactors. In the NFC domain, such an arrangement only occurs in an abstraction of a self-recycling system in which there is a dedicated recycling complex associated with a fast reactor. It is reasonable for a

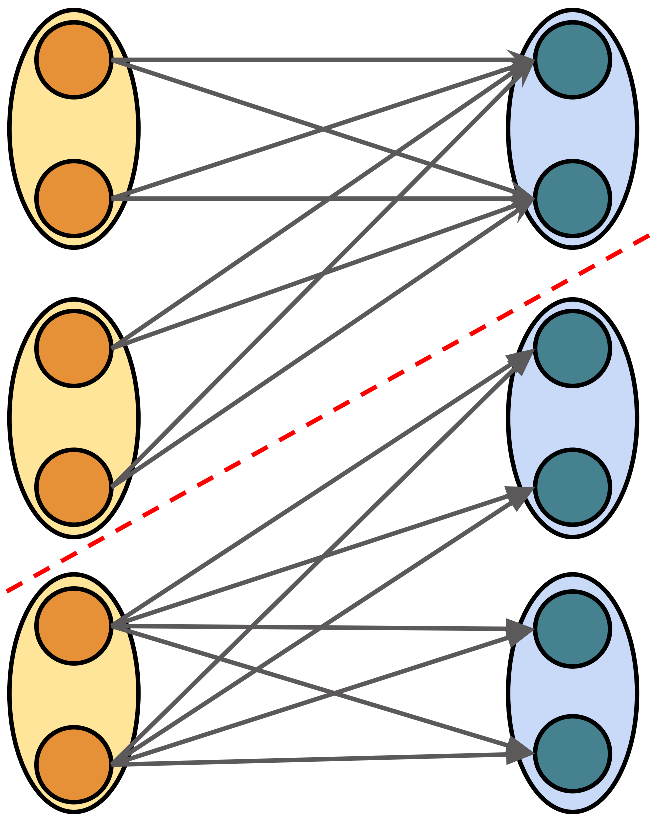

Figure 3.2: A separable Exchange Graph with nodes grouped by portfolio and the separating partition shown as a red dashed line.

self-recycling reactor system to be implemented in such a way that it does not participate in the DRE for self-refueling purposes. Accordingly, this condition is expected to be met in most use cases of the DRE.

In the second case, separability is broken by a requester requesting commodities across a separating partition. Again, a minimal example is shown in Figure 3.5. This case can arise in practice when modeling an NFC system where both a reactor and a repository compete for some commodity. While this is a valid modeling case under certain assumptions and simplifications, it is not very realistic. In general fuel that can be used by a reactor has been processed differently than material to be sent to a repository. If an instance of a DRE does not meet this requirement, it will not be able to be subdivided into smaller instances.

Because the majority of fuel cycles analyzed will meet both conditions, most of DRE instances will be able to be separated into at least two distinct instances which can solved independently of one another.

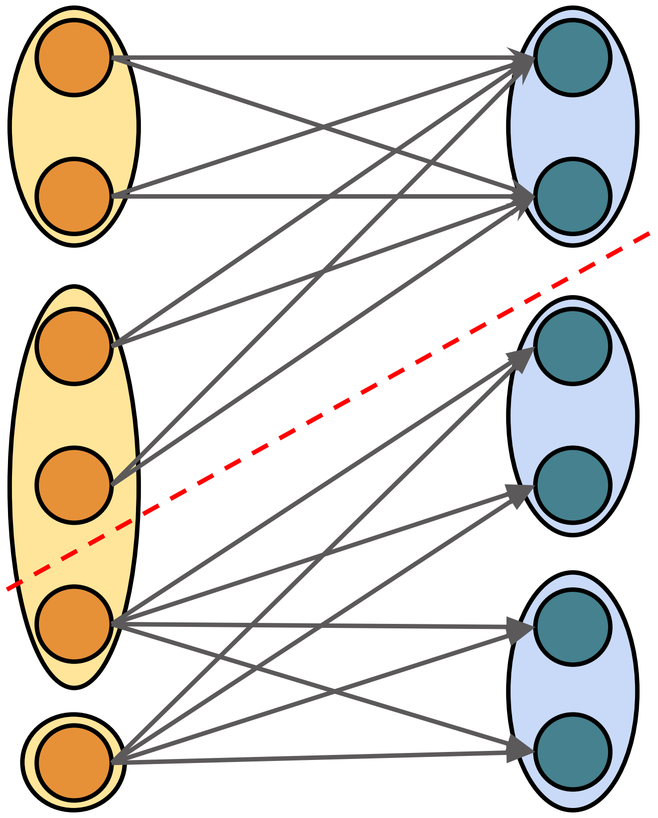

Figure 3.3: An Exchange Graph with nodes grouped by portfolio that is *not* separable because a portfolio crosses the node partition.

One instance will be associated with the front end of the fuel cycle where reactors are requesting fuel. The other instance will be associated with the back end of the fuel cycle, where reactors are supplying used fuel.

Separability is important to this work for two reasons. First, as described in section 1.3.3, an unmanageable problem instance that is separable can result in two (or more) manageable problem instances. If a truly inseparable cycle is modeled in practice, and it is found to be an intractable problem, heuristic solutions can be used. Second, because a given fuel cycle can be separated into its front-end and back-end components, the remainder of this works performs analyses on each component independently.

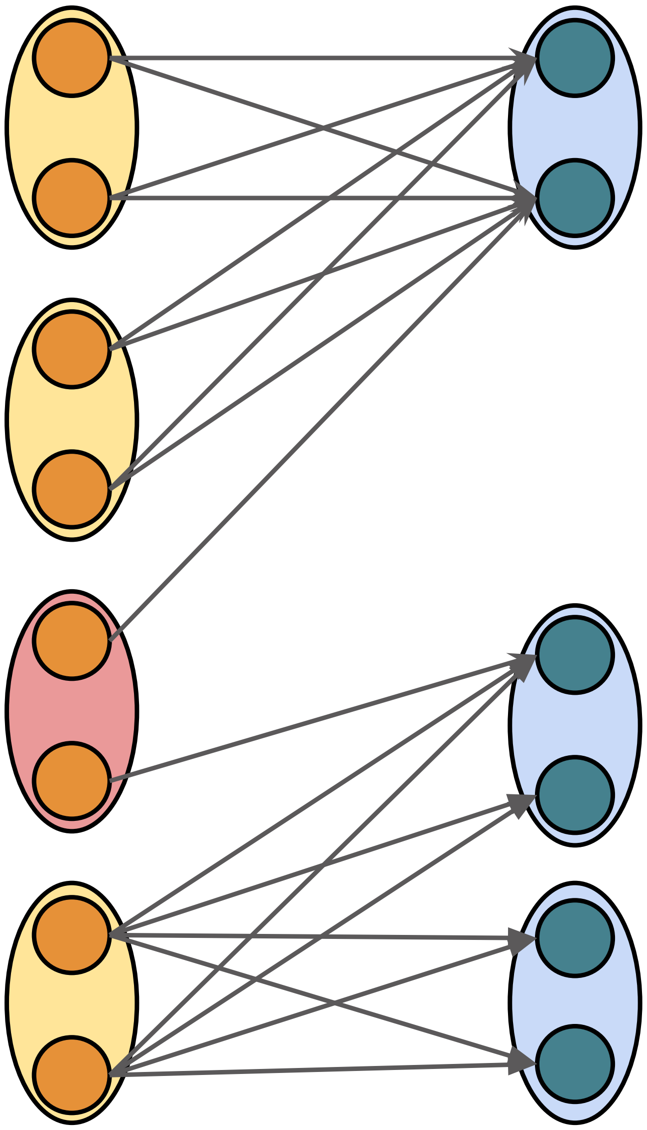

Figure 3.4: An Exchange Graph where separability broken by a supplier. This occurs in NFC modeling if assumption 1 is broken.

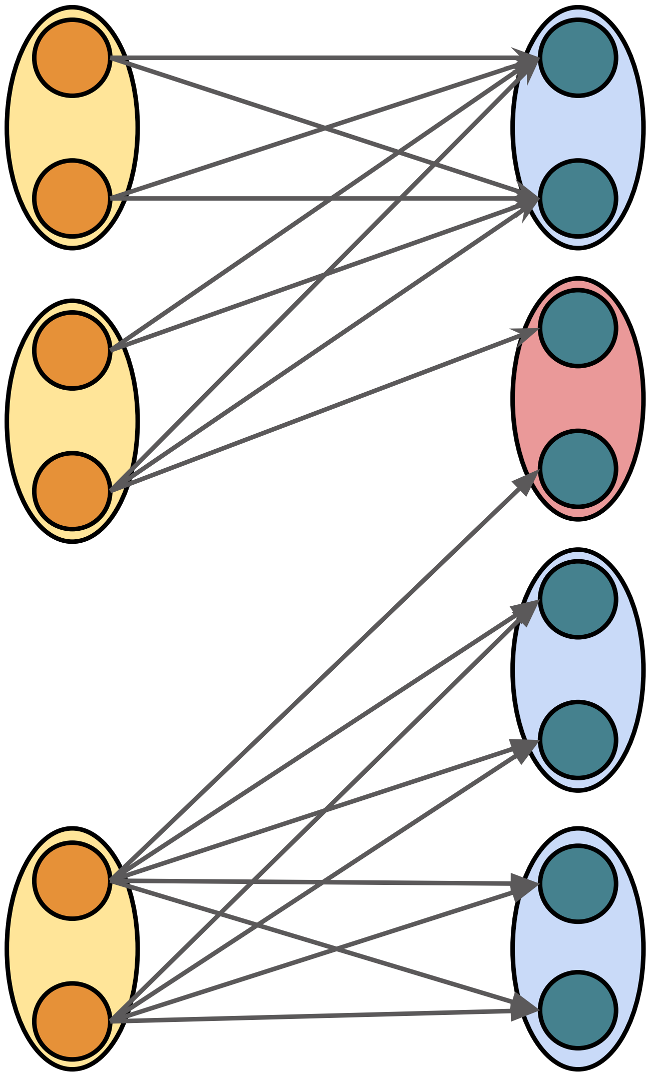

Figure 3.5: An Exchange Graph where separability broken by a requester. This occurs in NFC modeling if assumption 2 is broken.

3.1.3 Exchange Parameters

The generation of exchanges requires a set of parameters. For instance, a critical parameter is the number of reactors in an exchange. Exchange generation parameters can be divided into two classifications, *fundamental* parameters and *instance* parameters. All exchange species share some fundamental parameters and instance parameters, a discussion of which is the focus of this section. Species also define their own set of instance parameters to complete the full set of parameters needed to define an instance of an exchange.

3.1.3.1 Fundamental Parameters

The fundamental parameters are related to the common features of all instances described in section 3.1.1. Each fundamental parameter is a switch that sets the level of *fidelity* of a given exchange. As such, they are each denoted as f_x, where the x subscript describes the parameter.

The most critical parameter is related to the "fidelity" of the fuel cycle being modeled, f_{fc}. A value of zero indicates modeling the OT fuel cycle, one is used for the MOX fuel cycle, and two the ThOX fuel cycle. The parameter-to-fuel-cycle is summarized in Table 3.7. As fuel cycle fidelity increases, the number of commodities increases, and thus the number of possible connections between suppliers and consumers that exist increases, because some entities trade in multiple commodities.

Table 3.7: A mapping between fuel cycles and f_{fc} values.

Fidelity (Fuel Cycle)	f_{fc}
UOX	0
MOX	1
ThOX	2

The second parameter is reactor fidelity, f_{rx}. Reactors can make requests or provide supply based either on their entire batch or for each assembly in a batch. An $f$rx value of zero indicates reactors trading full batches, and a value of one indicates reactors trading individual assemblies. Trading individual assemblies is of higher fidelity because the number of possible trades, and thus variables in the NFCTP formulation, increases by an order of magnitude. The parameter-to-reactor-fidelity mapping is shown in Table 3.8.

Finally, the fidelity with with objective value coefficients are generated can be varied. This parameter is denoted f_{loc} because it governs the degree to which location is taken into account in Equation 3.2. The mapping between f_{loc} and parameters in Equation 3.2 is shown in Table 3.9. As f_{loc} increases, the size of

Table 3.8: A mapping between reactor fidelity and f_{rx} values.

Fidelity (Reactor)	f_{rx}
Batches	0
Assemblies	1

the distribution of possible objective coefficient values increases. When f_{loc} is zero, the number of possible objective coefficient values is equal to the product of the number of requester types and the number of commodities. Increasing f_{loc} by one, the total possible values increases by a factor of ten, because there are ten possible regional-preference values. Finally, when f_{loc} is two, the number of possible objective values is uncountably infinite [16].

Table 3.9: f_{loc} Effects on Objective Coefficient Values in Equation 3.2.

Fidelity (Location)	f_{loc}	δ_{reg}	δ_{loc}
No region or location data	0	0	0
Region data	1	1	0
Region and location data	2	1	0

3.1.3.2 Instance Parameters

Fundamental parameters represent switches that change the notion of the fidelity of the exchange being generated, for example the difference between a once-through fuel cycle and a fuel cycle with recycling. Instance parameters, on the other hand, change the *shape* and *size* of instances in a given population. In addition to an instance's shape and size, instance parameters can also affect *coefficient generation*. While fundamental parameters are related basic modeling assumptions, instance parameters are related to the specifics of an instance, given those basic modeling assumptions. Both species of exchange instances share some instance parameters, namely those related to the population of reactors in a given exchange and objective coefficient generation.

Reactor Population

Instances are broadly defined by a parameter representing the number of reactors that exist in an exchange instance, n_{rx}. Next, the split between thermal and fast reactors is defined by a parameter defining the ratio of thermal reactors to all reactors in the system, $r_{rx,\text{Th}}$. Assuming $f_{\text{fc}} > 0$, the number of thermal and fast reactors is given by

$$n_{rx,\text{Th}} = r_{rx,\text{Th}} n_{rx}, \tag{3.9}$$

and

$$n_{rx,f} = n_{rx} - n_{rx,\text{Th}}. \tag{3.10}$$

If f_{fc} is zero, a OT fuel cycle is modeled, thus n_{rx} is equal to $n_{rx,th}$ as there are only thermal reactors in the exchange. If a MOX fuel cycle is modeled, the number of FMOX reactors, $n_{rx,\text{FMOX}}$, is trivially equal to the number of fast reactors. However, for a ThOX fuel cycle, i.e., $f_{\text{fc}} > 1$, the number of FMOX and FThOX reactors is determined by a parameter defining the ratio of Thorium-fueled fast reactors to the total population of fast reactors, $r_{rx,\text{FThOX}}$, such that

$$n_{rx,\text{FThOX}} = r_{rx,\text{FThOX}} n_{rx,f} \tag{3.11}$$

and

$$n_{rx,\text{FMOX}} = (1 - r_{rx,\text{FThOX}}) n_{rx,f}. \tag{3.12}$$

In the event that the determined number of reactors is non-integral, the value is rounded to the nearest integer, with an imposed minimum value of unity.

Objective Coefficients

As shown in Equation 3.3, the value of an objective coefficient has two components, preference due to a commodity, p_c, and preference due to the relative location between two entities, p_l. It is not obvious to what degree, if any, the relative values of the two components affect formulation performance. Accordingly, a ratio parameter, $r_{l,c}$, is introduced to allow for investigating such effects.

3.1.3.3 Parameter Summary

A summary of species-independent parameters is provided in Table 3.10.

Table 3.10: Parameter Description Summary for Species-Independent Parameters.

Parameter	Type	Description
f_{fc}	Fundamental	The fuel cycle "fidelity" of an instance (which fuel cycle is being modeled).
f_{rx}	Fundamental	The reactor fidelity of an instance (whether individual assemblies are modeled or whole batches are modeled).
f_{loc}	Fundamental	The location fidelity of an instance (to what degree is facility location included in objective coefficients).
n_{rx}	Instance	The number of reactors in an instance.
$r_{rx,\text{Th}}$	Instance	The ratio of thermal reactors to all reactors in an instance, if appropriate.
$r_{rx,\text{FThOX}}$	Instance	The ratio of ThOX-based fast reactors to all fast reactors, if appropriate.
$r_{l,c}$	Instance	The weight given to location preference with respect to commodity preference.

3.1.4 Front-End Exchanges

A front-end exchange is one in which reactors request fuel and supporting facilities supply fuel resources. Given a specified reactor population, a supporting facility population is determined, as described in section 3.1.4.1. Conceptually, the information gathering procedure for this exchange begins with the RFB phase where reactors make requests for commodities with a given quantity and enrichment. Enrichment in this case is a simple resource quality proxy for an isotopic vector. Supporting facilities are then polled to provide a response to these requests during the RRFB phase. Managers of reactors would then adjust preferences based on implemented strategies. The remainder of this section describes how front-end exchange generation models the information gathering procedure, starting with the generation of requests in section 3.1.4.2, followed by the generation of supply responses in section 3.1.4.3. The PA phase is modeled using the location proxy described in section 3.1.1.4. Throughout the discussion on generating front-end exchanges, instance parameters are defined. A summary of all front-end specific instance parameters is described in section 3.1.4.4.

3.1.4.1 Support Facility Population

It is assumed that there is a single type of support facility, or supporter, for each type of commodity used in the fuel cycle. Further, each supporter is paired with a reactor type, i.e., there is a reactor type which is the *primary consumer* of each supporter type. The primary consumer-supplier relationship is modeled within the formulation by choosing preferences such that there is a maximum preference for the provided relationship (described in the following sections). A summary of these relationships is provided in Table 3.11.

Table 3.11: A mapping between commodities and the supporter type of that commodity.

Commodities	**Supporter**	**Primary Consumer**
UOX	UOX	Thermal
TMOX	TMOX	Thermal
FMOX	FMOX	FMOX
FThOX	FThOX	FThOX

The number of each type of supporter in a front-end exchange instance is a function of of the number of primary consumers as well as configurable parameters. Supporter types are divided into two groups: those who primarily support thermal reactors and those who primarily support fast reactors. The number of thermal fuel supporters is determined to be the product of the number of thermal reactors and a ratio parameter, $r_{s,\text{Th}}$, i.e.,

$$n_{s,\text{Th}} = r_{s,\text{Th}} n_{rx,\text{Th}}. \tag{3.13}$$

The number of TMOX supporters, assuming $f_{\text{fc}} > 0$, is then determined by a parameter defined as the ratio of TMOX to UOX supporters, $r_{s,\text{TMOX,UOX}}$, such that the number of UOX and TMOX supporters is

$$n_{s,\text{UOX}} = \frac{n_{s,\text{Th}}}{1 + r_{s,\text{TMOX,UOX}}} \tag{3.14}$$

and

$$n_{s,\text{TMOX}} = n_{s,\text{Th}} - n_{s,\text{UOX}}. \tag{3.15}$$

The number of fast reactor fuel supporters is determined directly from the number of associated fast reactors in the exchange using ratio parameters, $r_{s,\text{FMOX}}$ and $r_{s,\text{FThOX}}$. Assuming $f_{\text{fc}} > 0$, the number of FMOX supporters is given as

$$n_{s,\text{FMOX}} = r_{s,\text{FMOX}} n_{rx,\text{FMOX}}. \tag{3.16}$$

Similarly, assuming $f_{\text{fc}} > 1$, the number of FThOX supporters is given as

$$n_{s,\text{FThOX}} = r_{s,\text{FThOX}} n_{rx,\text{FThOX}}. \tag{3.17}$$

3.1.4.2 Request Generation

Reactors make *mutual* requests for all commodities that they can consume as described in Table 3.12. Again, a mutual request set is a group of requests of which any single request will meet a given demand. When reactors request a single batch, i.e., when f_{rx} is zero, a single request is made per commodity. When requesting n_a assemblies, a request is made per assembly per commodity, with the number of assemblies denoted previously in Table 3.2. A single request portfolio encompasses all requests, with a portfolio quantity equal to the reactor's batch size.

It is assumed that fuel is requested at some enrichment level dependent on the reactor type. Each reactor in an exchange will choose a batch enrichment level given a uniform distribution. Recycled fuel is modeled as being composed of a target element oxide and topped up with natural uranium oxide; the mixing ratio is again based on reactor type. For recycled fuel, the associated enrichment level describes the enrichment of the fissile isotope in the *target* element. For example, MOX fuel with 45% enrichment implies that of the elemental Plutonium in the mixture, 45% is comprised of isotopic ^{239}Pu . Finally, each reactor has a preference assignment over its set of consumable commodities.

Thermal reactors can consume UOX fuel as well as both MOX variants. It is assumed that thermal reactors would prefer to consume thermal MOX fuel in order to maintain any equilibrium status of the cycle. UOX fuel is next preferred. Finally "fast" MOX is modeled as a type of fuel that is usable by thermal reactors, i.e., it has thermally-fissile plutonium; however it is assumed that the property of the plutonium vector is more amenable to fast-spectrum reactors. Therefore it is least preferential. Preference values for each commodity are summarized in Table 3.12. A normal operating enrichment range of $[3.5, 5.5]$ is used for UOX fuel. MOX-based fuels are assumed to be comprised of 7% Plutonium-oxide with 93% Uranium-oxide top up [12] and an enrichment range of $[55, 65]$ [10]. In practice, many reactor concepts restrict the fraction of an LWR's core that can be made up of MOX fuel rather than UOX fuel due to a reduced safety margin. Accordingly, a tuneable parameter is added to the model, f_{mox}, which denotes the fraction of a request that can be made up of MOX-based fuel. This fraction is only relevant if reactors are operating in assembly mode, i.e., if f_{rx} is unity.

MOX and ThOX fast reactors utilize the same governing request parameters but have a different preference distribution over commodities. It is assumed that a Thorium-based fast reactor prefers Thorium-based fast reactor fuel over MOX-based fast reactor fuel and *vice versa*. Additionally, both fast reactor types can utilize thermal MOX fuel or medium-enriched UOX, but prefer fast reactor-based fuels. Preference values for each commodity type and reactor are summarized in Table 3.12. Both fast reactor types select a UOX enrichment in $[15, 20]$[10], with an upper limit set by LEU legal enrichment limits. All recycled fuels commodities are assigned an enrichment range of $[55, 65]$ [10] and have a composition of 20% of the target element (Plutonium or Thorium), with the given enrichment of it's primary fissile isotope (^{239}Pu or ^{233}U), and 80% Uranium top up [10]. Note that no large-scale fast reactors have been fueled by Thorium-based fuels. Accordingly, Thorium-related values are broad generalizations. The purpose of including another fuel type is to expand on the complexity of possible connections between facilities in a given fuel cycle, while including somewhat realistic constraining values. The constraining values used by any individual analyst will vary, perhaps greatly, and thus only reasonable values are required by this study.

A summary of chosen chosen request parameters based on reactor and commodity types is shown in Table 3.12.

Table 3.12: A summary of reactor request parameters.

Reactor Type	Commodity	Enrichment Range	Target Element Fraction (%), f_{el}	Commodity Preference, p_c
Thermal	UOX	$[3.5, 5.5]$	100	0.5
	TMOX	$[55, 65]$	7	1
	FMOX	$[55, 65]$	7	0.1
FMOX	UOX	$[15, 20]$	100	0.1
	TMOX	$[55, 65]$	20	0.5
	FMOX	$[55, 65]$	20	1
	FThOX	$[55, 65]$	20	0.25
FThOX	UOX	$[15, 20]$	100	0.1
	TMOX	$[55, 65]$	20	0.25
	FMOX	$[55, 65]$	20	0.5
	FThOX	$[55, 65]$	20	1

3.1.4.3 Supply Generation

With all requests known, each supporting supply facility responds to all requests for their assigned commodity, creating an associated supply node and an arc between the supply node and request node. Constraint coefficients are determined for each arc based on the requested enrichment associated with that arc. Furthermore, a right-hand side (RHS), b_s^k, is provided for each constraint in addition to a coefficient conversion function.

Each supplier has two types of constraints for which coefficients must be calculated: a *process* constraint and an *inventory* constraint. A process constraint models a situation in which the amount of supplied fuel is constrained physically; only so much fuel can be made in one time step. An inventory constraint models a situation in which a supplier is constrained by the available material inventory on hand. Both constraints are a function of requested quantity and fuel enrichment.

UOX Constraints

A UOX supplying facility is assumed to be constrained by a SWU process constraint and a natural Uranium inventory constraint. Assuming general operating parameters, including a tails assay of 0.3 and a feed assay of natural Uranium, 0.711, constraint coefficients can be applied to arcs. The SWU coefficient conversion function is previously described in Equation 2.15 while the natural Uranium conversion function is described in Equation 2.14. Therefore, for UOX supplying facilities,

$$\beta_s^{\text{proc}}(\epsilon) = \beta_s^{\text{SWU}}(\epsilon) \tag{3.18}$$

and

$$\beta_s^{\text{inv}}(\epsilon) = \beta_s^{\text{NU}}(\epsilon). \tag{3.19}$$

In order to determine a constraining RHS, the proposed Eagle Rock Enrichment Plant is chosen as a model. It purports to have a SWU capacity of $3.3E6$ Million SWU per year. Accordingly, the process constraint RHS is chosen to be an approximate monthly value,

$$b_s^{\text{SWU}} =\sim 2.75E5 \frac{\text{SWU}}{\text{month}}. \tag{3.20}$$

Any inventory constraint will be based on the present state of a facility at a given simulation time step. Therefore, a sufficiently reasonable value must be provided without actual simulation data. Because two constraints are added, investigating their relative effects is of interest, which leads to a strategy for generating an inventory constraining value by deriving it from the process constraining value. In order to make such comparisons, the two RHS values must be equivalent in both units and with respect to the expected coefficient values associated with each constraint. Accordingly, a translation constant is defined to achieve both aims. The translation , τ_s, constant is taken to be a ratio of constraint coefficients for the average enrichment of a support facility's primary consumer, i.e.,

$$\tau_s = \frac{\beta_s^{\text{inv}}(\bar{\epsilon_r})}{\beta_s^{\text{proc}}(\bar{\epsilon_r})}. \tag{3.21}$$

Thus, a UOX supporting facility uses a average enrichment, $\bar{\epsilon_r}$, of 4.5 because that is the median of the thermal reactor enrichment range. Further, a ratio coefficient parameter, $r_{inv,proc}$, is added in order to investigate interesting cases from a formulation point of view. If $r_{inv,proc} > 1$, then the process constraint RHS is smaller and thus the process constraint is more likely to be engaged in an feasible solution than the inventory constraint. On the other hand, if $r_{inv,proc} < 1$, the inventory constraint is more likely to be engaged. The determination of the inventory RHS is identical for all supporting facilities and is defined in Equation 3.22.

$$b_s^{\text{inv}} = r_{inv,proc} \tau_s b_s^{\text{proc}}. \tag{3.22}$$

Recycled Commodity Constraints

Due to the lack of commercially viable, well documented fast reactor fuel suppliers, a simple linear surrogate model is assumed for an inventory constraint. The primary inventory of any recycling facility is the amount of fissile material it has on hand. Therefore, using constants defined in Table 3.12, the coefficient function conversion function is chosen to be

$$\beta_s^{\text{inv}}(\epsilon) = f_{el}\epsilon. \tag{3.23}$$

There are many possible process constraints that could be used, such as heat production or radiotoxicity; however, each of these requires a detailed isotopic composition to be relevant. Accordingly, a commodity-informed mass throughput constraint is used. Per the current IAEA practice [33], and extrapolating the same effect for reprocessing ^{233}U , a factor of 100 is added for for Plutonium and Thorium-based commodities. The process constraint coefficient function is defined as

$$\beta_s^{\text{proc}} = 100 f_{el}. \tag{3.24}$$

From previous conversations with industry representatives [46], a reasonable size for a processing plant is 800 tonnes per year, which is similar to the Rokkasho plant in Japan [33]. Given the request parameters defined in Table 3.12, an 800 t Uranium / 8 t Plutonium facility could service on the order of 2-3 fast reactors or ∼2 thermal reactors with $\frac{1}{3}$ a request as MOX. The yearly process limit is again translated to a monthly limit, resulting in a constraint RHS value of

$$b_s^{\text{proc}} =\sim 66.7 \frac{\text{t}}{\text{month}}. \tag{3.25}$$

The inventory constraint RHS is determined identically to the UOX case.

3.1.4.4 Parameter Summary

A summary of front-end exchange species-dependent instance parameters is provided in Table 3.13.

Table 3.13: Parameter Description Summary for Front-End Exchange Instance Parameters.

Parameter	Description
f_{mox}	The fraction of thermal reactor requests that can be met with mox fuel.
$r_{s,\text{Th}}$	The ratio of thermal support facilities to thermal reactors.
$r_{s,\text{TMOX,UOX}}$	The ratio of TMOX to UOX support facilities.
$r_{s,\text{FMOX}}$	The ratio of FMOX support facilities to FMOX reactors.
$r_{s,\text{FThOX}}$	The ratio of FThOX support facilities to FThOX reactors.
$r_{inv,proc}$	The ratio of the inventory RHS to the process RHS.

3.1.5 Back-End Exchanges

A back-end exchange models the transfer of used fuel from reactors to supporting facilities, such as reprocessing facilities and repositories. During the information gathering process, supporting facilities make requests for commodities that can either be used directly in the recycling process or need to be stored, temporarily or permanently. Reactors then respond based on output fuel to each request during the RRFB phase. Throughout the discussion on generating back-end exchanges, instance parameters are defined. A summary of all back-end specific instance parameters is shown in Table 3.15.

3.1.5.1 Support Facility Population

Four classes of supporting facilities are modeled in back-end exchanges: a thermal fuel recycling facility, a facility that recycles fast MOX fuel, a facility that recycles fast ThOX fuel, and a repository. As thermal fuel recycling facilities are the only thermal supporting facilities, the number of such facilities in a given back-end exchange is trivially $n_{s,\text{Th}}$. As is the case with front-end exchanges, there is a class of supporting facility for each fast fuel commodity. The methodology for determining the population of each facility type is identical to front-end exchanges:

$$n_{s,\text{FMOX}} = r_{s,\text{FMOX}} n_{rx,\text{FMOX}} \tag{3.26}$$

and

$$n_{s,\text{FThOX}} = r_{s,\text{FThOX}} n_{rx,\text{FThOX}}. \tag{3.27}$$

Back-end exchanges include repositories, a facility type not present in front-end exchanges. A simple ratio parameter, r_{repo} is applied based on the total number of other supporting facilities, i.e.,

$$n_{s,\text{repo}} = r_{s,\text{repo}}(n_{s,\text{Th}} + n_{s,\text{FMOX}} + n_{s,\text{FThOX}}). \tag{3.28}$$

3.1.5.2 Request Generation

It is assumed that any recycling facility will accept UOX fuel. However, MOX recycling facilities can not process ThOX-based fuels, and ThOX facilities can not process MOX-based fuels. Additionally, fast MOX facilities prefer fast MOX fuel, while thermal facilities prefer thermal MOX fuel. Finally, repositories can accept all commodities; however, it is a consumer of last resort. The assigned preference value as a function of commodity type and supporting facility type, p_c is shown in Table 3.14.

Table 3.14: p_c Value Mapping between Back-End Supporting Facilities and Commodities.

Commodity / Supporting Facility	UOX	TMOX	FMOX	FThOX
TMOX	1	1	0.5	N/A
FMOX	0.5	1	1	N/A
FThOX	0.3	N/A	N/A	1
Repo	0.1	0.1	0.1	0.1

A single request for the facility's processing capacity is made for each commodity. Recycling facilities define their request quantity using the same 800 ton per year limit discussed in section 3.1.4. Repositories, however, use a limit based on the Yucca Mountain statutory limit of 77,000 tons and assuming a 30-year operating lifetime, i.e., period of time in which fuel can enter the facility. Thus, a repository's monthly request quantity is determined to be ~ 215t.

A fissile quantity constraint is added for each recycling facility. The fissile constraint models a situation in which recycling facilities have a demand for fissile material. The amount of fissile material required by recycling facilities is based on their primary consumer. It is determined to be the product of the facility's mass constraint and the mean amount of fissile material in a primary consumer's request per unit mass, as shown in Equation 3.29. This constraint can be considered as "recycling facilities request fissile material quantities as if all reactors in the system are average primary consumers".

$$b_r^{\text{fiss}} = \bar{\epsilon} f_{el} b_r^{\text{mass}}. \tag{3.29}$$

The fissile constraint coefficient is simply the amount of fissile material for a given supply, as described in Equation 3.30.

$$\beta_r^{\text{fiss}}(\epsilon) = \epsilon f_{el}. \tag{3.30}$$

3.1.5.3 Supply Generation

A key difference between the front-end and back-end exchanges is that in front-end exchanges, reactors request fuel, and thus can make a single request per commodity per assembly. In back-end exchanges, commodities must be assigned to each assembly. Accordingly, a key parameter in back-end exchanges is the commodity distribution for assemblies. A normalized uniform distribution parameter is provided for each reactor type with a value for each commodity type that reactor can consume as defined in Equation 3.31.

$$
\begin{aligned}
d_{\text{Th}} &= [x_{\text{UOX}}, x_{\text{TMOX}}, x_{\text{FMOX}}],\ x_i \in [0, 1) \\
d_{\text{FMOX}} &= [x_{\text{UOX}}, x_{\text{TMOX}}, x_{\text{FMOX}}, x_{\text{FThOX}}],\ x_i \in [0, 1) \\
d_{\text{FThOX}} &= [x_{\text{UOX}}, x_{\text{TMOX}}, x_{\text{FMOX}}, x_{\text{FThOX}}],\ x_i \in [0, 1)
\end{aligned}
\tag{3.31}
$$

If an exchange is in batch mode, i.e., f_{rx} is zero, then this distribution acts as a selection distribution, where each x_i represents a probability that the batch will be of that commodity. If in assembly mode, then commodities are assigned to each assembly given the relative x_i values. The assignment of commodities to number of assemblies for a given reactor type is done by rounding the product of x_i and the total number of assemblies, starting with the lowest value of x_i. The final assignment is then taken as the difference between the total number of assemblies and the previously assigned values.

For example, consider a fast reactor with a distribution $d_{\text{Th}} = [\frac{3}{4}, \frac{1}{4}, 0]$ and number of assemblies $n_a = 39$. The assembly-commodity breakdown would be calculated as

$$
\begin{aligned}
n_{\text{FMOX}} &= \text{round}(x_{\text{FMOX}} n_a) = 0 \\
n_{\text{TMOX}} &= \text{round}(x_{\text{TMOX}} n_a) = 10 \\
n_{\text{UOX}} &= n_a - n_{\text{TMOX}} - n_{\text{FMOX}} = 29.
\end{aligned}
$$

Once a commodity is assigned either to a single batch or a selection of assemblies, the remaining supply generation methodology is identical. If f_{rx} is zero, the following discussion uses the term assembly to mean either an individual assembly or a batch. That is, a reactor in a back-end exchange has a single assembly to supply. If f_{rx} is one, then it has n_a assemblies to supply, where n_a is defined in Table 3.3 for each reactor type.

In order to assign enrichment values to each assembly, a single random value is chosen, $x \in [0, 1)$. Each assembly is then assigned an enrichment based on the assembly's commodity and enrichment range, as defined in Table 3.12. This modeling assumption supports a situation in which, for a given batch, equivalent fissile enrichments were used across commodities. For example, consider a Thermal reactor with x chosen to be 0.55. All UOX assemblies would be assigned an enrichment value of 4.6, and each

MOX-based assembly would be assigned an enrichment value of 60.5.

A bid portfolio is assigned to each assembly. Given the commodity of each assembly, a supply response is provided to each supporting facility that requests that commodity. For example, given a UOX assembly, a reply is sent to each supporting facility, as all supporting facilities accept UOX, shown in Table 3.14. The set of supply responses associated with a single assembly is denoted a *mutual* set. That is, each supply node corresponds to a single assembly that should not be split between supporting facilities.

3.1.5.4 Parameter Summary

A summary of back-end exchange species-dependent instance parameters is provided in Table 3.15.

Table 3.15: Parameter Description Summary for Back-End Exchange Instance Parameters.

Parameter	Description
d_{Th}	thermal reactor assembly distribution
d_{FMOX}	fast mox reactor assembly distribution
d_{FThOX}	fast thox reactor assembly distribution
r_{repo}	repository to supporting facility ratio

3.2 Experimental Tools

In order to explore the large number of possible exchange instances described in section 3.1, a sophisticated instance generation and solving framework is needed. This section describes the design principles and implementation details of a new software package called Cyclopts (Cyclus Optimization Studies). Cyclopts, written primarily in Python with a C++ layer used to interface with Cyclus, provides a general framework for sampling a parameter space, defining problem instances for a given point in parameter space, and solving a problem instance under a variety of conditions. While this section focuses on the Cyclopts workflow, implementation, and high-throughput computing (HTC) capabilities, details specific to the database layout and command line interface (CLI) are treated lightly. A full treatment of the the database layout is provided in Appendix C, and the CLI is detailed in Appendix D.

3.2.1 Terminology

Cyclopts supports a two-tier definition of problem instances, borrowing terms from biological classification. Problem *families* describe a general form of problem instance. For example, the Traveling Salesman Problem

(TSP) could be implemented as a problem family. In this analysis, a single problem family, the NFCTP, is investigated. The NFCTP can be considered a problem family because any given instance of the NFCTP will have the same general structure. Whether or not the LP or MILP formulation is used is dependent on whether or not arcs in the Exchange Graph are labeled as exclusive or not. If there are no exclusive arcs, the LP formulation is used; otherwise, the MILP formulation is used.

Each problem family can have any number of *species*. One can conceptualize the relationship as a tree structure, in which families are parent nodes and species are child nodes. A problem species defines the methodology for generating *instances* of a problem family. Using the TSP example above, a problem species may be "the greater Atlanta metropolitan area", for which the effect of regional gas prices may be studied. For the NFCTP study, front-end and back-end exchanges form two separate species. Each species can have unique parameters in addition to family-related parameters, as is the case for the two species studied.

3.2.2 Design

The full Cyclopts stack is comprised of three phases: generation of parameter space, generation of instances, and execution of instances. The workflow begins with user input detailing a range of values for a set of parameters. Cyclopts then translates the input into a parameter space by enumerating all possible combinations of parameters. For example, if parameters x and y have defined values of $[1, 2]$ and $[3, 4, 5]$, respectively, Cyclopts will generate a parameter space comprised of six points in (x, y) notation: $(1, 3)$, $(1, 4)$, $(1, 5)$, $(2, 3)$, $(2, 4)$, and $(2, 5)$. Each point is then then provided to a problem species in order to generate one or more problem instances. Species are expected to define defaults for all parameters as user input may define values for only a subset of available parameters.

Given a point in parameter space, an instance can be generated. If there are any stochastic effects during instance generation, many instances may be generated. Again, because parameters are species dependent, the logic of instance generation from a set of parameters is the task of a problem species. Following instance generation, instances can be executed. Cyclopts supports multiple solution options by design. The same instance may be solved with both a heuristic and a full optimization solver, for example. Once an instance of a problem is defined, it is independent of any species-level effects. Accordingly, instance execution and related logic is the domain of problem families.

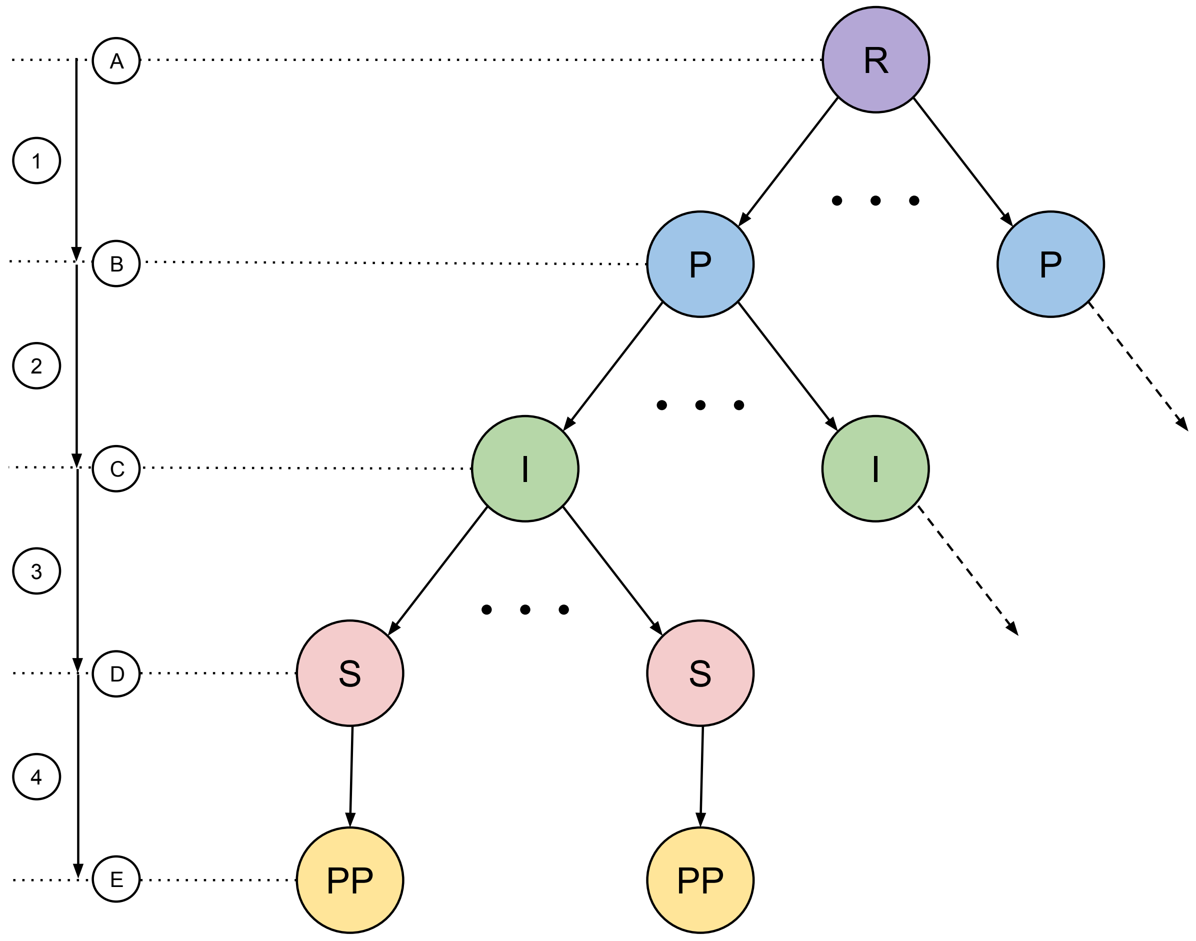

Figure 3.6: A graphical representation of the Cyclopts object tree-structure. For any parent node, there is a one-to-many mapping of children nodes. The node types in the tree structure are defined in Table 3.16. The actions associated with moving between each level are explained in Table 3.17.

A summary of the high-level Cyclopts workflow and entities is presented in Figure 3.6. Note that objects generated as the workflow moves from parameter space to instance solution form a tree structure.

Table 3.16: Cyclopts object tree structure node types as shown in Figure 3.6.

Label	**Node Type**	**Description**
A	Root	A definition of the full parameter space as provided by a user.
B	Parameter	A fully defined point in the parameter space.
C	Instance	A fully defined instance of a given problem.
D	Solution	A solution to an instance of a problem determined by an appropriate solver.
E	Post-process	Post-processed information, given all parent nodes.

Table 3.17: Cyclopts actions generating child nodes as shown in Figure 3.6. The Cyclopts entity, e.g., the family or species, associated with each action is also listed.

Label	Action	Entity Responsible
1	Translate a parameter space into all possible points.	Cyclopts Core
2	Convert a parameter point into a number of problem instances.	Problem Species
3	Execute a problem instance given a solver and record the solution.	Problem Family
4	Post-process a solution and instance, recording relevant information.	Problem Family & Species

3.2.3 Persistence Mechanisms

While the root node in Figure 3.6 is generated from a user-provided input file, each subsequent level in the hierarchy represents a stateful object: a point in parameter space, a problem instance, and a solution. Each stateful object can be written to and read from disc. Cyclopts also incorporates a post-processing step, during which all related objects may be analyzed and aggregate data may be collected and written to disc. While any input/output (I/O) persistence mechanism is valid, Cyclopts is currently implemented using the Hierarchical Data Format (HDF5) [59] via PyTables [8].

Data in HDF5 is stored hierarchically, similar to a file system. At the root node of the file-system-like structure, a *group* is defined for problem family and problem species data, named `Family` and `Species`, respectively. A *dataset* for aggregate results named `Results` is also defined. A path in HDF5 is designated in a UNIX-like manner. For example, the path to `Family` would be `/Family`, indicating that the group is directly under the root node, `/`. Further, groups are defined for each kind of family and species. The DRE problem family records data in the group `/Family/ResourceExchange`, front-end exchanges record data in the group `/Species/StructuredRequest`, and back-end exchanges record data in the group `/Species/StructuredSupply`.

Each stateful object is given a Universally Unique Identifier (UUID) by which it can be identified for future reading and analysis. The UUID is used in two distinct capacities: as a *primary key* in a dataset for future identification or as the name of a group. Whether to aggregate data in one large dataset or divide data into datasets for each object is a design decision informed by practical performance. A study of the trade-offs between each approach is presented in section C.5. As a result of that study, for objects that are both read and written, the latter approach is taken.

A description of all data gathered for each family and species for conversion, execution, and post-processing, as well as a I/O performance study, is detailed in Appendix C.

3.2.4 Implementation

Cyclopts defines abstract application programming interfaces (APIs) for both families and species in the `ProblemFamily` and `ProblemSpecies` classes, respectively. While many parts of an API are related to the workflow discussed in section 3.2.2, others are related to the persistence mechanisms discussed in section 3.2.3. The full API is described in detail in the Cyclopts documentation [26]. The `ExchangeFamily` class implements a concrete, NFCTP-specific `ProblemFamily` interface. The `StructuredRequest` class implements a concrete, front-end-exchange interface of the `ProblemSpecies` class. Similarly, the `StructuredSupply` implements a back-end interface to the class.

Given a point in parameter space, both the `StructuredSupply` and `StructuredRequest` generate an instance of an `ExchangeGraph` per the rules described in section 3.1. Cyclopts is nominally written in Python and Cyclus is written in C++. In order to construct objects that can interact with Cyclus, an interoperability layer is required.

A series of C++ wrapper objects, namely arc, node, and group objects, are defined which mirror the constituents of an `ExchangeGraph`, as described in section 2.3.4. These objects are then translated into Cython [11] by use of the XDress software package [52]. Python can directly call into Cython libraries, similarly, Cython can directly call C and C++ libraries. Hence, an interoperability layer is established.

3.2.4.1 Solvers and Performance Timing

Once an instance of an `ExchangeGraph` has been generated, it can be solved. Cyclopts supports three types of solvers: Coin linear programming (Clp), Coin branch and cut (Cbc), and the `GreedySolver`, an implementation of the Greedy Heuristic in Cyclus. If either the Greedy or Cbc solvers are invoked, an appropriate instance of a `ExchangeSolver` is constructed with the *exclusive orders* flag turned on. If the CLP solve is invoked, an associated `ExchangeSolver` instance is constructed with the *exclusive orders* flag turned off. In short, Cbc and Greedy solvers solve the MILP formulation of the NFCTP, and the CLP solver solves the LP formulation.

Given an instance of an `ExchangeGraph` and `ExchangeSolver`, the `Solve` method of the `ExchangeSolver` is invoked. Before and after the `Solve` function call, the `CoinCpuTime()` function is called and the result is stored. The difference between the two resulting values is recorded as the time required to determine a solution. The implementation of the `CoinCpuTime()` function is open and easily available [24]. It simply adds the seconds and microseconds fields of the `ru_utime` structure populated by the standard UNIX `getrusage()` function.

3.2.5 High Throughput Computing

Cyclopts can be executed locally using the `cyclopts exec` command-line interface (CLI) described in Appendix D. When exploring a large parameter space, of which each point can generate a large number of unique instances, local execution on a single machine is insufficient. In order to overcome this limitation, support for HTCondor-based systems has been implemented in Cyclopts and available using the `cyclopts condor-submit` CLI. HTCondor [58] is a high throughput computing (HTC) framework that supports sophisticated job scheduling over a very large, distributed network of individual and clustered computers.

HTC systems are ideal for analyses in which many independent executions must be performed. Upon completion, the results may be aggregated and analyzed. The resource-exchange use case fits such a design specification with a single caveat: because it is a first-of-a-kind performance analysis, timing results are crucial. Therefore, the systems on which instances are executed must be equivalent in order to compare different timing results. Support is provided in Cyclopts for identifying execute nodes that conform to a series of architecture and related constraints in order to support this analysis limitation.

3.2.5.1 Remote Execution and Operation

In order to efficiently schedule a large number of optimization problems, the WorkQueue framework [15] is utilized. WorkQueue is a HTCondor-aware master-worker implementation. A master process exists at some location and manages the scheduling jobs to be run. Workers, in the form of persistent HTCondor jobs, ask the master for the next job to be run after a previous job has been completed. A master-worker system is especially useful in HTCondor environments in which resources are limited and must be specifically targeted, as is the case with the aforementioned timing studies.

Cyclopts launches a master process that requires a series of execute nodes to target, a problem instance database, and a list of solvers to execute on each instance. A copy of the instance database is sent to every targeted execute node. Note that an execute node may have many execution threads, each of which can be used to execute instances individually. The master manages an instance queue from which jobs are provided to workers. Upon completion, a worker will request a new instance of the master. The full set of instances in a database are hence efficiently executed.

3.2.5.2 Packaging and Environment

A common issue in remote execution environments is package dependency. When access to an execution node is provided, a user must assume that only the barest of environments exist. For example, if a user is provided an Ubuntu-based execution node, the user generally must assume that it is a fresh Ubuntu installation. Accordingly, package management in a highly distributed, heterogeneous environment is a difficult problem.

Luckily, solutions exist for distributed package management. Cyclopts utilizes the Code, Data, and Environment (CDE) [32] tool to manage its execution environment. CDE provides a virtualized environment based on the local execution of a command. Using CDE with the given command, all libraries and utilities used during the process execution are monitored. Upon process exit, every object in the filesystem that was invoked is copied into a virtual environment. That virtual environment can then be packaged and distributed. Upon landing on a foreign system, a user can enter the CDE environment and execute the supported command.

Cyclopts provides a CLI, `cyclopts cde`, that will package Cyclopts itself into a virtual environment and ship the environment to a HTCondor submit node. As part of the Cyclopts HTCondor job execution, the CDE environment is copied to each execute node. It is therefore easy to incorporate changes in a local copy of Cyclopts to the corresponding remote execution.

4 Experiments & Results

Given a parameterized framework for generating instances of resource exchanges, experiments are designed and executed to explore the efficiency and quality of solutions provided by different solvers. Section 4.1 describes the experimentation apparati, including the computational tools, solvers and relevant output. Two experimental campaigns were conducted. A scaling campaign, described in section 4.2, was performed in order to investigate formulation behavior as a function of problem size. Section 4.3 then describes the results of a stochastic campaign. Finally, a summary of findings is provided in section 4.4.

4.1 Experimental Setup

An experiment consists of a set of resource-exchange graph instances executed with a collection of configured solvers. When a solution is found, the solution (i.e., the flow vector), the time required to reach the solution, the objective value (i.e., the dot-product of cost and flow vectors), and the simulation objective value (i.e., the dot-product of preference and flow vectors), are recorded. Because solution time is a quantity of interest, all instances in an experiment must be executed on homogeneous architecture. Furthermore, all experiments must be executed on equivalent systems in order to quantify valid comparisons in solutions times across experimental campaigns.

Six execution nodes on UW-Madison Center for High-Throughput Computing (CHTC) HTCondor system form the homogeneous environment used to conduct the experiments herein described. Each execute node is comprised of a 2.90 GHz eight-core Intel Xeon E5-2690 [2] processor with 128 GB of RAM. Processor hyper-threading was disabled for the duration of the experimental campaign to allow comparisons between solution times.

For each experimental study, an input database consisting of persisted resource exchange graph instances is generated. A copy of the database is transferred from a user's submit node to each of the

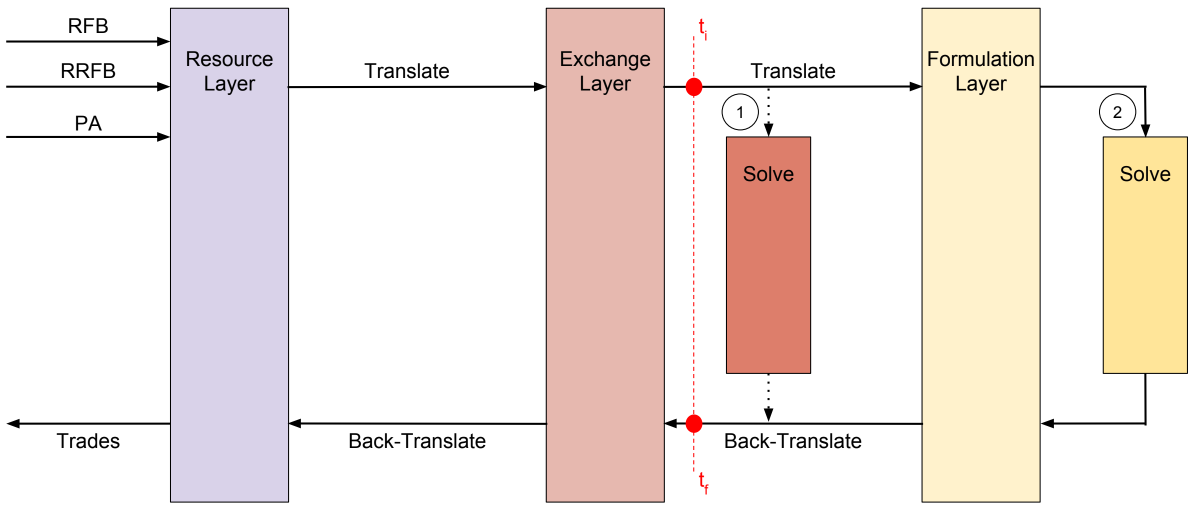

Figure 4.1: The time points for comparing different solutions using Equation 4.1.

six execution nodes. A WorkQueue master process is initiated. For every execute node, workers are initialized using WorkQueue's `condor_submit_workers` CLI. The master maintains a queue of instances to be solved, assigning instances to workers as workers become available. Upon completion, the input database is removed from each execution node, and the results are collected from the user's submit node. The resulting database is then post-processed and analyzed.

4.1.1 Solvers and Formulations

Three solvers are executed for each resource exchange graph instance: the Greedy Heuristic, described in section 2.3.3.6, COIN's LP solver (Clp), and COIN's branch-and-cut solver (Cbc). Each problem instance is constructed as an `ExchangeGraph`, i.e., at the *exchange layer* shown in Figure 2.13 and Figure 4.1. The Greedy Heuristic is applied directly to the `ExchangeGraph`. The Clp and Cbc solvers require a translation to the *formulation layer*. The Clp solver is applied to the LP formulation of the NFCTP and the Cbc solver is applied to the MILP formulation. The solution time, t_s of a given solver is defined as the time required to return a vector of arc flows given an `ExchangeGraph` instance as shown in Figure 4.1.

$$t_s = t_f - t_i \tag{4.1}$$

The Greedy Heuristic has linear-like scaling. Each request portfolio is treated in the algorithm. For each request node in the portfolio, incoming arcs are sorted, which is a $\mathcal{O}(n \log n)$ operation. The algorithm terminates when each request portfolio has either been satisfied or found to be unsatisfiable. In the worst case, every node is perused, resulting in $\mathcal{O}(n \log \frac{n}{N})$, with n arcs and N nodes. This scaling is better than $\mathcal{O}(n \log n)$ because arcs in the system are partitionable. As both Clp and Cbc solve NP-Hard problems, there is no *a priori* expected performance behavior. However, LP problems solved with the Simplex method are known in practice to scale by the number of constraints in the system. As described in Appendix A, the method iterates over vertices of the polytope defining a instance's feasible space, which are defined by constraints. MILP problems must be solved by enumeration, as shown in Appendix B, thus their scaling is chiefly a function of the number of instance variables.

4.1.2 System Parameters

Section 3.1.3 describes the parameters defining both Front and Back-End exchanges. Each combination of fundamental parameters represents a significant modeling assumption. Therefore, every experiment is conducted for every combination of fundamental parameters, comprising eighteen combinations in total. For each exchange type, a reference instance parameter vector is chosen.

Reference instance parameter vectors for front and back-end exchanges are shown in Tables 4.1 and 4.2, respectively. Reactor population and core composition values were chosen in line with what a user might reasonably find in a simulation, such as a 75%-25% thermal-to-fast reactor split and 33% possible MOX-fuel residency in thermal reactors. Support facility population values were chosen to model a "worst case" simulation. A "best case" simulation is one in which reactor supply and demand is evenly distributed. For instance, a supporting facility with monthly capacity values that can support two reactors a month is capable of supporting twenty-four reactors a year. A "worst case" simulation is one in which there are time steps where all reactors interact with the DRE simultaneously. Such a simulation will have a number of supporters commiserate with the total number of reactors. These parameter vectors are associated with such "worst case" time steps.

4.1.3 Analysis Metrics

The most obvious metrics to compare between solutions is the solution time, t_s, and objective function value z,

Table 4.1: Reference Values for Front-End Exchange Instance Parameters.

Parameter	Reference Value
$r_{rx,\text{Th}}$	0.75
$r_{rx,\text{FThOX}}$	0.25
$r_{l,c}$	1
f_{mox}	0.33
$r_{s,\text{Th}}$	0.08
$r_{s,\text{TMOX,UOX}}$	1.
$r_{s,\text{FMOX}}$	0.2
$r_{s,\text{FThOX}}$	0.2
$r_{inv,proc}$	1

Table 4.2: Reference Values for Back-End Exchange Instance Parameters.

Parameter	Reference Value
$r_{rx,\text{Th}}$	0.75
$r_{rx,\text{FThOX}}$	0.25
$r_{l,c}$	1
f_{mox}	0.33
$r_{s,\text{Th}}$	0.08
$r_{s,\text{TMOX,UOX}}$	1.
$r_{s,\text{FMOX}}$	0.2
$r_{s,\text{FThOX}}$	0.2
$r_{s,\text{Repo}}$	0.2
d_{Th}	UOX: 2/3, TMOX: 1/3, FMOX: 0
d_{FMOX}	UOX: 1/4, TMOX: 0, FMOX: 3/4, FThOX: 0
d_{FThOX}	UOX: 1/4, TMOX: 0, FMOX: 0, FThOX: 3/4

$$z = \sum_{(i,j) \in A} c_{i,j} x_{i,j}. \tag{4.2}$$

For any given instance, the optimal objective value, z^*, is associated with a set of flows, X^*. By definition, an optimal solution, i.e.,

$$z^* = \sum_{(i,j) \in A} c_{i,j} x^*_{i,j}. \tag{4.3}$$

will have a lower system cost than or an equivalent system cost to any feasible solution. There may be multiple possible optimal solutions. By necessity, any solution to the NFCTP will include flows along false arcs if such flows exist. One can also consider solution metrics that only account for arcs that exist in a simulation, A_{sim}. Because the simulation operates in preference-space, a "simulation objective" is defined as

$$z_{\text{sim}} = \sum_{(i,j) \in A_{\text{sim}}} p_{i,j} x_{i,j}, \tag{4.4}$$

and the simulation objective associated with an optimal solution is defined as

$$z^*_{\text{sim}} = \sum_{(i,j) \in A_{\text{sim}}} p_{i,j} x^*_{i,j}, \tag{4.5}$$

Each of the above metrics compare aggregate values between solutions. For any two solutions to identical problem instances, though, more detailed comparisons could be made. The individual flow values, and values derived therefrom, can be compared directly using well-known normative measure. One example is the root mean square (RMS) of constituents of the objective function, shown in Equation 4.6.

$$RMS_{\text{z}} = \sqrt{\frac{1}{N} \sum_i c_i^2 (x_{i,1} - x_{i,2})^2} \tag{4.6}$$

While such an analysis is common in the realm of nuclear engineering, it is less appropriate for comparing solutions to optimization problems. An instance of an optimization problem may be *degenerate*, i.e., it may have multiple equivalent optimal solutions. Consider a two-arc system with arc costs of unity and a total flow constraint of unity. A spectrum equivalent solutions exist between $(0, 1)$ and $(1, 0)$. When comparing any solution, the difference in objective function value is, necessarily, zero. However, using an RMS analysis, the solutions of $(0, 1)$ and $(1, 0)$ would report the largest possible error. Therefore, only metrics that involve aggregate measures of solutions are used in the following analysis.

4.2 Scalability Campaign

Given the base parameter values described in Tables 4.1 and 4.2, exchanges were generated by scaling the number of reactors in the system. The smallest system modeled included five reactors. The largest exchanges included five-hundred reactors, a value chosen because there are approximately five-hundred reactors currently operating (437) or under construction (71) in the world [5]. Therefore, the largest exchanges modeled represent a time step in a simulation in which the world-wide fleet of reactors are all supplying or consuming a batch of fuel.

Front and back-end exchanges are explored similarly in the scalability campaign. For all 18 combinations of fundamental parameters and each solver, a set of reference cases are established, where the only varying parameter is the number of reactors. Both the solution time and objective value metrics are compared as the problem size increases. Additionally, the effect of convergence criteria for the Cbc solver is investigated.

Individual figures for each experiment are provided for each fuel cycle modeled (f_{fc}) and each solver. Each figure summarizes the results for all combinations of f_{rx} and f_{loc}. The layout for each six-pane figure is shown below.

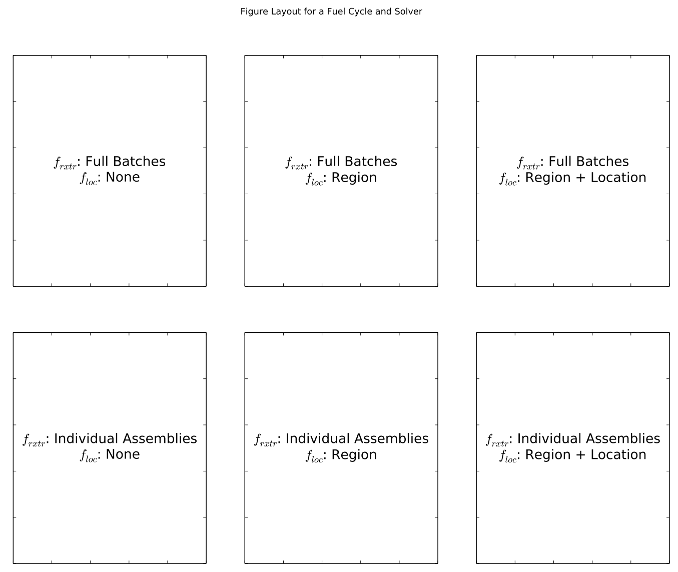
Figure Layout for a Fuel Cycle and Solver
f_{rxtr}: Full Batches
f_{loc}: None
f_{rxtr}: Full Batches
f_{loc}: Region
f_{rxtr}: Full Batches
f_{loc}: Region + Location
f_{rxtr}: Individual Assemblies
f_{loc}: None
f_{rxtr}: Individual Assemblies
f_{loc}: Region
f_{rxtr}: Individual Assemblies
f_{loc}: Region + Location

Figure 4.2: The general figure layout displaying results for different fundamental parameter values.

4.2.1 Front-End Exchanges

4.2.1.1 Reference Case

Reference cases were generated for front-end exchanges by scaling the number of reactors in each exchange. A step size of 5 reactors was used for the range of $[5, 100]$ and a step size of 25 was used from $(100, 500]$. Both the number of variables and number of constraints in a problem are measures of problem scaling. In the NFCTP, constraints are provided by trading entities, and the number of variables is equal to the number of arcs in a given exchange graph. Accordingly, understanding how each quantity scales with the number of reactors is important.

Figure 4.3 shows how the number of arcs scale with problem size for the MOX fuel cycle, and Figure 4.4 shows the same results for the number of constraints. The number of constraints scales linearly, for it is a purely function of the number of entities in an exchange. However, the number of arcs scales by $\mathcal{O}(n^2)$. During exchange generation, the number of suppliers is a function of the number of reactors. Further, each reactor and each supplier have an arc connecting them if the reactor can consume the supplier's commodity. When the addition of a reactor also causes the addition of a support facility, based on parameter vector values, arcs are added for the new reactor and for every reactor previously existing in the system. Therefore, there is an an $\mathcal{O}(n^2)$ relationship between reactors and arcs. Both relationships hold true regardless of the fuel cycle being modeled, and, as can be seen, are also independent of other fundamental parameters. The arc population magnitude, however, is a function of f_{rxtr}. For an f_{rxtr} of 1, i.e., reactors order individual assemblies, the number of arcs per reactor is $\mathcal{O}(n_a)$. When reactors order full batches, the number of arcs per reactor is $\mathcal{O}(1)$.

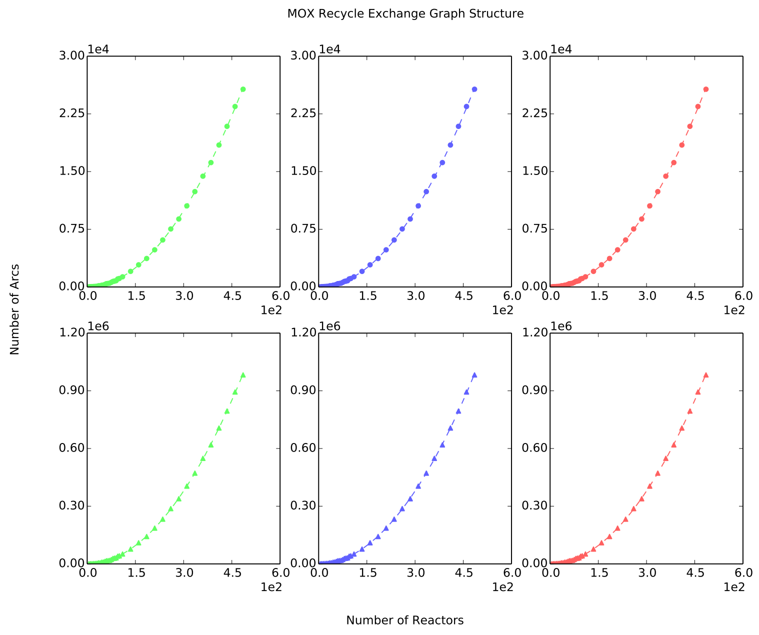

Figure 4.3: Arc population scaling with the number of reactors with corresponding quadratic fits.

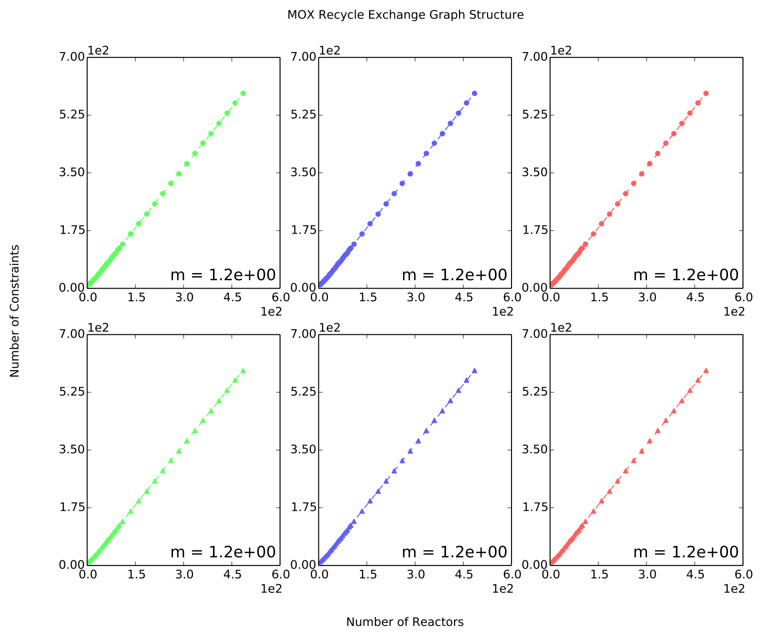

Figure 4.4: Constraint population scaling with the number of reactors with corresponding linear fits.

Greedy Solver

Figures 4.5 to 4.7 show the Greedy Solver results as the number of arcs increases for the OT, MOX, and ThOX fuel cycles, respectively. Plotted with each data set is a linear fit with associated slope value. As discussed in section 4.1, one expects linear-like scaling, which is observed in practice. This scaling behavior is consistent across all fundamental parameters. Of note, however, is that the scaling constant does increase when moving from low-fidelity reactor models to higher-fidelity models.

Clp Solver

As discussed in section 4.1, LP solution scaling is more naturally observed as a function of the number of constraints. Figures 4.8 to 4.10 show the Clp solution times as the number of constraints increases. As can be seen, approximate $\mathcal{O}(n^2)$ scaling is observed. Like the Greedy solver, this scaling is independent of any fundamental parameters. Low-fidelity reactor problems solve very quickly (under a second). Larger, high-fidelity reactor problems can take much longer to solve, i.e., tens of seconds. It does not appear that solution time magnitudes are affected by the choice of location parameter.

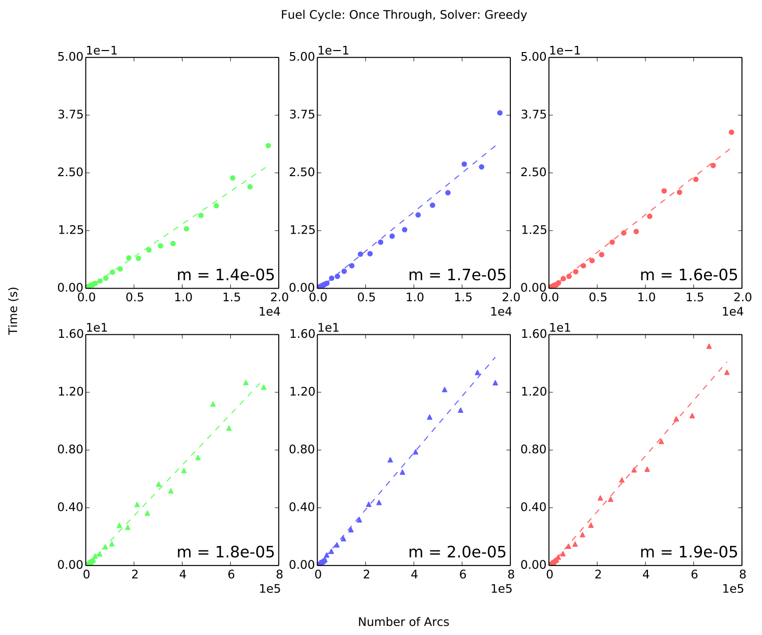

Figure 4.5: Greedy Solver results for the OT fuel cycle as the number of arcs increases.

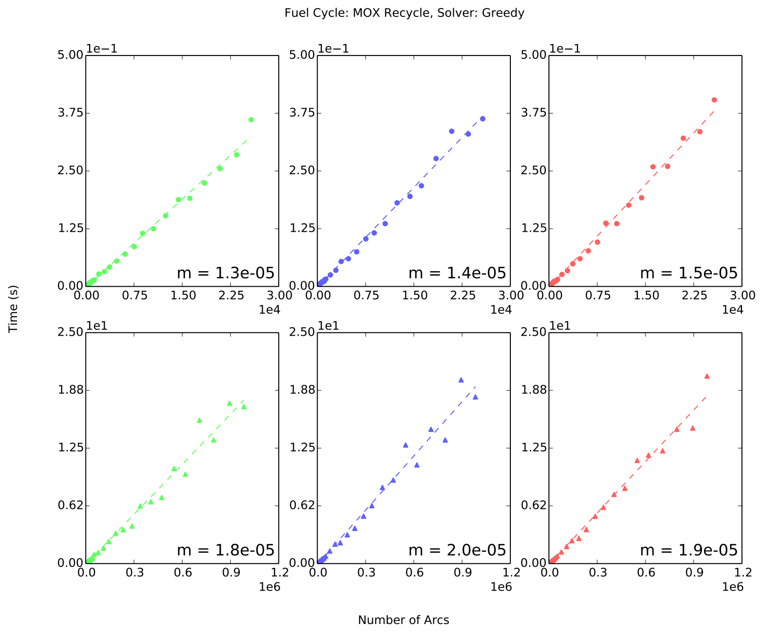

Figure 4.6: Greedy Solver results for the MOX fuel cycle as the number of arcs increases.

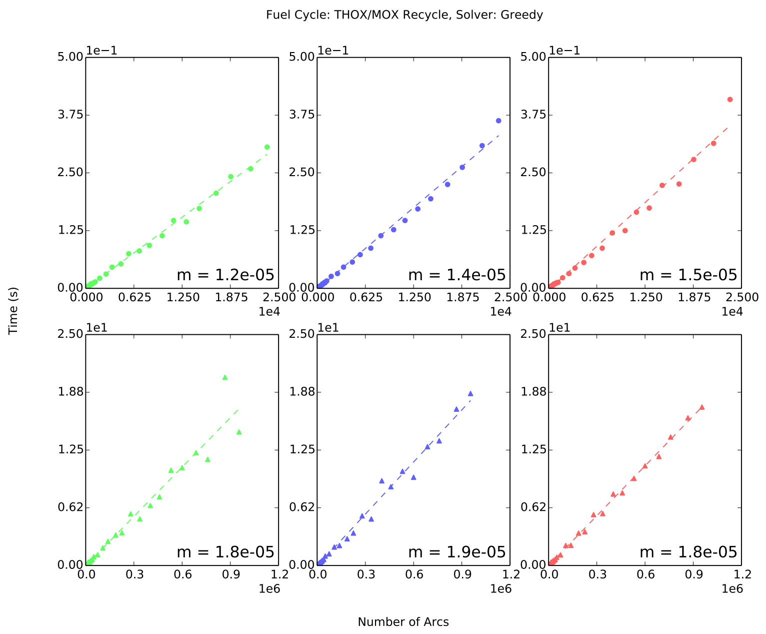

Figure 4.7: Greedy Solver results for the ThOX fuel cycle as the number of arcs increases.

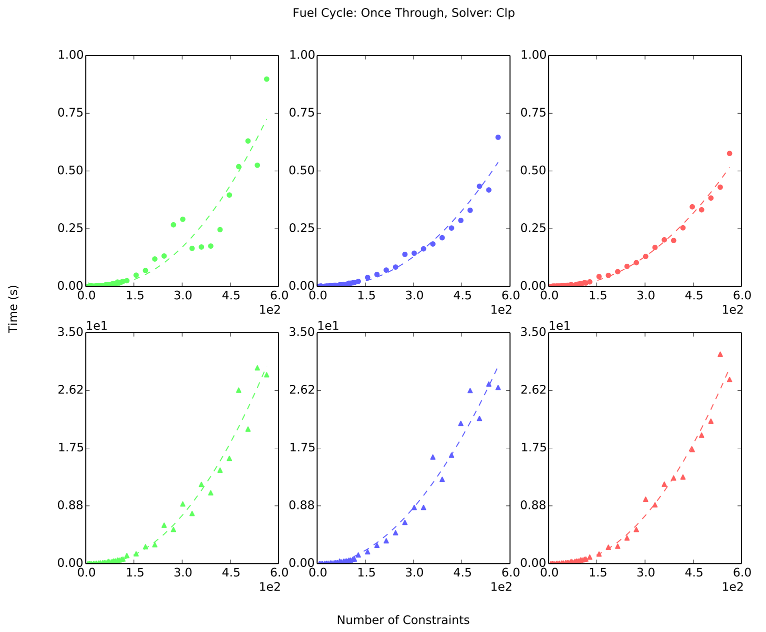

Figure 4.8: Clp Solver results for the OT fuel cycle as the number of constraints increases.

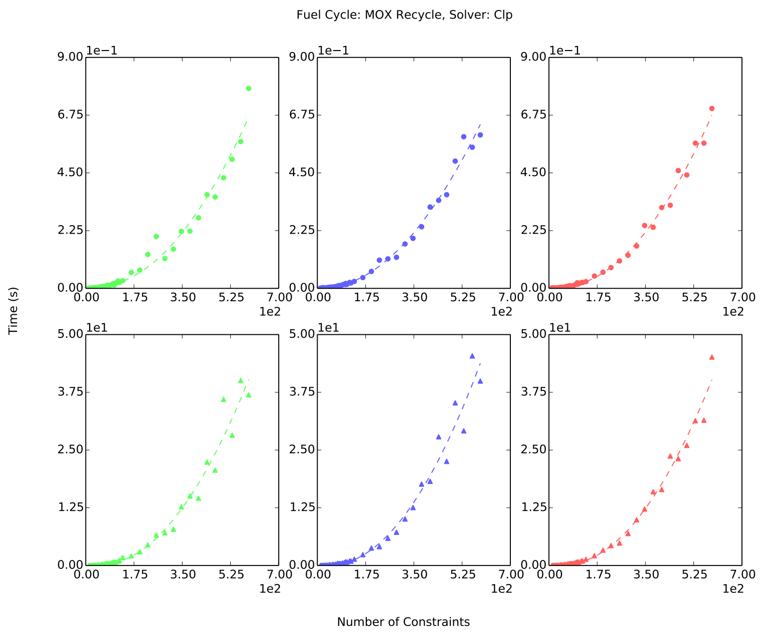

Figure 4.9: Clp Solver results for the MOX fuel cycle as the number of constraints increases.

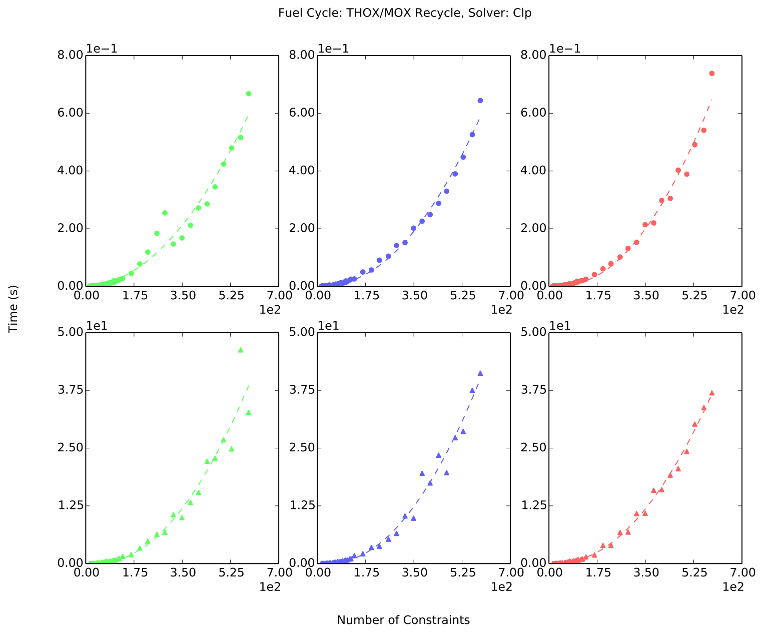

Figure 4.10: Clp Solver results for the ThOX fuel cycle as the number of constraints increases.

Cbc Solver

The performance of the Cbc solver is much more sporadic than either the Clp or Greedy solvers. This is to be expected, for MILP optimization problems are NP-Hard. Further, they are solved using enumeration techniques that do not perform characteristically well, as the Simplex method does.

Cbc was limited to 3 hours for each problem, and a 1% ratio-gap convergence criteria was applied. The term *ratio gap* is in reference to the current known upper and lower bound on the optimal objective value. During the branch-and-bound algorithm, such bounds are maintained and updated between each solution iteration (explained in more detail in Appendix B). The solver reports an optimal solution when the criteria shown in Equation 4.7 is true.

$$\frac{z_U - z_L}{z_U} \leq 0.01 \tag{4.7}$$

In each of the figures below, only instances reaching convergence are displayed in order to attempt to ascertain any related trends. Figures 4.11 to 4.13 show timing results as a function of the number of reactors in the system. Figures 4.14 to 4.16 show timing results as a function of the number of arcs in the system. Note that in each case below, a log-linear graph is used. In each frame, the percentage of converged instances is provided.

Immediately obvious, and slightly counter intuitive, is that the population of converged instances is larger for assembly-based exchanges rather than batch-based exchanges, even though the number of variables in the problem is much lower for batch-based exchanges. Additionally, the Cbc solver converged in many fewer instances for low reactor fidelity in the OT fuel cycle than either MOX or ThOX cycles. Low-fidelity once-through cases have the least amount of “choice” in the system. There is a single commodity, consumer type, and supplier type. Regardless of fuel cycle, reactor fidelity, or objective coefficient strategy, the Cbc solver experiences exponential scaling with problem size.

4.2.1.2 Solution Comparison

Solutions between any two solvers can be compared either in the formulation layer or in the exchange layer. Comparison in the formulation layer is achieved by comparing objective function values (Equation 4.3), whereas comparison in the exchange layer is achieved by comparing a measure the flows and preferences for a given solution (Equation 4.5).

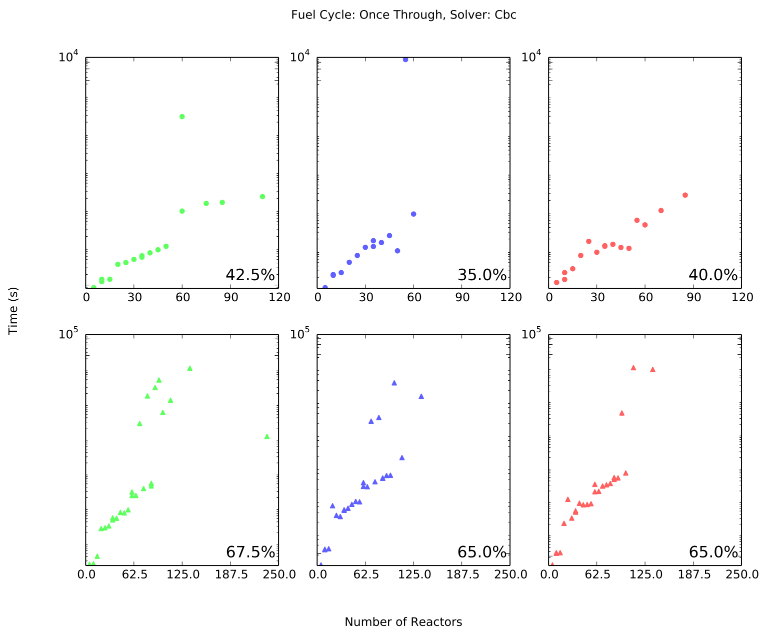

Figure 4.11: Cbc Solver results for the OT fuel cycle as the number of reactors increases.

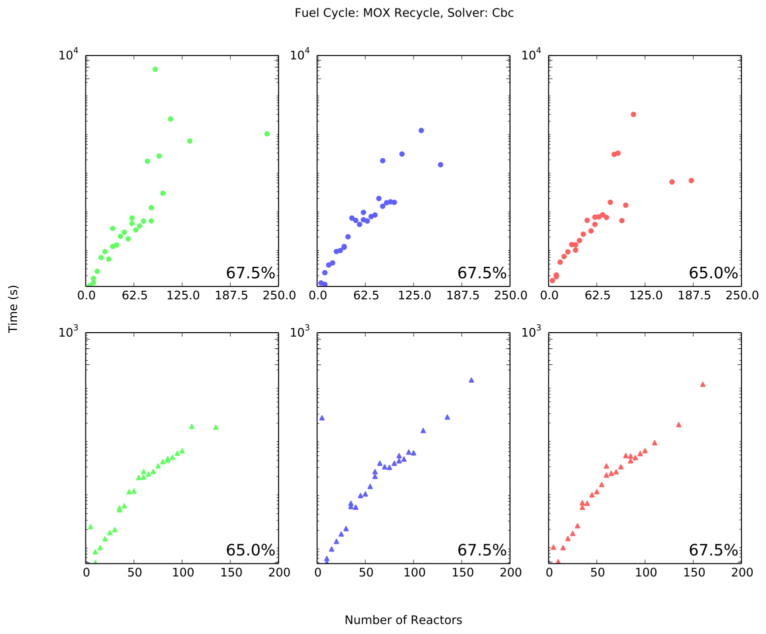

Figure 4.12: Cbc Solver results for the MOX fuel cycle as the number of reactors increases.

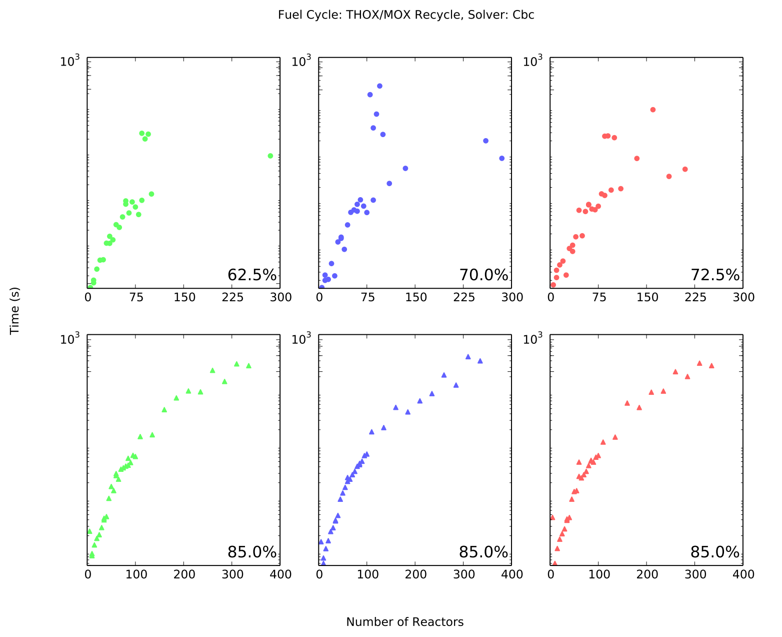

Figure 4.13: Cbc Solver results for the ThOX fuel cycle as the number of reactors increases.

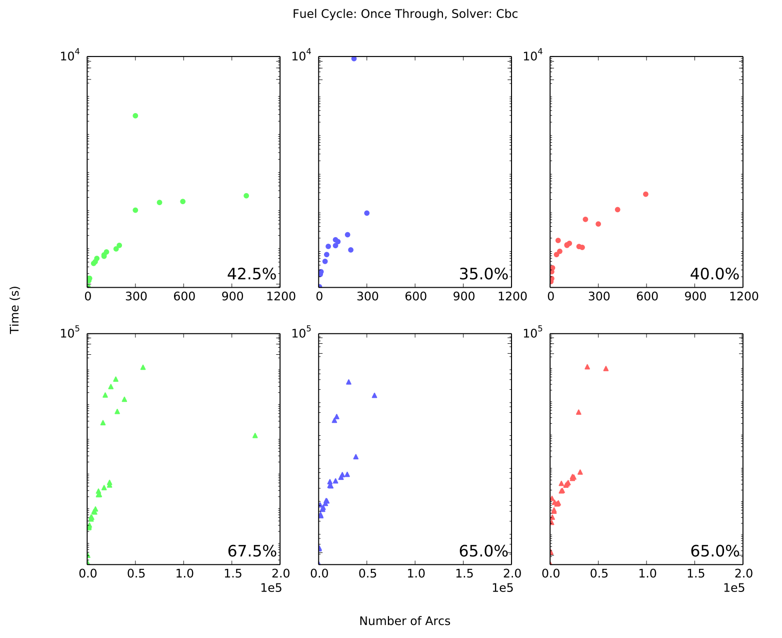

Figure 4.14: Cbc Solver results for the OT fuel cycle as the number of arcs increases.

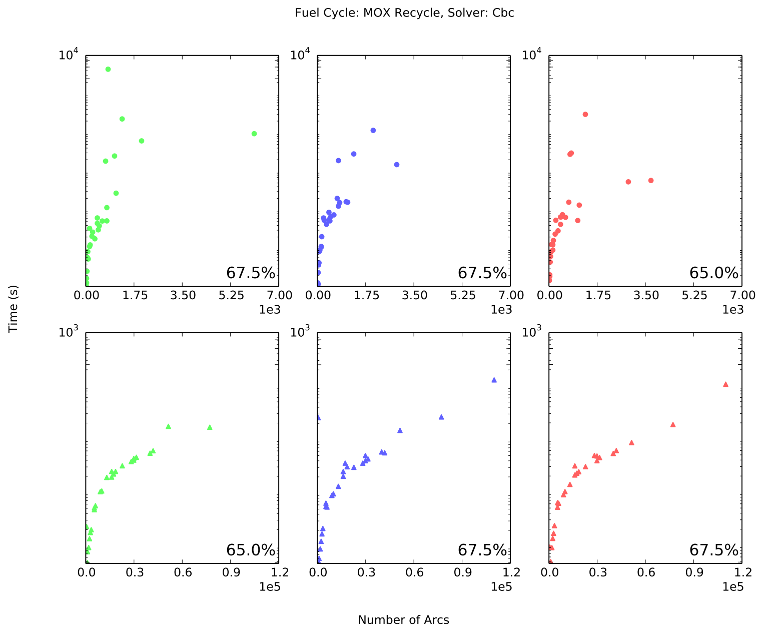

Figure 4.15: Cbc Solver results for the MOX fuel cycle as the number of arcs increases.

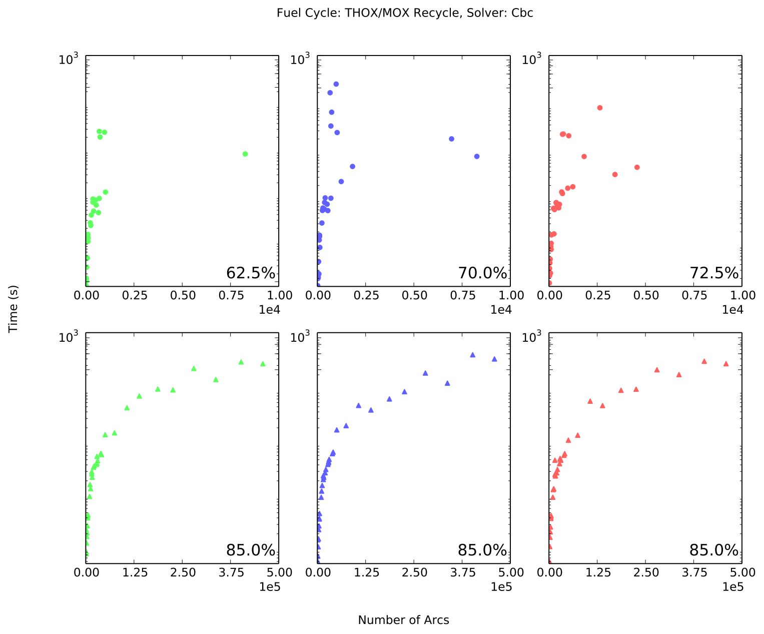

Figure 4.16: Cbc Solver results for the ThOX fuel cycle as the number of arcs increases.

The objective function includes the costs and flows on false arcs which inflates the objective function value. While the false arcs are necessary for guaranteeing a feasible solution, their resulting flows are not taken into account when the DRE back-translates from the formulation to the exchange layer. Accordingly, z^*_{sim} is used as the primary comparison metric. While $z^* \leq z$ is true in cost space, $z^*_{\text{sim}} \geq z_{\text{sim}}$ is *not* necessarily true in practice for MILP instances. Any MILP solver must use some convergence criteria, which takes into account flow values along false arcs.

Comparisons are made between the Greedy solver and the Cbc solver, provided the Cbc solver converged. A converged Cbc solution is guaranteed optimal within the provided tolerance, and is therefore considered to be z^* with a corresponding set of flows X^*. Because solutions increase in magnitude with increasing problem size, a relative comparison is made, as shown in Equation 4.8. The resulting features are similar across fuel cycles. Accordingly, an example for the MOX fuel cycle is shown in Figure 4.17.

$$\frac{z^*_{\text{sim}} - z_{\text{sim,Greedy}}}{z^*_{\text{sim}}} \tag{4.8}$$

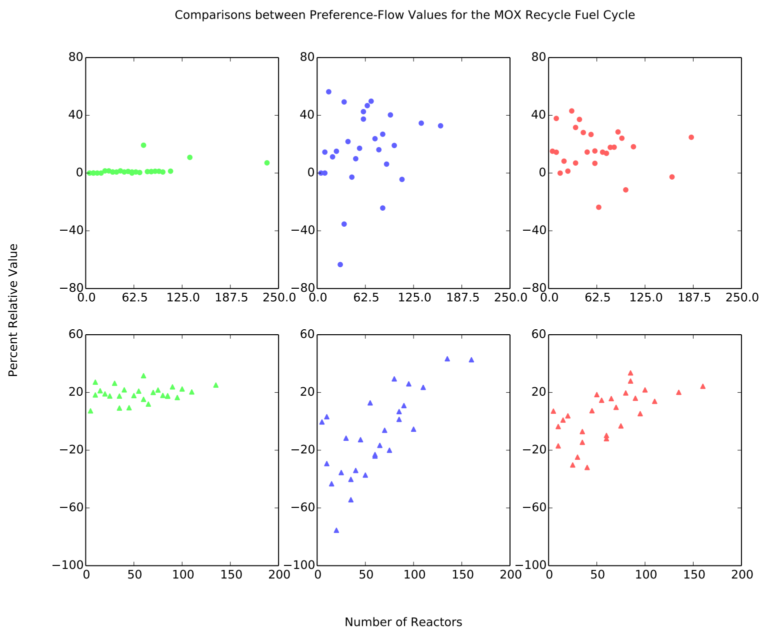

Figure 4.17: Comparisons between relative simulation metrics between the Cbc solver and the Greedy solver for MOX fuel cycles. Only converged Cbc solutions are compared.

Comparing the results in which there are no location-based preferences (green), there is a clear correlation between the number of variables and the relative benefit of using Cbc. The Greedy heuristic performs better when the number of possible assignments is small. This is not unexpected; as the number of decision variables increases, making optimal decisions should, in theory, result in a increasingly better outcomes than making heuristic-based decisions.

Two features of interest arise when comparing the cases in which there is a coarse location preference (blue) and a fine location preference (red). First, some values are negative, implying that the Greedy solver provides a better preference-space solution than the Cbc solver. Importantly, this is true only in preference space; the Cbc always performs better in cost space, by definition. The Greedy solver is allowed to provide better answers in preference space for two reasons: the problem is highly constrained, and false arcs have an arbitrarily high unit cost. Cbc converges when the criteria in Equation 4.7 is met. When a problem is highly constrained, many false arcs will be activated, contributing a large amount to the objective function. If the choice between two possible flows is sufficiently small, i.e., small relative to z^*, then either solution may be returned upon convergence depending on the branch-and-bound search path. Thus, good solutions in preference-space are somewhat lost in the "noise" of cost-space.

Second, this effect is reduced when the objective choice in cost-space increase, as shown in case of fine location preference. In other words, the relative benefit of using a heuristic over a full Cbc solve in preference space for the problems run above appears to be a function of the size of *possible* objective coefficient values. Furthermore, the effect of objective coefficient population size decreases as problem size increases. However, as can be seen from Figure 4.17, this benefit requires a relatively large, high-fidelity reactor simulation, i.e., more than $\sim$80 reactors, to be consistently observed.

This behavior is more pronounced as the number of possible connections increases. Consider the ThOX fuel-cycle results shown in Figure 4.18. Again, large variations are observed for instances with some location preference with small reactor populations, especially for high-fidelity reactor instances. However, as the reactor populations increase, Cbc solutions appear to asymptotically approach relative values close to the base line set by simulations in which there are no location-based preferences.

The Greedy solver can provide quite good preference-space results relative to the Cbc solver when an exchange is highly constrained *and* when the cost coefficient assigned to false arcs is relatively large. This effect can be observed by adjusting the false-arc cost coefficient, e.g., as shown in Equation 4.9. Two exchanges for which the Greedy solver performed better in preference-space were chosen to demonstrate the effect. The results are shown in Table 4.3. Note that it every case, z is smaller for the Greedy solver

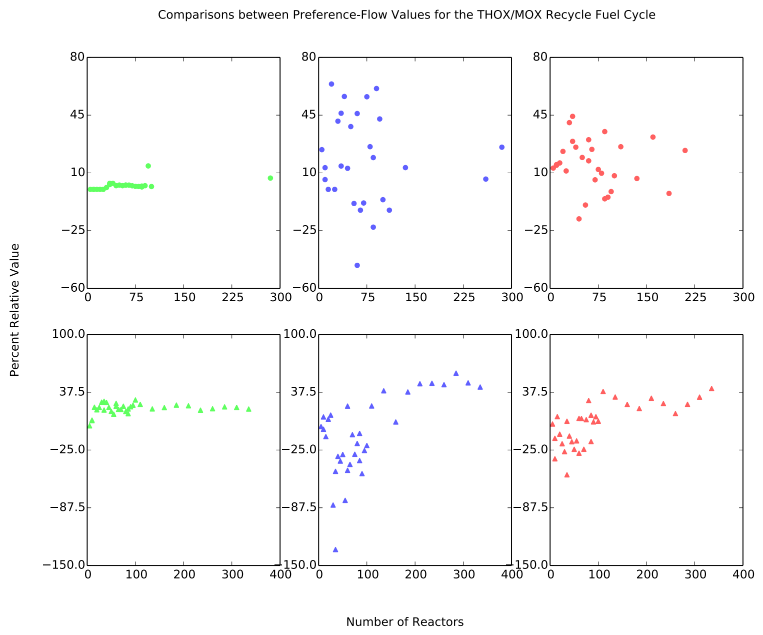

Figure 4.18: Comparisons between relative simulation metrics between the Cbc solver and the Greedy solver for ThOX fuel cycles. Only converged Cbc solutions are compared.

than Cbc; however, z_{sim} for the Greedy solver is larger than the same value with a large Cbc false-arc cost and smaller than values related to small Cbc false-arc costs.

$$c_{\text{false}} = \frac{1}{p_{\text{max}}} + 1 \tag{4.9}$$

Table 4.3: Results from Reducing False-Arc Cost Coefficients.

Simulation ID	**Greedy**		**Cbc, Large Cost**		**Cbc, Small Cost**	
	z (large/small)	z_{sim}	z	z_{sim}	z	z_{sim}
54a5a92ce1ad43e9a713abf114b58a06	5.2e8/1.9e6	1.41e5	5.0e8	1.38e5	1.8e6	1.98e5
938d808a4bd84346b54f38fcb4992386	3.97e8/1.40e6	1.08e5	3.81e8	8.8e4	1.38e6	1.12e5

4.2.1.3 Convergence Criteria

The Cbc solver is highly tuneable. As with many iterative solution techniques, the most critical tuneable criteria affecting the balance between solution quality and solution time is the convergence criteria. It is not clear to what degree solution quality will matter for users of Cyclus. Accordingly, a short exploratory experiment was conducted to examine to what degree convergence criteria affects solution time.

Cbc uses either an absolute or relative upper and lower-bound gap tolerance as possible convergence criteria. All results discussed use the relative gap, termed *ratio gap* in Cbc parlance, as shown in Equation 4.7. For each of the 18 combinations of fundamental parameters, 10 instances of exchanges were executed, spanning a reactor population range of 10 to 500. Figure 4.19 displays the results for runs with reactors trading full batches for ratio gap values of 0.1, 1, and 10%. Figure 4.20 displays the results for runs with reactors trading individual assemblies for ratio gap values of 1 and 10%.

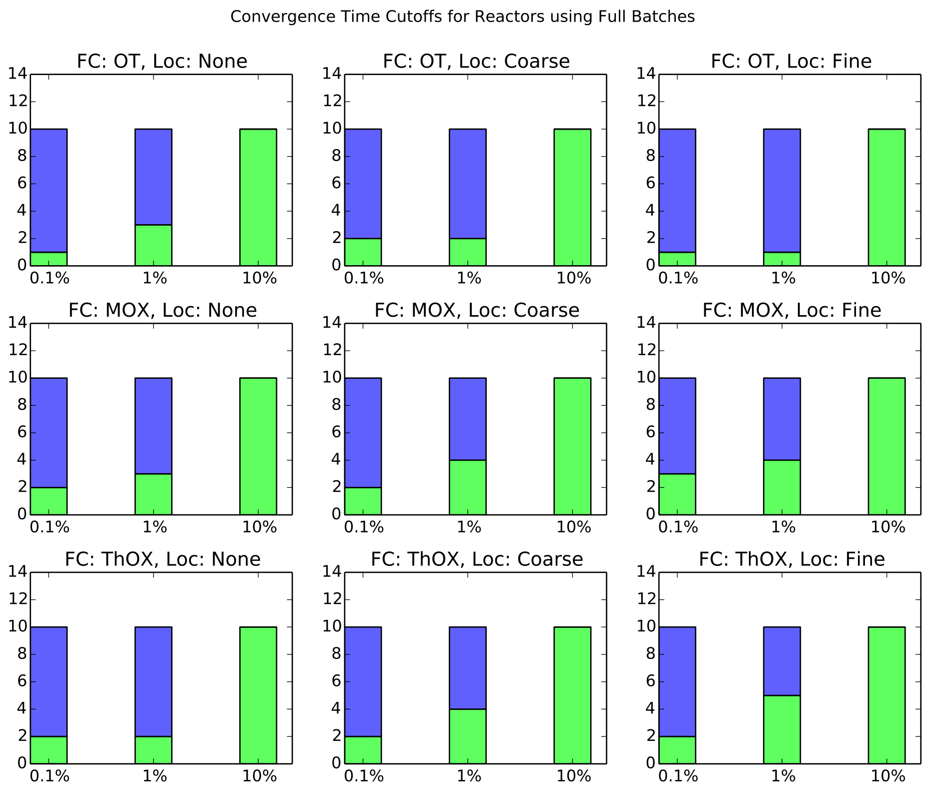

Figure 4.19: Effects of increasing convergence criteria on Front-End exchanges with reactors exchanging batches. Each bar is divided into how many instances converged (green) and did not converge (blue).

Increasing the convergence criteria for smaller problems, i.e., those with reactors requesting a single batch of fuel, has a greater effect than increasing the convergence criteria for larger problems. It is somewhat surprising that the increase from a 1% relative bound gap to a 10% gap allows full convergence

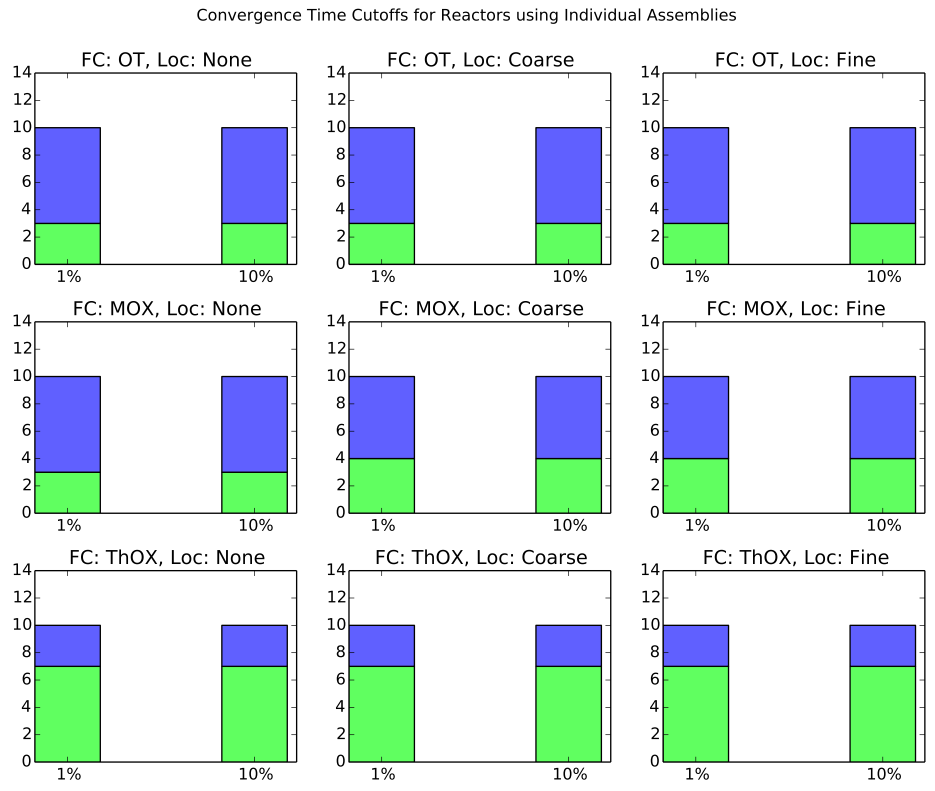

Figure 4.20: Effects of increasing convergence criteria on Front-End exchanges with reactors exchanging assemblies. Each bar is divided into how many instances converged (green) and did not converge (blue).

in the smaller case and has no effect in the larger case. Considering the discussion in section 4.2.2.2, it is likely that the large convergence effect is due increasing the "noise" effect of actual arcs. However, some speed ups in solution times are expected when relaxing convergence criteria, and those speed ups will likely be more profound in smaller-sized problems than larger problems based on this analysis.

4.2.2 Back-End Exchanges

Many of the results of the back-end exchanges mirror those of the front-end exchanges. Therefore, this section will discuss only differences between the two cases. The fact that so many similarities exist is somewhat striking, because from a simulation perspective, back-end exchanges are quite different than front-end exchanges. First, a single request is made for each commodity type that can be consumed by support facilities. Therefore, many fewer requests exist in the system. Next, the supply of used fuel is known. A new supporting facility type, repositories, are also added, resulting in a slightly higher total arc population per reactor. As with the front-end exchanges, the arc population scales as $\mathcal{O}(n^2)$, as can be seen in Figure 4.21, and the constraint population scales as $\mathcal{O}(n)$.

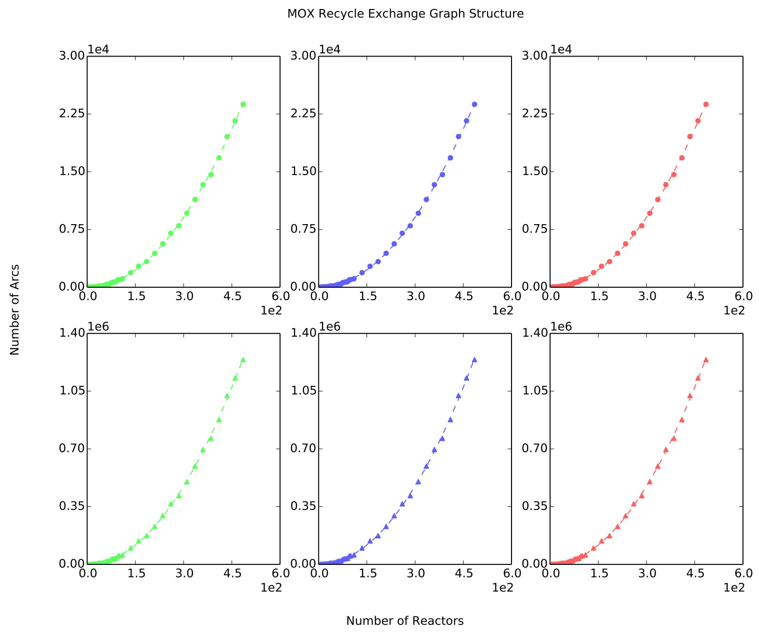

Figure 4.21: Arc population scaling with the number of reactors with corresponding linear fits.

4.2.2.1 Reference Case

Greedy Solver

The Greedy solver performs quite similarly to the front-end case. Its performance is again linear-like in the number of arcs. An example of the MOX fuel cycle is shown in Figure 4.22. As can be seen, the linear coefficient in back-end cases is approximately twice the coefficient of front-end cases. This observation holds irrespective of fuel cycle.

Clp Solver

Back-end exchanges solved with Clp were found to behave similarly for each fuel cycle modeled. The results for MOX-based exchanges is shown in 4.23 with quadratic fits. A key distinction is observed between

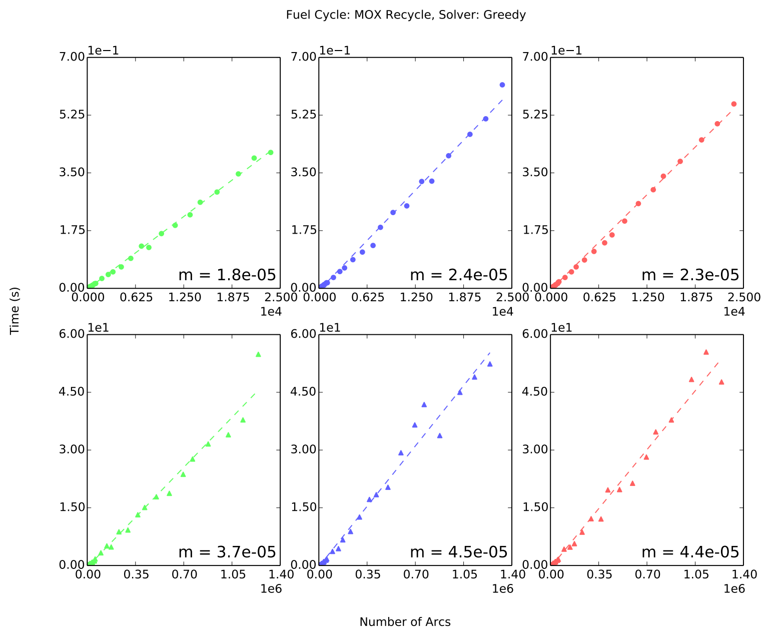

Figure 4.22: Greedy Solver results for the MOX fuel cycle as the number of arcs increases.

back-end and front-end exchanges: while front-end exchanges follow tight $\mathcal{O}(n^2)$ scaling in all cases, back-end exchanges with higher-reactor fidelity clearly have a larger spread in this trend. Furthermore, significantly higher run times are observed for back-end exchanges, with a maximum run time of ∼40 seconds for front-end exchanges and ∼150-300 seconds for back-end exchanges.

Cbc Solver

Cbc behavior is also similar to the front-end case. Importantly, exponential scaling with problem size is again apparent, as can be seen in Figure 4.24. The primary difference between the two exchange types is the increased population of converged solutions for high-fidelity reactor instances. Whereas reactor fidelity was not a large factor with respect to convergence probability in front-end exchanges, it appears to be a large factor for back-end exchanges.

4.2.2.2 Solution Comparison

Two notable features can be seen when comparing Greedy and Cbc solutions in Figure 4.25. The first is that Cbc almost always performs better than Greedy in preference space. Further, $z^*_{\text{sim}} \leq z_{\text{sim,Greedy}}$ is true

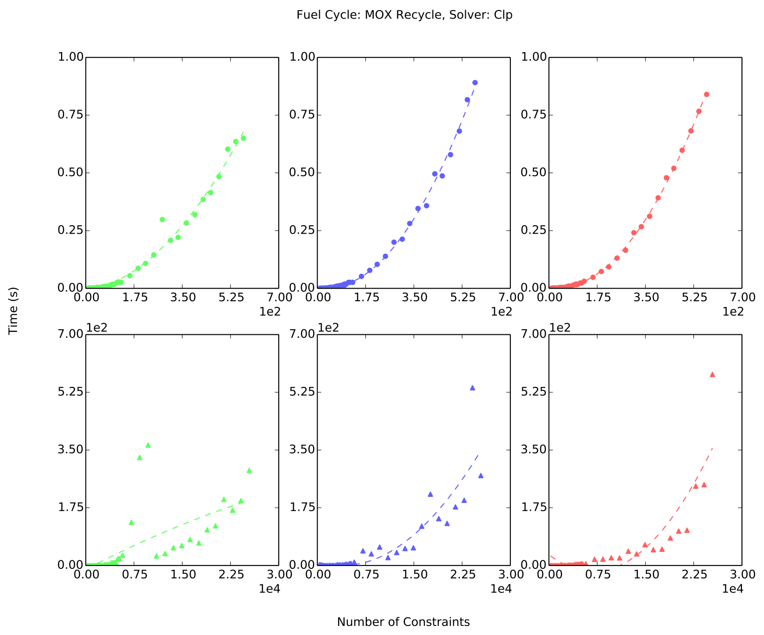

Figure 4.23: Clp Solver results for the MOX fuel cycle as the number of constraints increases.

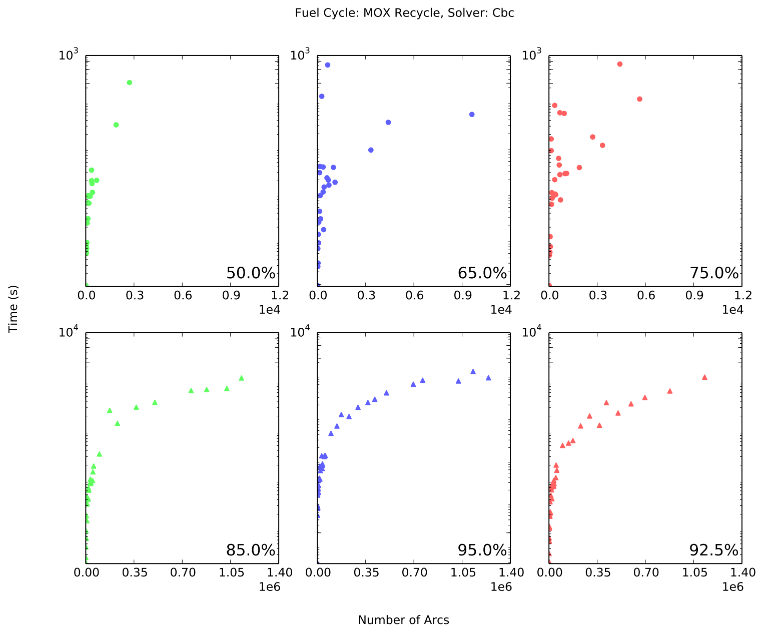

Figure 4.24: Cbc Solver results for the MOX fuel cycle as the number of arcs increases.

only for very small exchanges. Additionally, Cbc preference space results relative to the Greedy solver are, in general, better for low-fidelity reactor exchanges than high-fidelity exchanges.

Secondly, the simulation-objective gain from using Cbc appears to be problem-size independent when there are a large number of variables in back-end exchanges. As can be seen high-fidelity reactor results in Figure 4.25, when the number of reactors is large, a relative gain in simulation objective of 30%-40% is realized by using Cbc. The final value is a function of the degree to which location-based preferences are used.

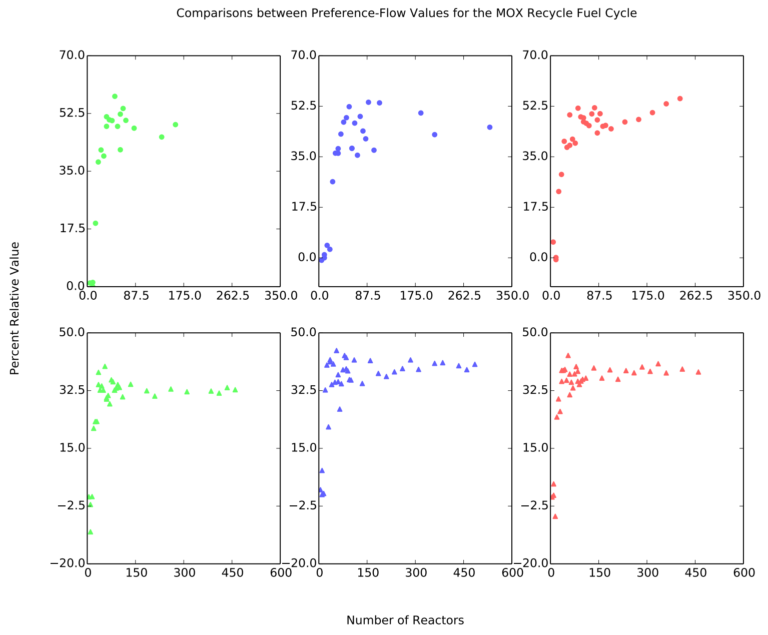

Figure 4.25: Comparisons between relative simulation metrics between the Cbc solver and the Greedy solver for MOX fuel cycles. Only converged Cbc solutions are compared.

4.3 Stochastic Campaign

The generation of exchanges is designed as a stochastic process. Each reactor is assigned a target enrichment per consumable commodity in a given range, and each facility is assigned a location value. Enrichment stochasticity results in perturbed constraint matrix coefficients, and location stochasticity results in perturbed objective coefficient if f_{loc} is set to include location-based preference.

In section 4.2, a single exchange instance was investigated for each solver as problem size scale was investigated. While clear patterns were inferred as a result of the study, stochastic effects were left unexplored. Accordingly, a second study was undertaken in order to examine how sensitive both solution values and execution time are to randomness in instance generation.

For this study, instances are generated for a single parameter vector. It is best to compare converged Cbc solutions for confidence in the resulting objective value and in order to run all desired instances in a reasonable amount of time. Accordingly, the total reactor population was set at 65 based on the scaling study results. Specifically, this is approximately the largest reactor population for which all configurations of parameters converged (see Figure 4.11). For each combination of fundamental parameters, 1000 instances were generated and executed by each solver for both front and back-end exchanges.

In both sections, cumulative observations are presented. Figures are presented in two panes for a given fuel cycle, with low-fidelity reactor instances on the left and high-fidelity reactor instances on the right. In each pane, the results for all three location fidelity parameters are shown.

4.3.1 Front-End Exchanges

4.3.1.1 Fundamental Parameter Variation

The Greedy Solver and Clp both proved to have relatively stable solution times. The timing results are similar across fuel cycles. Accordingly, MOX fuel cycle results are shown for the Greedy Solver in Figure 4.26, and Clp results are shown in Figure 4.27. Apart from solution time stability, the primary observation for both the Clp and Greedy solvers is related to location fidelity effects. Although there is a ranking in average solution time by location parameter *for a given collection of fundamental parameters*, that ranking changes for each collection of parameters, as can be seen in the presented figures.

The Cbc solver showed much less stability in solution times. Further, behavior was found to be different for each fuel cycle. Figures 4.28 to 4.30 show the timing results for the OT, MOX, and ThOX fuel cycles, respectively.

Given the variety in stability by fuel cycle, viewing the underlying timing distributions can also provide insight. Accordingly, Figure 4.31 shows all timing distributions for low-fidelity reactor models, and Figure 4.32 displays the corresponding results for high-fidelity models.

A number of interesting features exist in the figures related to the Cbc timing study. Significantly, reactors in batch-mode have poor convergence for once-through cycles, regardless of location parameter.

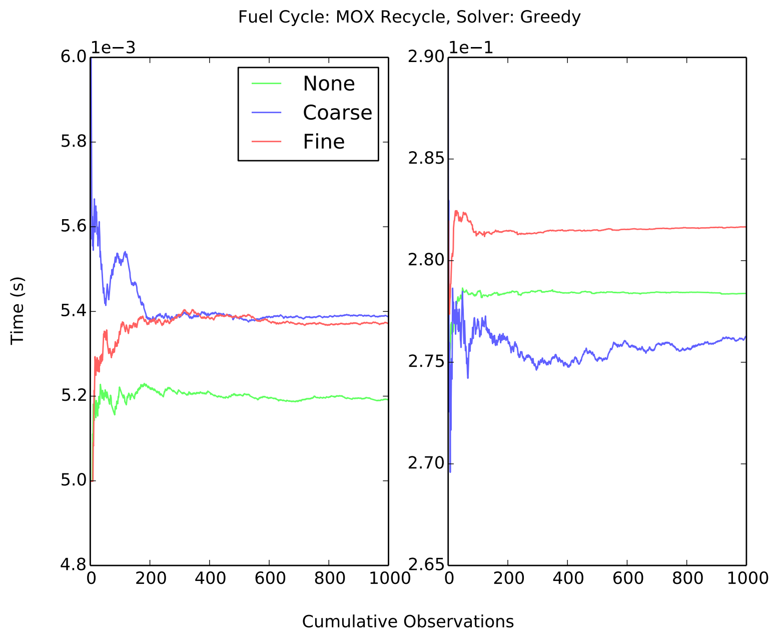

Figure 4.26: Cumulative average solution time of the Greedy solver for the front-end MOX fuel cycle. Low-fidelity reactor instances comprise the left pane, and high-fidelity reactor instances comprise the right pane. Each colored line represents a different objective coefficient location fidelity.

This echoes observations in the scaling study. For the remaining fuel cycles, outliers consistently are observed for low-fidelity reactor models, also regardless of location parameter. This effect is seen in the cumulative observation figures most easily. Each occurrence of a timing outlier causes a large jump in the observation. The converged, non-outlier instances with location-based preferences show single-mode structure, whereas those without location preferences appear to be bi-modal.

Conversely, high-fidelity reactor instances were found to almost always converge. Outliers were found in the once-through fuel cycle results, but not in other fuel cycles. Again, this behavior results in choppy cumulative observations. Both the MOX and ThOX fuel cycles were shown to be comparatively stable. Further, both cases have clear tri-modal solution-time populations.

4.3.1.2 Simulation Objective vs. Solution Time

The trade off of choosing to use a full-fledged optimization solver over a heuristic is one between solution fidelity and solution time. Figure 4.33 shows a comparison between the Cbc and Greedy solvers. For each instance, relative simulation-objective values were computed as shown in Equation 4.8. Similarly,

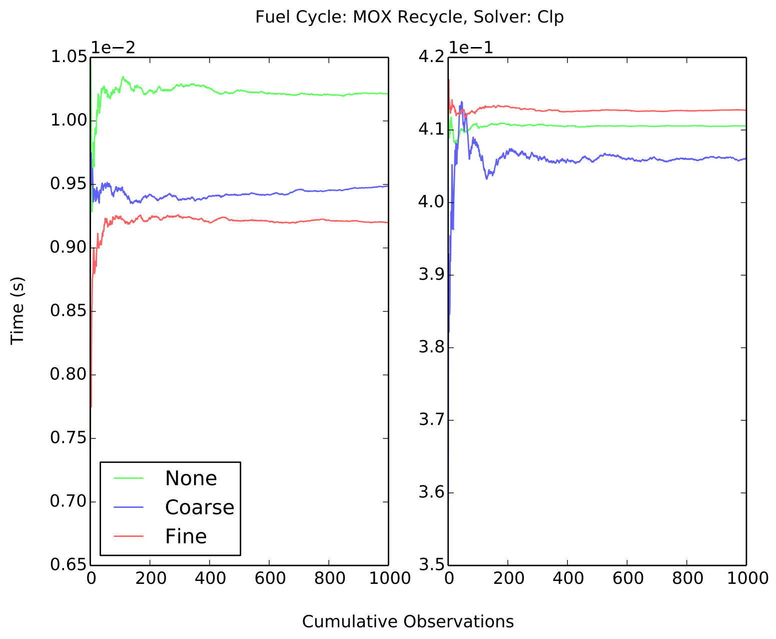

Figure 4.27: Cumulative average solution time of the Clp solver for the front-end MOX fuel cycle.

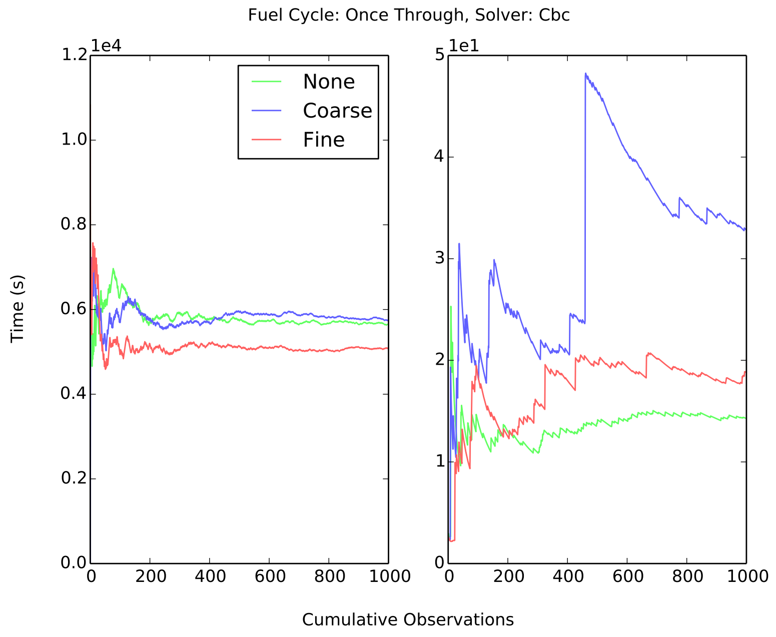

Figure 4.28: Cumulative average solution time of the Cbc solver for the front-end once-through fuel cycle.

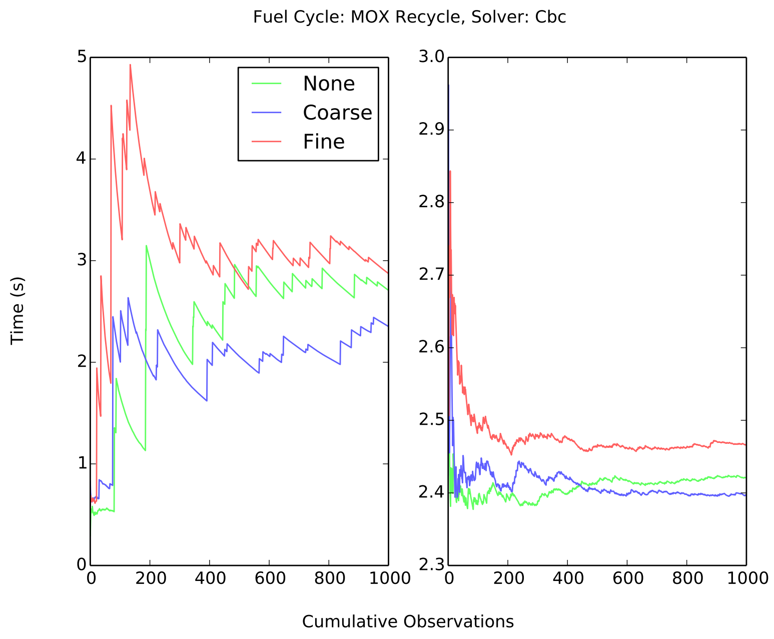

Figure 4.29: Cumulative average solution time of the Cbc solver for the front-end MOX fuel cycle.

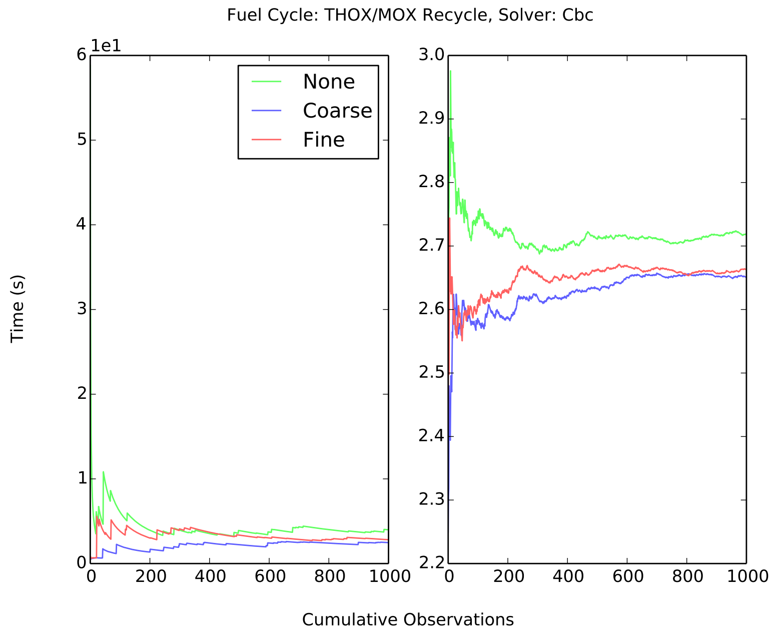

Figure 4.30: Cumulative average solution time of the Cbc solver for the front-end ThOX fuel cycle.

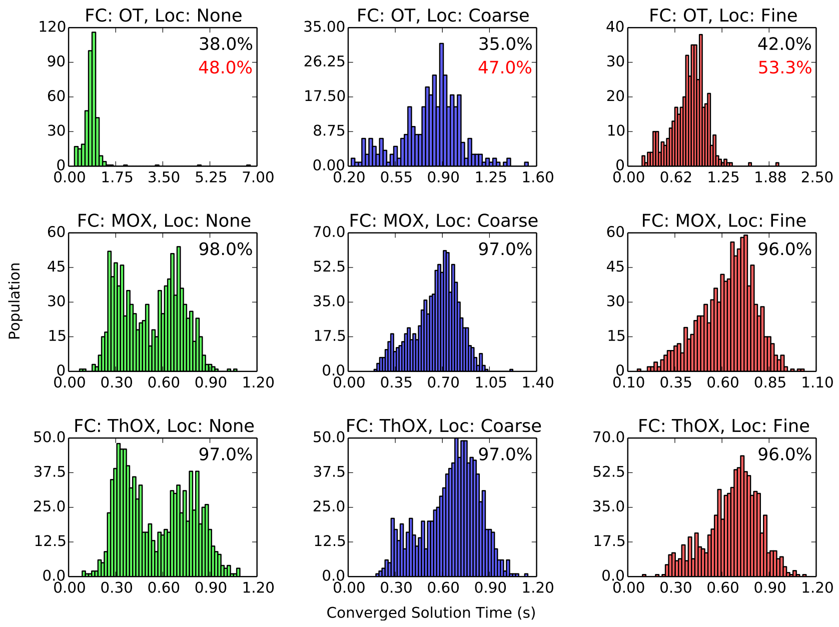

Figure 4.31: The distribution of converged solution times for all low-fidelity reactor instances. Fuel cycle fidelity increases from top to bottom, and location fidelity increases from right to left. Outliers have been filtered out in order to show distribution shape. The instance population percentage, i.e., the percentage not defined as outliers, is shown is provided in black, if relevant. The percentages of all converged instances is shown in red, if relevant.

relative solution times were computed. The average result found for high-fidelity reactor instances solved with Cbc was taken as a reference point, plotted at the origin in Figure 4.33. Values are plotted for each combination of fundamental parameters. A data point's position along the x-axis indicates the average deviation from the high-fidelity model's simulation objective, i.e., the dot product of preference and system flow. The y-axis position indicates the average relative time difference.

In most every case, the results are as expected. The high-fidelity, Cbc-based solution provides a better simulation objective for a more expensive time. In general, the Greedy always performs much faster than Cbc, as expected from a heuristic. Except for the Once-through case, all Cbc solves require similar times, and the highest fidelity simulation provides the answer with the highest objective measure. However, as was seen in the scalability study, the Cbc does not always provide a better simulation-based metric when large costs are concerned. Interestingly, the coarse-location fidelity Greedy solver results provided a better average simulation objective.

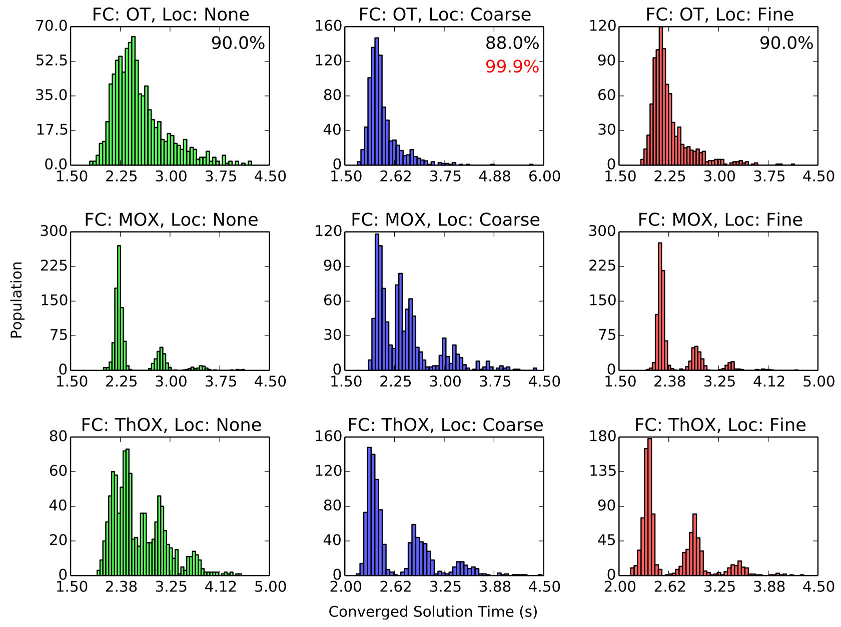

Figure 4.32: The distribution of converged solution times for all high-fidelity reactor instances. Fuel cycle fidelity increases from top to bottom, and location fidelity increases from right to left. Percentages are identical to Figure 4.31.

4.3.1.3 False Arc Cost Effects

As was shown for previously, providing a small pseudo cost results in a higher simulation-based objective metric. For example, Figure 4.33 displayed results in which the Greedy solver provided a higher average simulation-based metric than the Cbc solver. Figure 4.34 displays the underlying data with additional data for small-cost Cbc solves. As can be seen, in the lower center pane, the average preference-flow metric increases in a Cbc-high-cost, Greedy, Cbc-low-cost order.

4.3.2 Back-End Exchanges

As with the scalability study, back-end exchanges perform quite similarly to front-end exchanges. Accordingly, basic results are reviewed, and results that lead to recommendations are discussed in more detail.

Both the Greedy and Clp solvers again proved to have relatively stable solution times. An example of the Greedy solver observed solution times for the MOX fuel cycle is shown in Figure 4.35. Clp results are

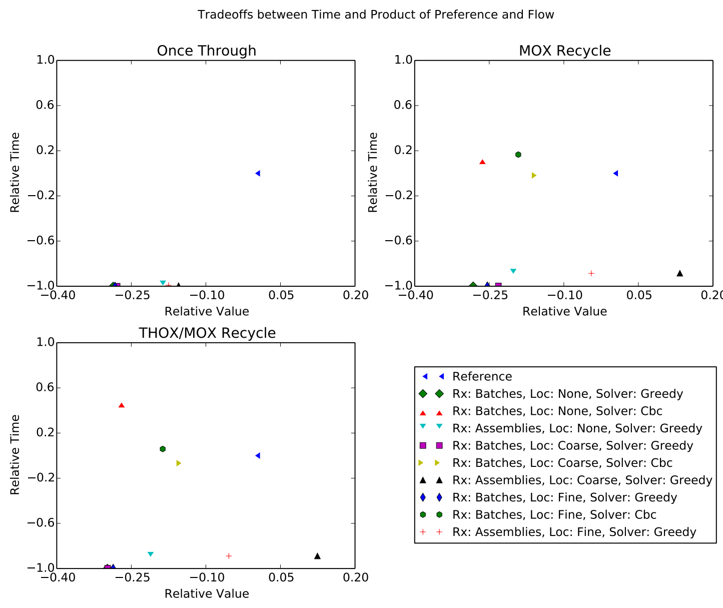

Figure 4.33: A comparison of simulation-objective values and solution times between instances solved with Greedy and Cbc solvers. Reference values are comprised of high-fidelity reactor instances solved with Cbc. Each other combination of fundamental parameter and solvers are then compared against the reference. Note that the once-through fuel cycle pane does not include other Cbc solvers, because their solution times were very long.

shown in Figure 4.36.

The Clp solution time required for high-fidelity reactor models with no location preference was found to be significantly higher than for instances which included a location preference. A similar trend was found for Cbc solutions, as shown in Figures 4.37 and 4.38 for the MOX and ThOX fuel cycles, respectively. The timing discrepancy exists in low-fidelity reactor models in the Cbc case, and is also much more pronounced. These results indicate that solution times, in certain instances, can be significantly reduced if solvers can better differentiate between arcs based on objective coefficients. Therefore, a possible speed-up strategy can include "salting" the preference vector by adding a random δ_p to each entry. Such a strategy will likely be useful in only certain cases, and should be tested only if long run times are encountered.

4.4 Summary

The performance and output of the Greedy heuristic, Clp solver, and Cbc solver was tested on a large number of exchange graph instances. Linear-like, i.e., $\mathcal{O}(n \log \frac{n}{N})$, problem-size scaling was confirmed for

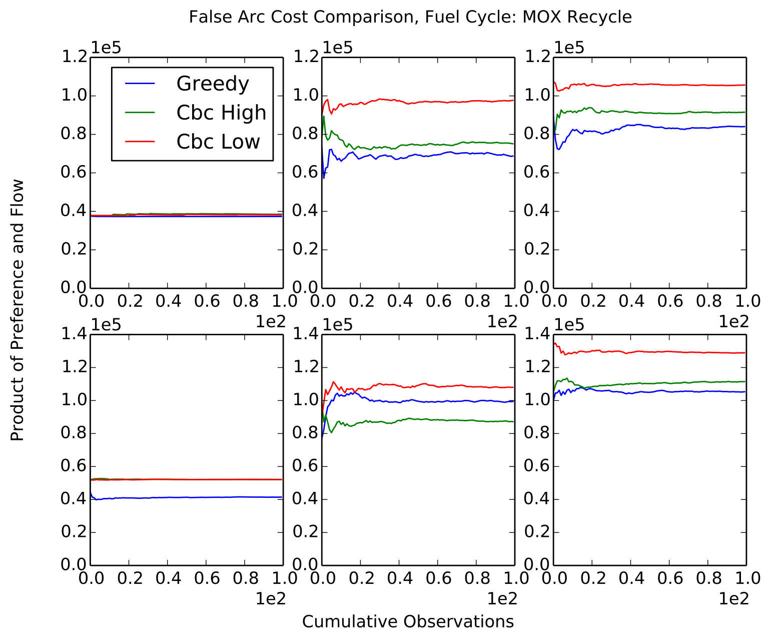

Figure 4.34: Simulation-objective values for 100 instances solved with the Greedy solver, a Cbc solver with a high false arc cost, and the Cbc solver with a low false arc cost.

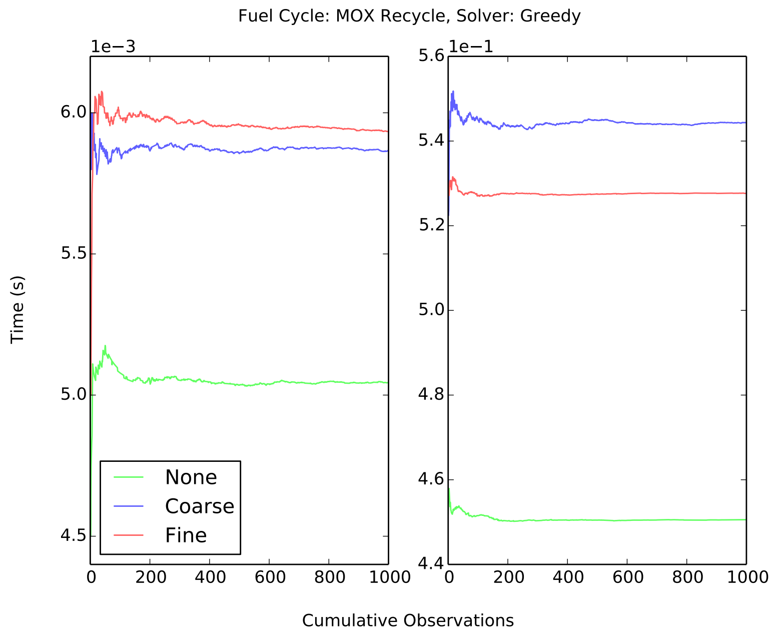

Figure 4.35: Cumulative average solution time of the Greedy solver for the back-end MOX fuel cycle.

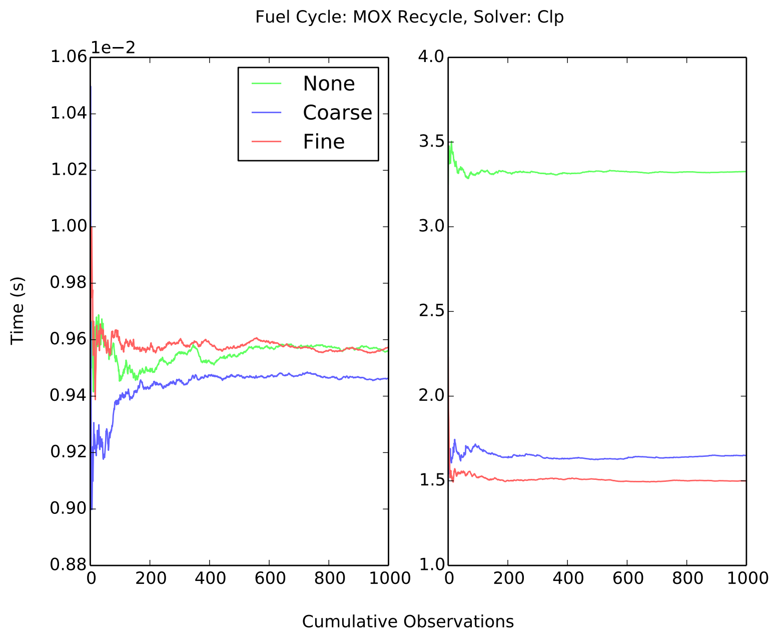

Figure 4.36: Cumulative average solution time of the Clp solver for the back-end MOX fuel cycle.

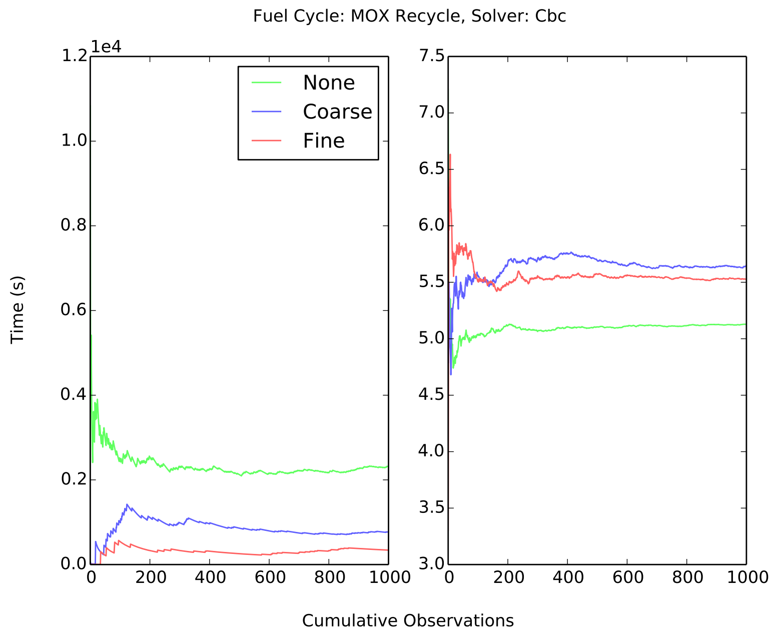

Figure 4.37: Cumulative average solution time of the Cbc solver for the back-end MOX fuel cycle.

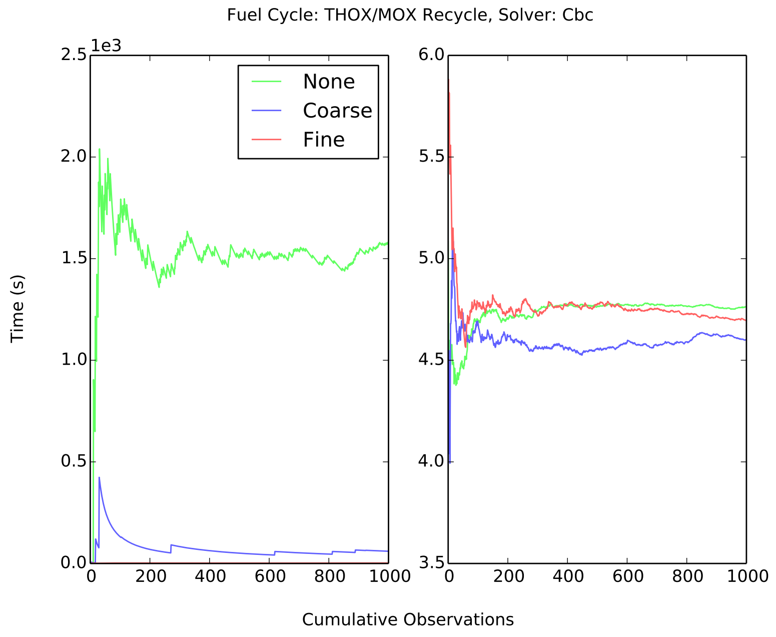

Figure 4.38: Cumulative average solution time of the Cbc solver for the back-end ThOX fuel cycle.

the Greedy heuristic, $\mathcal{O}(n^2)$-like scaling was observed for most Clp-solved instances, and exponential scaling was observed for instances solved with Cbc. The Greedy heuristic performs quite well even for realistic, large NFCTP instances, solving in tens of seconds on the testing system. Cbc was found to be largely sporadic. Many instances solved quite quickly; however, outliers were observed in almost all experiments that greatly increased average solution time.

The translation between exchange graph to NFCTP-instance was found be quite sensitive. Both the false arc cost and the cost translation function were found to have a large effect on preference-based metrics. Some Cbc solutions with a high false arc cost were observed to be worse than Greedy solutions in preference space. This effect was largely mitigated by reducing false arc cost.

This work has shown that resource exchange instances can be solved reliably and optimally in Cyclus with open-source MILP solvers, even if those solutions may take a relatively long time. Many Cyclus users will likely find heuristics to be acceptable for their work, especially for analyses requiring a large number of fuel cycle simulations. Users will be empowered, however, to solve a reference simulation optimally, for example, and compare results with simulations using a heuristic. The user can then either continue to solve resource exchanges optimally with different options, such as running time restrictions or loosened convergence criteria, or change to a heuristic. This capability represents a novel step forward in simply modeling nuclear fuel cycles as well as analyzing dynamic effects within a given fuel cycle.

5 Summary

This body of work sought to enhance the state of the art in dynamic fuel cycle simulation. Prior to this effort, most decision making related to a given simulation was made by a human analyst prior to simulation execution. Further, even *ex situ*, human-based, decision making was limited to collections of entities and macroscopic descriptions of commodities. Both of these effects resulted simulation platforms lacking either physics fidelity, entity relation fidelity, or both.

The Cyclus dynamic fuel cycle simulator and this work are inextricably tied. While many of the concepts and methods described herein may be applied to any implementation of a non-trivial supply-demand model, the development of this work was spurred by the need for such methods implemented in Cyclus .

Upon its inception, Cyclus had a variety of goals. An analyst's ability to choose the level of physical, social, and economic fidelity was of chief concern. This behavior is supported in Cyclus through a plug-in framework of various archetypes. Thus, an analyst could use a high-fidelity reactor archetype or a low-fidelity archetype, for instance. An equally important concept of Cyclus was the ability model a variety of fuel cycles with similar archetypes. Therefore, the Cyclus kernel was required to abstract away fuel-cycle specific behavior. Finally, the developers of Cyclus also desired to model regional interaction mechanisms, such as tariffs or other international trade instruments. In short, it was required to solve the *general case* of nuclear fuel cycle simulation.

5.1 Statement of Work

The goal of this work was, chiefly, to design, implement, and analyze a highly-flexible, physics and economics-informed simulation engine. The engine was split into two primary conceptual categories: entity deployment and entity interaction. Developing a sophisticated entity interaction mechanism was the chief focus of the majority of the presented *oeuvre*.

5.1.1 Modeling & Simulation

Significant design constraints were placed on the design of a entity interaction mechanism. First, it must support arbitrary physics and chemical constraints, as well as general supply-chain constraints, such as inventory and processing constraints. Further, it must model the competition of resources among entities for which demand and supply of resources may be fungible. Finally, arbitrary social phenomenon must be able to be translated to the interaction framework. The resulting interaction mechanism, termed the Dynamic Resource Exchange (DRE), was informed chiefly from the fields of supply-chain management, agent-based modeling, and mathematical programming.

The DRE allows agents to inform both system supply and demand of resources through a request-bid framework. Physics fidelity is provided to agents in this framework by utilizing fully specified `Resource` objects. For example, nuclear fuel demand can be specified directly by an ideal isotopic vector in a `Material` object. Once supply and demand is known, social interaction models can be applied to affect resource flow-driving mechanisms. For example, a tariff can be modeled by uniformly reducing preferences of transactions between agents outside of a given `Region`. Presently, a cardinal preference model is used as the flow-driving mechanism.

The DRE is comprised of three layers: a resource layers, with which agents interact, an exchange layer, and a formulation layer. Supply, demand, and preferences are defined in the resource layer, for a specific type of `Resource` object. The exchange layer provides a general resource exchange representation, irrespective of a specific object type. The representation is comprised of a bipartite graph of supply and demand nodes, supply and demand constraints, and a measure of preference for each proposed connection between nodes. The DRE can be solved either in the exchange layer or by translating the exchange into a minimum-cost, network-flow problem, resulting in the formulation layer. Translation to LPs and MILPs are both supported, where MILPs are required if entities require individual, quantized resources. Such a case arises when one would like to model individual reactor assemblies.

5.1.2 Experimentation

After the DRE framework was designed and implemented, it was tested and analyzed. The full Cyclus simulator is still nascent with respect to full-featured archetypes that would utilize DRE-specific features.

Therefore, instances of resource exchanges were required to be generated. The generation of instances is based on a parameter vector, comprised of fundamental and instance parameters. Fundamental parameters include which fuel cycle is being modeled and whether reactors request single batches of fuel or a collection of individual assemblies of fuel. Instance parameters include the number of reactors being modeled and the number of support facilities being modeled, among many others. Exchanges representing the front-end and back-end of the nuclear fuel cycle were generated separately.

Generated exchanges were solved with the Greedy Heuristic in the exchange layer and the COIN-OR Clp and Cbc solvers in the formulation layer. A number of key observations were ascertained by both scaling and stochastic experimental campaigns. First, the Greedy solver exhibited $\mathcal{O}(n \log \frac{n}{N})$ (linear-like) scaling with problem size (i.e., the number of variables) for all configurations of exchanges, matching theory. Somewhat surprisingly, the Clp solver was also shown to have efficient scaling behavior as well. The theoretical complexity of the general case of simplex method is not known, although it is known to be "efficient in practice" [29]. Further, solving relaxations of LPs is a fundamental step in branch-and-bound algorithms for MILPs (e.g., those used in Cbc). Therefore, exploring the behavior of given problem structures as LPs is of interest. In this work, Clp solves of front-end exchanges were found to have $\mathcal{O}(n^2)$ scaling in the number of constraints, regardless of fundamental parameter configuration. Back-end exchanges also experienced $\mathcal{O}(n^2)$ behavior for smaller problem sizes. At larger problem sizes, especially for instances with high variability in objective coefficient distribution, the quadratic scaling trend tended to break down.

Solving optimization problems is known to be NP-Hard, and solution times are known to scale exponentially in the general case. This behavior was observed for the Cbc solver in both front and back-end exchanges. A solution time limit of three hours was placed on Cbc calculations, and many cases converged for both exchange types in the scaling campaign. Threshold problem sizes were observed in each exchange type and fundamental parameter configuration. Maximum convergence probability was found in high-fidelity reactor instances for non-once-through fuel cycles. Low-fidelity reactor, once-through fuel cycle instances were found to have particularly poor convergence probability.

Performing comparative solution analyses between the Greedy and Cbc solvers illuminated the importance of false-arc costs in the MILP formulation of the NFCTP. Specifically, multiple exchange instances were observed to have better performance-space results using the heuristic over a full-fledged optimal solve in cost-space. This behavior was found to be the result of using a large false-arc unit cost in the NFCTP formulation in conjunction with a relative optimal value convergence criteria. A relative-value criteria is required in order to be used in a general simulation framework. Replacing the large cost with a small cost, however, proved to reverse the observation in both the scaling and stochastic campaigns.

Finally, the use of location-based preferences was found to have little or no effect on Cbc solution times in most cases. However, for back-end exchanges of MOX and ThOX fuel cycles, cases in which no location-based preferences were applied were found to have significantly longer solution times than cases in which they were applied. Therefore, applying a small objective perturbation may prove a reasonable speed up strategy for some exchange structures.

5.1.3 Recommendations

Because of the development of the DRE, users may now apply physical, economic, and social models to NFC simulation. The choice of solver will largely depend on the fidelity of the associated models and underlying data. The Greedy solver will always provide a feasible solution to the given exchange instance, applying any physical, chemical, or supply-chain constraints. Therefore, if a user has a low-fidelity economic or social model, then the Greedy solver will likely meet the users needs.

With higher-fidelity economic and social models, obtaining a optimal solution becomes paramount. While an LP, and thus Clp, can be used to model approximations of fuel cycles, to-date user and developer experience has found the modeling of individual fuel assemblies to be conceptually simpler both to use and code. In short, having a binary decision regarding a supplied resource is simpler than managing the acceptance of an arbitrary number of partial resources, especially in a multi-commodity system. Requiring optimal solutions and using quantized resource transfers necessitates using a MILP solver, such as Cbc.

Users can expect exponential solution time scaling when using Cbc. For a relatively small convergence criteria, i.e., 1%, most front-end instances with a reactor population greater than 150 were found not to converge within a 3-hour time limit. Back-end instances showed better convergence behavior with problem size. The Cbc solver was found to perform better with higher reactor fidelity, a promising result for this use case. With a small arc-cost implemented in the kernel by default, users may find greater speed ups by loosening the default convergence criteria. However, tuning the specific convergence criteria and setting a solution-time ceiling will be dependent on the an individual's use case. Finally, after a preference-perturbation option is implemented, users may find significant speed ups through its use.

5.2 Suggested Future Work

Immediate future work involves implementing the solver options and related features that have been used in this work. To date, the Cyclus code base includes only the option to utilize the Greedy solver. The Cbc and Clp solver options have been implemented for this work privately, and should be straightforward to incorporate more broadly. Further, exchange partitioning was a fundamental feature utilized in this work that has not yet been implemented in Cyclus . Finally, the preference "salting" feature and related user-facing interface will be implemented as well.

This work was motivated chiefly by a noted lack of features present in other fuel cycle simulators. The DRE mechanism supports many identified features, namely competition among consumers, constrained supply, and the inclusion of a framework that supports regional (e.g., geographic) effects. In order to make use of the features made available by the DRE, however, appropriate Cyclus archetypes must be implemented. Archetype developers are already implementing models that make use of the ability to model constrained supply for work related to the DOE's Fuel Cycle Options (FCO) campaign. Further, the IAEA is known to support the DESAE simulator [9] in order to use its minimal regional-interaction mechanisms. Implementing the appropriate archetypes to extend this use case is a clear use-case of potential future work that will immediately impact users. Making use of consumer and supplier competition, although interesting, is less straightforward. Adequate background work will be required in order to develop sufficient models that support both economic and nuclear engineering effects.

The DRE implements a one-phase, consumer-based, preference-setting interaction mechanism. Users of the method have already expressed interest in an N-phase capability, in which suppliers could effectively investigate the consumer's preference function with respect to their available supply of resources. Such capability can be provided by encapsulating and exposing the preference-setting interface. However, delineating the interface between manager agents, will require sufficient attention to support all use cases. In general, however, extending the DRE mechanism to support a multi-period bidding procedure is a fruitful area for future work.

The solvers used in the DRE are also potential subjects of future work, based on future use cases and user requirements. As implemented, the Greedy solver sorts exchange entities based on average arc preference. Use cases may find other metrics to be superior given certain economic or social models. Furthermore, the Greedy solver is but one heuristic approach. Other approaches may be implemented and tested for various use cases, as needed. Finally, a minimum-cost formulation was used in the formulation layer of the DRE. Adding a maximum-preference formulation is both possible and relatively straightforward.

The work herein described utilized the both the Cbc and Clp solver via their provided driver APIs. Many more switches are available to both, which may be investigated further should their relative performance found to be lacking. Further, the Cbc solver can be specifically implemented for the Cyclus use case, although this is highly discouraged by the Cbc development community.

Importantly, the Open Solver Interface (OSI) was used when developing the in-code model of the NFCTP. While the Cbc solver was used because of its permissive license. Users with access to proprietary MILP solvers, such as CPLEX, can reuse much of the available framework. Should such a use case be requested, implementing interface extensions should be straightforward.

5.3 Closing Remarks

A novel way to model dynamic, nuclear fuel cycles has been proposed, designed, implemented, and tested. New features include competition between suppliers and consumers, constrained supply and consumption, and the inclusion of extra-facility effects, such as state-level relationships. This work provides a general framework on which nuclear fuel cycles can be modeled. As the Cyclus ecosystem grows, the features implemented herein will be used and tested. It is the sincere desire of the author that fuel cycles with high potential to better the human condition can be identified more easily by sophisticated analysts using Cyclus and the DRE framework more generally. Energy use has long been known to better quality of life. Choosing the best way of developing and deploying the world's energy infrastructure will be the challenge of the current generation and for generations to come.

A Linear Programming and the Simplex Method

Linear programming is a sophisticated technique to explore large, constrained option spaces to find optimal solutions to systems of equations given some objective. The seminal paper on linear programming techniques was provided by Kantorovich in 1940 [38]; however, this occurred during World War II, and was thus kept secret. An efficient solution technique (called the Simplex Method was provided by Dantzig in 1947 and published in 1951 [21], effectively opening the field. The realm of linear programming is well studied in the field of optimization sciences.

A.1 Linear Programming

Linear programs (LPs) have a relatively simple general construction. There are a set of n decision variables forming a vector, x, a cost vector, c, a constraint matrix, A, associated with m constraints and a right-hand-side threshold vector, b. The standard form for linear programs is as follows.

$$\min_{x} \; z = c^\top x \tag{A.1a}$$
$$\text{s.t.} \; Ax \geq b \tag{A.1b}$$
$$x \geq 0 \tag{A.1c}$$

It is important to note that LPs can be formulated in many ways, e.g. as minimization problems, with equality constraints, etc. Most texts cover the standard transformations required to turn a given formulation into the standard form. In general, any LP can be transformed into the standard form.

The $m \times n$ dimensional constraint matrix defines an n-dimensional option space. Any set of values for the vector of decision variables, x, that does not violate a constraint (including those bounding the variables) is termed a *feasible solution*. It is not only possible, but likely that a given problem formation has many feasible solutions, in which case any feasible solution that optimally satisfies the objective function (i.e., provides a global minimum in the case of the standard form) is termed an *optimal solution*. It is also possible, for a given set of constraints, for there to be no feasible solutions and thus no optimal solution. Such a problem formulation is termed *infeasible*. More commonly, the set of constraints forms a *feasible region*. Take for example the following formulation:

$$\max \ 3x_1 + 2x_2 \tag{A.2a}$$
$$\text{s.t.} \ -2x_1 + x_2 \leq 1 \tag{A.2b}$$
$$x_1 + x_2 \leq 5 \tag{A.2c}$$
$$x_1 \in [0, 4] \tag{A.2d}$$
$$x_2 \geq 0 \tag{A.2e}$$

which creates the feasible solution space shown in yellow in Figure A.1.

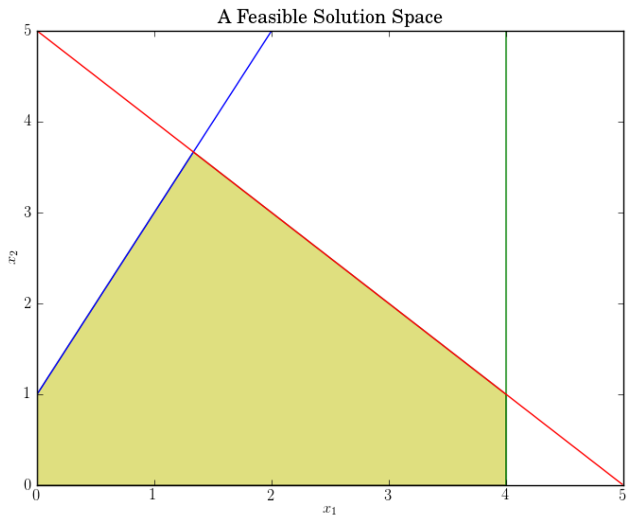

Figure A.1: An example of a feasible solution space.

The program can become infeasible by adjusting a constraint. Take for instance, an increased boundary constraint for x_2.

$$\max \ 3x_1 + 2x_2 \tag{A.3a}$$
$$\text{s.t.} \ -2x_1 + x_2 \leq 1 \tag{A.3b}$$
$$x_1 + x_2 \leq 5 \tag{A.3c}$$
$$x_1 \in [0, 4] \tag{A.3d}$$
$$x_2 \geq 4 \tag{A.3e}$$

This arrangement results in the infeasible linear program shown in Figure A.2, where the updated constraint's effect is shown in red.

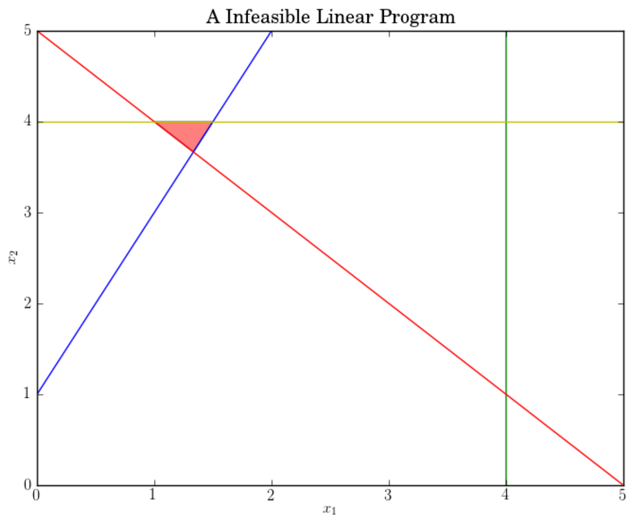

Figure A.2: An example of a infeasible solution space.

The standard form of a linear program shown in Equation A.1 is an example of a *primal* linear program. A distinction is made between a primal linear program and its *dual*. Duality theory is involved and only treated lightly in this review. The standard form of the dual of Equation A.1 is given in Equation A.4.

$$\max_{u} \quad w = b^\top u \tag{A.4a}$$

$$\text{s.t.} \quad A^\top u \leq c \tag{A.4b}$$

$$u \geq 0 \tag{A.4c}$$

A few critical differences exist. First note that the objective directions are switched: if a primal form has a minimization objective, its dual has a maximization objective. The constraint matrix is now $m \times n$-dimensional (it is in fact the original constraint matrix transposed). There is a new series of decision variables that form the corresponding solution space, i.e., the positive vector u, as shown in Equation A.4a. These variables are related to the original right-hand side of the constraint formulation, the vector b. The costs of the original problem, c, now form the right-hand side of the dual's constraint formulation, Equation A.4b.

The concept of duality is critical in the field of mathematical programming because it provides well-defined optimality characteristics of a given program. These are achieved via the *Strong Duality Theorem* and *Weak Duality Theorem*, shown below as stated in [23].

Theorem A.1 (Weak Duality Theorem). *If x is primal feasible and u is dual feasible, then the dual objective function evaluated at u is less than or equal to the primal objective function at x.*

The Weak Duality Theorem provides inextricable linkage between a primal feasible solution and dual feasible solution. If a dual feasible solution is found, it provides a lower bound on the optimal solution. If a primal feasible solution is found, it provides an upper bound on the optimal solution. Both of these criteria, in tandem, help to greatly reduce the required search space during optimization sweeps.

Theorem A.2 (Strong Duality Theorem)**.** *Exactly one of the following three alternatives hold:*

1. *Both primal and dual problems are feasible and consequently both have optimal solutions with equal extrema*
2. *Exactly one of the problems is infeasible and consequently the other problem has and unbounded objective function in the direction of optimization on its feasible region*
3. *Both primal and dual problems are infeasible*

The Strong Duality Theorem provides the backbone for much of linear programming theory and application. It states that not only do feasible solutions to the primal and dual programs provide upper and lower bounds on optimal values, but that, in fact, the optimal values are *equal*. This provides a criterion to *know* when an optimal value is reached.

With this slight overview of the realm of linear programming, one can move on to solution techniques for problems that can be represented as linear programs.

A.2 The Simplex Method

The Simplex Method is a popular algorithm to solve linear programs first published by Dantzig [21]. Conceptually, it is quite intuitive, especially from a geometrical point of view. Before continuing in more detail, an overview of the method is provided via the example from the previous section.

Note that there are five vertices of the polygon (i.e., a polytope more generally) formed by the full set of constraints:

1. $(0, 0)$
2. $(0, 1)$
3. $(\frac{4}{3}, \frac{11}{3})$
4. $(4, 1)$
5. $(4, 0)$

The Simplex Method begins at a vertex, for example $(0, 0)$, and evaluates the objective function.

$$f(0, 0) = 3 * 0 + 2 * 0 = 0 \tag{A.5}$$

Neighbor vertices are then evaluated, in order to determine which provides the larger value (in the case of maximizing objectives).

$$f(0,1) = 3 * 0 + 2 * 1 = 2 \tag{A.6}$$

$$f(4,0) = 3 * 4 + 2 * 0 = 12 \tag{A.7}$$

The vertex $(4, 0)$ provides a larger objective function value, so the algorithm moves to this vertex and determines the next largest neighbor. In this simple example, there is only one choice, and it is trivially larger.

$$f(4,1) = 3 * 4 + 2 * 1 = 14 \tag{A.8}$$

Accordingly the algorithm moves a second time, analyzing the neighboring vertices again.

$$f(\frac{4}{3}, \frac{11}{3}) = 3 * \frac{4}{3} + 2 * \frac{11}{3} = \frac{34}{3} \tag{A.9}$$

At this last move, a terminating condition has been achieved: a vertex has been found for which all of its neighbors provide a lower value for the objective function, i.e.,

$$14 \geq 12 \text{ and } 14 \geq \frac{34}{3}. \tag{A.10}$$

Thus, the optimal value for (x_1, x_2) has been determined to be $(4, 1)$. It is immediately obvious that the simplex algorithm in this state is a hill climbing (or hill descending) algorithm. The chief reason why this is possible (i.e., why one is guaranteed to find a globally optimum solution) is that the objective function and constraints are *convex* functions of the decision variables. Convexification is required to find optimum solutions to both linear and integer programs.

In general, the Simplex Method is efficient, i.e., for most cases solutions are found in less than exponential time in the number of variables. There are certain program structures for which solution times are exponential, however, and it in such cases, more advanced *Interior Point* techniques are required. In general, Interior Point algorithms are often much faster than Simplex Algorithms [23], but are beyond the scope of this review.

To begin a more robust discussion of the Simplex Method, one must introduce the notion of *slack variables*. Slack variables are used to transform inequality constraints into equality constraints, effectively taking the "slack" out of the system. Slack variables are always positive, thus one could use a slack variable, s, to convert

$$\sum_i a_i x_i \leq b \tag{A.11}$$

to

$$\sum_i a_i x_i + s = b \tag{A.12}$$

and

$$\sum_i a_i x_i \geq b \tag{A.13}$$

to

$$\sum_i a_i x_i - s = b. \tag{A.14}$$

The addition of slack variables allows one to rewrite an LP given in the standard form of Equation A.1 as the *canonical form*.

$$\min_{x,s} \; z = c^\top x + 0^\top s \tag{A.15a}$$

$$\text{s.t. } Ax - b = s \tag{A.15b}$$

$$x, s \geq 0 \tag{A.15c}$$

Given that there are N decision variables and M constraints, the cardinality of x is N and the cardinality of s is M. Furthermore, in the literature, the x_i variables are termed *nonbasic* variables whereas the s_i variables are termed *basic* variables.

For any LP in the canonical form, the Simplex Algorithm can be applied to it to determine optimal values for its decision variables, or to determine that it is unbounded or infeasible. The basic structure of the method is outlined below in Algorithm 2.

```
Data: Decision variables, an objective function, and a set of constraints.
Result: Optimal values for the decision variables or a flag denoting infeasibility or unboundedness.
Get initial vertex;
if no vertex is found then
    feasible solution space is empty;
end
while not unbounded and not empty and not done do
    Select column, i, via pricing;
    if no column is found then
        optimal condition found;
        done;
    end
    Select row, j, via the ratio test;
    if no row is found then
        solution space is unbounded;
    end
    Perform a Jordan exchange on element (i, j);
end
```

Algorithm 2: The Simplex Algorithm

There are four core operations associated with the Simplex Algorithm:

1. finding an initial vertex
2. column pricing
3. row selection
4. exchanging elements

If finding an initial vertex is not trivial (e.g., if the origin is not a candidate), then the operation to do so requires use of the Simplex Method on a related LP where the origin is an available candidate. That process will be described last.

The primary concept required to understand the Simplex Method's operations is that of the *basis*. The basis begins as the set of decision variables. The algorithm progresses by moving slack variables into the basis, and it does so efficiently by analyzing the most "valuable" variables to target (i.e., which current basis variable affects the optimal value the most).

Column pricing and row selection are the operations that select the current basis and nonbasis variables to target. The Jordan exchange process translates the formulation into the new basis, exchanging basic and nonbasic variables. This is perhaps more intuitive from a geometrical point of view. Consider some starting vertex with many possible sides along which to move. The process of column pricing and row selection *chooses* the side along which to move, and the Jordan exchange *reorients* the problem. Revisiting the example problem, remember the first step. The vertex $(4, 0)$ was determined to be the best direction in which to move. After a Jordan exchange, the resulting LP would look like Figure A.3.

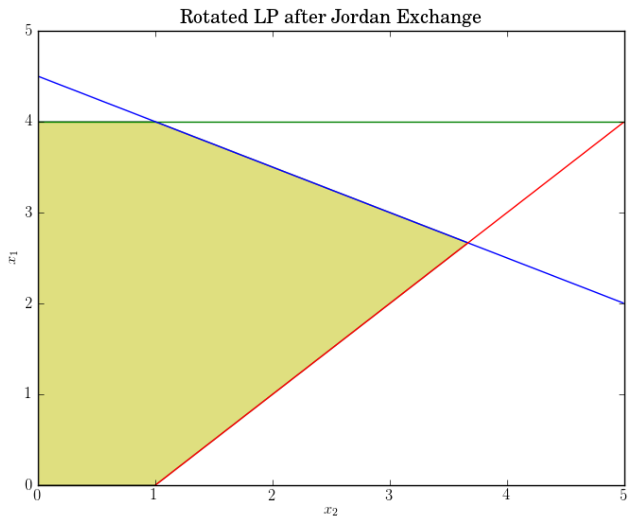

Figure A.3: The Reoriented LP after the first Jordan Exchange.

The column pricing operation selects the slack variable which will enter the basis. The most naive implementation is to select the variable which will have the largest effect on the objective function, i.e., which has the largest magnitude *reduced cost*. For instance, in Equation A.2a, x_1 has a reduced cost of 3, and x_2 has a reduced cost of 2 (note that both costs are in the same positive direction as the objective, i.e., maximization). Accordingly, choosing x_1 as the nonbasic exchange variable is a valid option. However, any algorithm may be used to make this selection, as long as the reduced cost is positive.

Row selection, i.e., selecting the basic variable to enter the basis, is performed via a ratio test. Given that a column j has been selected, the corresponding row is selected according to Equation A.16 in the case of a maximization objective and Equation A.17 in the case of a minimization objective.

$$\min\left\{\frac{-b_i}{A_{i,j}} \mid A_{i,j} > 0\right\} \tag{A.16}$$

$$\min\left\{\frac{-b_i}{A_{i,j}} \mid A_{i,j} < 0\right\} \tag{A.17}$$

The Jordan Exchange operation, which transforms a matrix $A \mapsto A'$ given a pivot $(\hat{\imath}, \hat{\jmath})$, is straightforward and is shown in Equation A.18.

$$a'_{\hat{\imath},\hat{\jmath}} = \frac{1}{a_{\hat{\imath},\hat{\jmath}}} \text{ for } i = \hat{\imath}, j = \hat{\jmath} \tag{A.18a}$$

$$a'_{\hat{\imath},j} = -\frac{a_{\hat{\imath},j}}{a_{\hat{\imath},\hat{\jmath}}} \text{ for } i = \hat{\imath}, j \neq \hat{\jmath} \tag{A.18b}$$

$$a'_{i,\hat{\jmath}} = \frac{a_{i,\hat{\jmath}}}{a_{\hat{\imath},\hat{\jmath}}} \text{ for } i \neq i, j = \hat{\jmath} \tag{A.18c}$$

$$a'_{i,j} = a_{i,j} - a_{i,\hat{\jmath}} a_{\hat{\imath},j} \text{ for } i \neq \hat{\imath}, j \neq \hat{\jmath} \tag{A.18d}$$

Finally, one must determine a starting vertex. The original linear program is modified as shown in §3.4 of [23]. For each row, i, if $b_i > 0$, then add an additional variable, x_0, to the constraint with coefficient $a_{i,0} = 1$. An initial feasible point is then immediately available for $x = 0 \; \forall \; i \neq 0$ and $x_0 = \max(\max(b), 0)$. The Simplex Method is then applied, with x_0 being the first variable to leave the basis. When x_0 returns to the basis, a suitable starting vertex results from the removal of x_0.

B Integer Programming and the Branch-And-Bound Method

Integer programming expands upon the possible problems that can be modeled by linear programming. Decision variables in linear programming are optimized on a continuum, i.e., all decision variables, x, are real numbers, $x \in \mathbb{R}^n$. Integer programming allows for certain decision variables, y, to take on integer-only values, i.e., $y \in \mathbb{Z}^n$. Strictly speaking, a programming formulation for which all decision variables are integer (i.e., integer and binary) is called an *integer program* (IP), whereas a programming formulation for which some decision variables are integer while others are linear is called a *mixed integer-linear program* (MILP). The discussion that follows is informed largely by Wolsey's text [62] from which I cite many definitions, etc. Additional clarification comes from course notes [42].

B.1 Integer Programming

Integer programming allows one to model specialized decision cases. Take for example one of the most well-known problems in mathematical programming and optimization, the *Knapsack Problem*. A version of the Knapsack problem is described as follows:

- A knapsack can hold at most b pounds.
- There are n possible items that can be placed in the bag.
- Each item is characterized by a preference, or benefit, c_i, and a weight, a_i
- One would like to maximize the benefit associated with a knapsack

The decision variables, y_is, for the Knapsack Problem provide its integer nature. Any given item in the above formulation can only be added once. Indeed, consider that for any viable solution, each item is in one of two distinct states: included in the knapsack or excluded from the knapsack. This duality of states provides a natural usage of binary variables, i.e., a variable that has only two states, 0 and 1. Accordingly, the Knapsack Problem as an integer program is formulated as Equation B.1.

$$\max \sum_{i \in I} c_i y_i \tag{B.1a}$$

$$\text{s.t.} \sum_{i \in I} a_i y_i \leq b \tag{B.1b}$$

$$y_i \in \{0, 1\} \qquad \forall i \in I \tag{B.1c}$$

Optimization problems are given as formulations, a series of inequality equations. Both domain knowledge and geometrical investigation can provide better formulations than may be evident from an initial formulation. Formally, a formulation forms a polyhedron.

Definition B.1. *A subset of* $\mathbb{R}^n$ *described by a finite set of linear constraints* $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ *is a* ***polyhedron****.*

Definition B.2. *A polyhedron* $P \subseteq \mathbb{R}^{n+p}$ *is a* ***formulation*** *for a set* $X \subseteq \mathbb{Z}^n \times \mathbb{R}^p$ *iff* $X = P \cap (\mathbb{Z}^n \times \mathbb{R}^p)$

As previously noted, more than one formulation can be viable for a given problem. Let us return to the Knapsack Problem in Equation B.1. Consider a knapsack with $b = 5$ and items with $a_1 = 2$, $a_2 = 3$, $a_3 = 4$. The original formulation is as follows.

$$\max \sum_{i \in I} c_i y_i \tag{B.2a}$$

$$\text{s.t. } 2y_1 + 3y_2 + 4y_3 \leq 5 \tag{B.2b}$$

$$y_i \in \{0, 1\} \qquad \forall i \in I \tag{B.2c}$$

The set of feasible solutions here forms a polyhedron from the points $Y = (0,0,0), (1,0,0), (0,1,0), (0,0,1), (1,1,0)$. The optimal solution will depend on values given to each item's benefit, c_i. However, the formulation as provided defines a solution space larger than the specific points mentioned here. One could add a constraint, say,

$$y_1 + y_3 \leq 1 \tag{B.3}$$

or

$$y_2 + y_3 \leq 1. \tag{B.4}$$

These two derived constraints state that the third item, if chosen, can not be included with either the first or the second item. The resulting formulation is shown in Equation B.5.

$$\max \sum_{i\in I} c_i y_i \tag{B.5a}$$

$$\text{s.t. } 2y_1 + 3y_2 + 4y_3 \leq 5 \tag{B.5b}$$

$$y_1 + y_3 \leq 1 \tag{B.5c}$$

$$y_2 + y_3 \leq 1 \tag{B.5d}$$

$$y_i \in \{0,1\} \qquad \forall i \in I \tag{B.5e}$$

It is obvious that Equation B.5 is a different formulation than Equation B.2. For example, the point $(0.9, 0.5, 0.4)$ resides in the feasible solution space of Equation B.2 but is outside of the feasible solution space of Equation B.5. Intuitively, a smaller solution space can be searched more quickly, thus *tighter* formulations require less time to solve in general.

The notion of one formulation being "better" than another can be formally expressed.

Definition B.3. *Given a feasible solution space set $X \subseteq \mathbb{R}^n$ and two formulations, P_1 and P_2, for X, P_1 is a **better formulation** than P_2 if $P_1 \subset P_2$.*

There is, of course, a limit to the formulations one can develop for a given problem. A fully-restricted solution space, i.e., one that is as tightly bounded as possible, is called the problem's *convex hull*.

Definition B.4. *Given a set $X \subseteq \mathbb{R}^n$, the **convex hull** of X, denoted $conv(X)$, is defined as: $conv(X) = \{x : x = \sum_{i=1}^{t} \lambda_i x_i, \sum_{i=1}^{t} \lambda_i = 1, \lambda_i \geq 0$ for $i = 1, \ldots, t$ over all finite subsets $\{x^1, \ldots, x^t\}$ of $X\}$.*

Because the extreme points of $conv(X)$ all lie in X, the equivalent LP can be used instead of the IP. Convex hull formulations are rarely seen in practice, however, because they require an exponential number of additional constraints [62]. While the convex hull of a given problem may not be discovered in practice, the feasible solution space most assuredly is reduced by most solution techniques. From a geometrical point of view, this acts as cutting off solution space from some original larger space through the addition of constraints as shown above. Accordingly, these additional constraints are termed *cutting planes*.

B.2 The Branch and Bound Algorithm

One of the most popular solution techniques used to solve integer programs is an algorithm called *Branch and Bound* (BNB). At its core, BNB is a divide-and-conquer search algorithm that uses an enumeration tree to find optimal solutions to $\mathcal{NP}$-hard IPs and MILPs. There are a number of ways to speed up the search based on general techniques and problem-specific insights, a number of which have been discussed in the previous section. This section highlights the basic nature of the algorithm and discusses lightly some of the variety of solution strategies available. Again, the discussion here comes largely from [62] and [42].

BNB utilizes the *relaxation* of a given IP or MILP. A relaxation is a related reformulation of a given problem that is generally easier to solve. In the case of a linear programming relaxation, integer variables in the IP or MILP are relaxed and allowed to be linear variables. Solving this formulation is advantageous because it provides an *upper bound* for the IP or MILP. Similarly, solving a dual, given that it is feasible and has a finite solution, or using some other heuristic provides a *lower bound*. These bounds allow the search tree to terminate, or *prune*, a given *branch*.

A simple example greatly helps to show the process of the BNB algorithm. Let us use a specific instance of Equation B.1. This example is contrite and does not involve pruning; it is useful simply to show how branching occurs. Consider a knapsack with three items. The items have benefits of 0.5, 1, and 1.4, respectively and weights of 2, 3, and 4 pounds, respectively. The knapsack can hold 5 pounds. The integer program is shown in Equation B.6 and the LP relaxation is shown in Equation B.7.

$$\max \ Z^{IP} = 0.5y_1 + y_2 + 1.4y_3 \tag{B.6a}$$
$$\text{s.t.} \ 2y_1 + 3y_2 + 4y_3 \leq 5 \tag{B.6b}$$
$$y_1, y_2, y_3 \in \{0, 1\} \tag{B.6c}$$

$$\max \ Z^{LP} = 0.5y_1 + y_2 + 1.4y_3 \tag{B.7a}$$
$$\text{s.t.} \ 2y_1 + 3y_2 + 4y_3 \leq 5 \tag{B.7b}$$
$$y_1, y_2, y_3 \in [0, 1] \tag{B.7c}$$

Solving the relaxation provides an upper bound of $Z^{LP} = \frac{26}{15}$, a (non-integer) solution of $y' = (0, \frac{1}{3}, 1)$ and a root node for the BNB search tree, shown in Figure B.1.

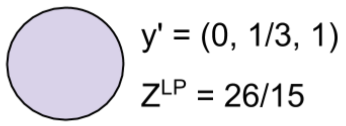

Figure B.1: The root node for the BNB algorithm associated with Equation B.6.

The algorithm then chooses a variable on which to *branch*. Formally branching divides the set of feasible solution spaces in two. Given a solution space S, branching on a binary variable y_i, produces two new solution spaces.

$$\begin{aligned} S_1 &= S \cap \{y : y_i = 0\} \\ S_2 &= S \cap \{y : y_i = 1\} \end{aligned} \tag{B.8}$$

If the variable is non-binary integer, given a non-integer feasible solution, y', one produces the following new spaces.

$$\begin{aligned} S_1 &= S \cap \{y : y_i \leq \lfloor y'_i \rfloor\} \\ S_2 &= S \cap \{y : y_i \geq \lceil y'_i \rceil\} \end{aligned} \tag{B.9}$$

Arbitrarily, for this example case, one could choose to branch on y_2. The resulting relaxations are shown as Equation B.10 for $y_2 = 0$ and Equation B.11 for $y_2 = 1$.

$$\max \; Z^{LP} = 0.5y_1 + 1.4y_3 \tag{B.10a}$$

$$\text{s.t. } 2y_1 + 4y_3 \leq 5 \tag{B.10b}$$

$$y_1, y_3 \in [0, 1] \tag{B.10c}$$

$$\max \; Z^{LP} = 0.5y_1 + 1 + 1.4y_3 \tag{B.11a}$$

$$\text{s.t. } 2y_1 + 3 + 4y_3 \leq 5 \tag{B.11b}$$

$$y_1, y_3 \in [0, 1] \tag{B.11c}$$

Each of these new subproblems become *active nodes* and are added to the *active list*. Active nodes are subproblem nodes that have been recognized by the algorithm and the next subproblem to solve is chosen from the active list by some strategy. For this simple case, both subproblems are solved and the resulting values are shown in Figure B.2.

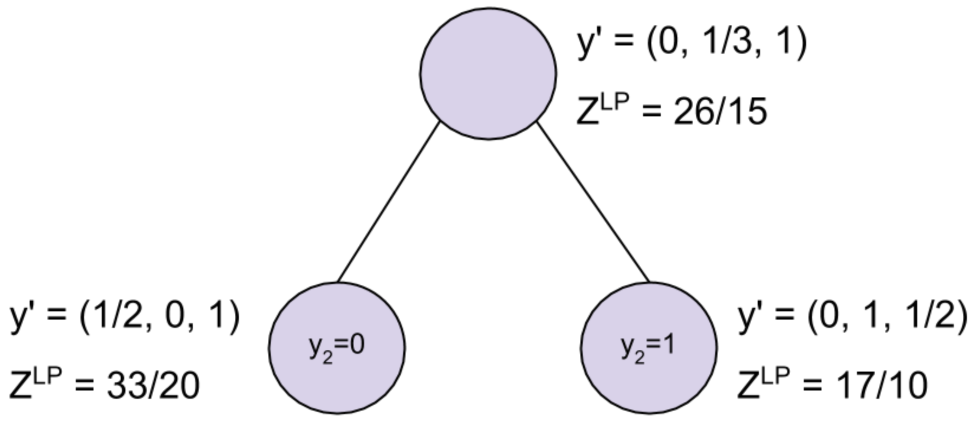

Figure B.2: The first two branches of the BNB algorithm associated with Equations B.10 and B.11.

One could continue in this manner, branching on subsequent variables and enumerating all possible solutions, to eventually reach the optimal solution of $y^* = (1, 1, 0)$. However, so far we have ignored pruning, the act of terminating a branch of the search tree, knowing that no further useful information can be gained from its investigation. Pruning provides the "bounding" aspect of the Branch and Bound algorithm.

At any point in the process of the BNB algorithm, there is a known global upper bound, U, to the optimal solution and lower bound, L, to the optimal solution. Accordingly, a branch of the enumeration search tree can be pruned in three instances:

1. The subproblem is optimal, given its subspace of the feasible option space.
2. The subproblem has a known upper bound that is lower than the global lower bound or the subproblem has a known lower bound that is larger than the global upper bound.
3. The subproblem is infeasible.

With the above background, the actual BNB algorithm can be presented.

```
Data: Decision variables, an objective function, and a set of constraints.
Result: Optimal values for the decision variables or a flag denoting infeasibility or unboundedness.
Perform any preprocessing operations;
Derive a lower bound, $L$, via a heuristic;
Place original problem on the active list;
while The active list is not empty do
    Use a strategy to select a candidate node ($S$) from the active list;
    Solve the LP relaxation to get an upper bound for the candidate, $U(S)$;
    if $U(S) > U$ then
        $U \leftarrow U(S)$;
    end
    if $S$ is infeasible then
        prune the branch;
    end
    else if $U(S) > L$ then
        $L \leftarrow U(S)$;
    end
    else if $U(S) < L$ then
        prune the branch;
    end
    else
        branch on $S$;
        add new subproblems to the active list;
    end
    Remove $S$ from the active list;
end
```

Algorithm 3: The Branch and Bound Algorithm

There are three ways to assist, or speed up, the Branch and Bound algorithm as highlighted above:

1. Preprocessing
2. Lower-bound heuristics
3. Node selection strategies

Preprocessing is a step provided by many solvers. It generally involves an investigation of the problem instance in order to minimize future work. Preprocessing can affect solution bounds by tightening bounds or providing cutoffs to the solver (i.e., preformed feasible solutions). It can speed up the internal Simplex Method processing by informing the solver as to good simplex pricing strategies. The feasible solution space can also be reduced by finding redundant constraints and providing cutting planes, as discussed in the previous section. Finally, the preprocessing step can *a priori* fix certain decision variables. The variable fixing algorithm is straightforward.

```
Data: An constraint matrix, $A$, objective coefficients, $c$, and decision variables $x$ with decision
      variable lower bounds, $l$, and upper bounds $u$.
Result: A (possibly empty) set of fixed variables.
foreach decision variable, $x_j$, do
    if $a_{i,j} \geq 0 \ \forall \ i$ and $c_j < 0$ then
        $x_j \leftarrow l_j$;
    end
    else if $a_{i,j} \leq 0 \ \forall \ i$ and $c_j > 0$ then
        $x_j \leftarrow u_j$;
    end
end
```

Algorithm 4: The Variable Fixing Algorithm for a Maximization Objective Function

A variety of lower-bound heuristics exist. Some of the most popular involve solving heavily restricted versions of the original problem or diving down the enumeration tree, rounding fractional integer values. There are also problem-specific heuristics that depend on well-known problem structures.

Finally, there exist nominally three well-used node selection strategies. The first is called the *Best Node Search* (BNS) which chooses the next best node in the active list based based on the node's upper bound. This requires large movement around the search tree, effectively solving dissimilar relaxations. The second is the well-known *Depth First Search* (DFS). A DFS for an IP-enumeration tree is beneficial because subsequent relaxations are related, which allows for *warm start* of the LP relaxations. Warm starts allow subsequent relaxations to be solved quickly because good approximations to the optimal solution can be provided. The final strategy is a *BNS-DFS hybrid*. The hybrid strategy involves estimating an optimal value, performing a DFS until the relaxation's optimal value is below that of the estimation, and then choosing the next-best node to continue.

C Cyclopts HDF5 Database Layout

This appendix details the exact database layout used by Cyclopts for the `ExchangeFamily`, `StructuredRequest` species, and `StructuredSupply` species.

C.1 Parameter Space

Both front-end and back-end species record the state of every point in a given parameter space in a data set called `/Species/<species type>/Points`, where `<species type>` is either `StructuredRequest` or `StructuredSupply`. Each point incorporates both fundamental and instance parameters as described in section 3.1. The tables associated with parameter spaces are described in Tables C.1-C.2.

Table C.1: Data-type description of the `/Species/StructuredRequest/Points` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
family	30-character string	A description of the problem family
f_fc	1-byte integer	As described in section 3.1
f_loc	1-byte integer	As described in section 3.1
f_mox	4-byte float	As described in section 3.1
f_rxtr	1-byte integer	As described in section 3.1
n_reg	4-byte unsigned integer	As described in section 3.1
n_rxtr	4-byte unsigned integer	As described in section 3.1
r_inv_proc	4-byte float	As described in section 3.1
r_l_c	4-byte float	As described in section 3.1
r_s_mox	4-byte float	As described in section 3.1
r_s_mox_uox	4-byte float	As described in section 3.1
r_s_th	4-byte float	As described in section 3.1
r_s_thox	4-byte float	As described in section 3.1
r_t_f	4-byte float	As described in section 3.1
r_th_pu	4-byte float	As described in section 3.1
seed	8-byte integer	The random seed used to generate an instance.

Table C.2: Datatype description of the `/Species/StructuredSupply/Points` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
family	30-character string	A description of the problem family
d_f_mox	4-length array of 8-byte floats	As described in section 3.1
d_f_thox	4-length array of 8-byte floats	As described in section 3.1
d_th	3-length array of 8-byte floats	As described in section 3.1
f_fc	1-byte integer	As described in section 3.1
f_loc	1-byte integer	As described in section 3.1
f_mox	4-byte float	As described in section 3.1
f_rxtr	1-byte integer	As described in section 3.1
n_reg	4-byte unsigned integer	As described in section 3.1
n_rxtr	4-byte unsigned integer	As described in section 3.1
r_inv_proc	4-byte float	As described in section 3.1
r_l_c	4-byte float	As described in section 3.1
r_repo	4-byte float	As described in section 3.1
r_s_mox	4-byte float	As described in section 3.1
r_s_mox_uox	4-byte float	As described in section 3.1
r_s_th	4-byte float	As described in section 3.1
r_s_thox	4-byte float	As described in section 3.1
r_t_f	4-byte float	As described in section 3.1
r_th_pu	4-byte float	As described in section 3.1
seed	8-byte integer	The random seed used to generate an instance.

C.2 Problem Instances

Problem instances are generated by problem species and are executed by problem families. Accordingly, both species and families can record information about instances. Front and back-end exchange species each record two types of information: details about each arc in an instance and a summary of species-specific information. The exchange family records information regarding each of the entities that comprise an instance: nodes, groups of nodes (having been translated from portfolios), and arcs. Further, aggregate summary information is also recorded.

C.2.1 Exchange Family

The exchange family records information regarding all major constructs in an exchange: nodes, groups, and arcs. A summary table is written to `/Family/ResourceExchange/ExchangeInstProperties`. Nodes and group data are recorded in an aggregate dataset located at `/Family/ResourceExchange/ExchangeNodes`, node group data is located at `/Family/ResourceExchange/ExchangeGroups`, and arc data is collected in the `/Family/ResourceExchange/ExchangeArcs` group. A dataset per instance UUID is used because it has been found to have better performance in the post-processing phase. A summary of family-specific instance data are detailed in Tables C.3-C.6.

Table C.3: Datatype description of the `/Family/ResourceExchange/ExchangeInstProperties` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
species	30-character string	A description of a problem species.
n_arcs	8-byte integer	The number of arcs in an NFCTP instance.
n_u_grps	8-byte integer	The number of supply groups in an NFCTP instance.
n_v_grps	8-byte integer	The number of demand groups in an NFCTP instance.
n_u_nodes	8-byte integer	The number of supply nodes in an NFCTP instance.
n_v_nodes	8-byte integer	The number of demand nodes in an NFCTP instance.
n_constrs	8-byte integer	The number of constraints in an NFCTP instance.
excl_frac	8-byte float	The fraction of arcs in a NFCTP graph that are exclusive.

Table C.4: Datatype description of the `/Family/ResourceExchange/ExchangeNodes` dataset.

Name	Data Type	Description
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
id	8-byte integer	A uniquely identifying value.
gid	8-byte integer	A unique value identifying an ExchangeGroup
kind	1-byte integer bitfield	Whether an object is associated with supply or demand.
qty	8-byte float	A quantity.
excl	1-byte integer bitfield	Whether or not an arc is exclusive.
excl_id	8-byte integer	A unique value identifying the mutually exclusive group an arc belongs to.

Table C.5: Datatype description of the `/Family/ResourceExchange/ExchangeGroups` dataset.

Name	Data Type	Description
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
id	8-byte integer	A uniquely identifying value.
kind	1-byte integer bitfield	Whether an object is associated with supply or demand.
caps	4-length array of 8-byte floats	Capacity RHS values.
cap_dirs	4-length array of 1-byte integer bitfields	Whether a constraint is greater or less-than
qty	8-byte float	A quantity.

C.2.2 Exchange Species

Both exchange species record information about each arc in an exchange instance. A parent group for arc data is defined under each species group. A group for each instance, whose name is the hex string of the UUID, is defined under the associated arc group. Finally, arc information associated with each instance is stored as a dataset in that instance's group. For example, the arc data for a given UUID of a front-end exchange is located as a dataset in the group `/Species/StructuredRequest/Arcs/<UUID hex>`. Summary information related to each species is also recorded in a data set for each species type located in the group `/Species/<species type>/Summary`. Tables describing species-specific instance

Table C.6: Datatype description of the `/Family/ResourceExchange/ExchangeArcs/<Instance UUID>` dataset.

Name	Data Type	Description
id	8-byte integer	A uniquely identifying value.
uid	8-byte integer	Supply node for an arc.
ucaps	4-length array of 8-byte floats	Capacity coefficients for a supply node.
vid	8-byte integer	Request node for an arc.
vcaps	4-length array of 8-byte floats	Capacity coefficients for a request node.
pref	8-byte float	Preference value of an arc.

data are detailed in Tables C.7-C.9.

Table C.7: Datatype description of the `/Species/<Species Type>/Arcs/<Instance UUID>` dataset.

Name	Data Type	Description
arcid	4-byte unsigned integer	The hex value of a UUID for an arc.
commod	4-byte unsigned integer	The commodity associated with an arc.
pref_c	4-byte float	Commodity-based preference of an arc.
pref_l	4-byte float	Location-based preference of an arc.

Table C.8: Datatype description of the `/Species/StructuredRequest/Summary` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
family	30-character string	A description of the problem family
n_r_th	4-byte unsigned integer	As described in section 3.1
n_r_f_mox	4-byte unsigned integer	As described in section 3.1
n_r_f_thox	4-byte unsigned integer	As described in section 3.1
n_s_uox	4-byte unsigned integer	As described in section 3.1
n_s_th_mox	4-byte unsigned integer	As described in section 3.1
n_s_f_mox	4-byte unsigned integer	As described in section 3.1
n_s_f_thox	4-byte unsigned integer	As described in section 3.1

C.3 Solutions

For every solution, data is added to the Cyclopts `/Results` dataset. Problem solutions are determined from problem instances, and are thus managed by a problem family. Aggregate solution information is provided in a family dataset `/Family/ResourceExchange/ExchangeSolutionProperties`. The full results of each solve, i.e., the amount of resources flowing across each arc, are recorded in a group specific to each solution UUID. Tables related to instance solutions are described in Tables C.10-C.12.

Table C.9: Datatype description of the `/Species/StructuredSupply/Summary` dataset.

Name	Data Type	Description
paramid	16-character string	The hex value of a UUID for a point in parameter space.
family	30-character string	A description of the problem family
n_r_th	4-byte unsigned integer	As described in section 3.1
n_r_f_mox	4-byte unsigned integer	As described in section 3.1
n_r_f_thox	4-byte unsigned integer	As described in section 3.1
n_s_uox	4-byte unsigned integer	As described in section 3.1
n_s_th_mox	4-byte unsigned integer	As described in section 3.1
n_s_f_mox	4-byte unsigned integer	As described in section 3.1
n_s_f_thox	4-byte unsigned integer	As described in section 3.1
n_s_repo	4-byte unsigned integer	As described in section 3.1

Table C.10: Datatype description of the `/Results` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an Exchange-Graph instance.
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
solver	30-character string	A description of the solver used.
problem	30-character string	A description of the problem family.
time	8-byte float	How long a solution took.
objective	8-byte float	The objective value associated with a solution.
cyclopts_version	12-character string	The version of Cyclopts used to generate a solution.
timestamp	26-character string	A timestamp of when a solution was ran.

Table C.11: Datatype description of the `/Family/ResourceExchange/`↪ `ExchangeInstSolutionProperties` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an ExchangeGraph instance.
instid	16-character string	The hex value of a UUID for an NFCTP graph instance.
pref_flow	8-byte float	The value of the product of preference and flow for arcs.
cyclus_version	20-character string	The version of Cyclus used to generate a solution.

Table C.12: Datatype description of the `/Family/ResourceExchange/ExchangeInstSolutions/<`↪ `Solution UUID>` dataset.

Name	Data Type	Description
arc_id	8-byte integer	
flow	8-byte float	

C.4 Post-Processing

Post-processing may be applied parameter, instance, and solution data. The exchange family, front-end species, and back-end species each contain a `PostProcess` dataset. Dataset layouts associated with post processing are described Tables C.13-C.15.

Table C.13: Datatype description of the `/Family/ResourceExchange/PostProcess` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an ExchangeGraph instance.
pref_flow	8-byte float	The value of the product of preference and flow for arcs.

Table C.14: Datatype description of the `/Species/StructuredRequest/PostProcess` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an ExchangeGraph instance.
c_pref_flow	8-byte float	The value of the product of commodity-based preference and flow for arcs.
l_pref_flow	8-byte float	The value of the product of location-based preference and flow for arcs.

Table C.15: Datatype description of the `/Species/StructuredSupply/PostProcess` dataset.

Name	Data Type	Description
solnid	16-character string	The hex value of a UUID for a solution to an ExchangeGraph instance.
c_pref_flow	8-byte float	The value of the product of commodity-based preference and flow for arcs.
l_pref_flow	8-byte float	The value of the product of location-based preference and flow for arcs.

C.5 Performance Studies

Chunk size is a critical parameter of HDF5 datasets that affects I/O performance. HDF5's storage layout is not contiguous; rather, data is separated into equal-sized *chunks*. Any reading or writing occurs on a chunk of data, rather than accessing an entire dataset. Accordingly, choosing a reasonable chunk size can greatly increase performance for known data access operations. In PyTables, the *compression level* of a dataset is also a tune-able parameter that affects I/O performance. Compression, of course, reduces overall database size. Therefore, an ideal compression is the largest possible that retains acceptable performance.

Originally, all Cyclopts datasets used a UUID-as-primary-key layout. For instance, rather than having tables with a layout described in Table C.6, a single table with an extra column naming the instance UUID was used. However, extremely long read times were encountered when post processing data. The basic procedure for performing a post-process operation included reading all rows associated with a UUID in an exchange species dataset, reading all rows associated with the same UUID in an exchange family dataset, selecting a value from each row (resulting in two vectors), and performing a dot product operation.

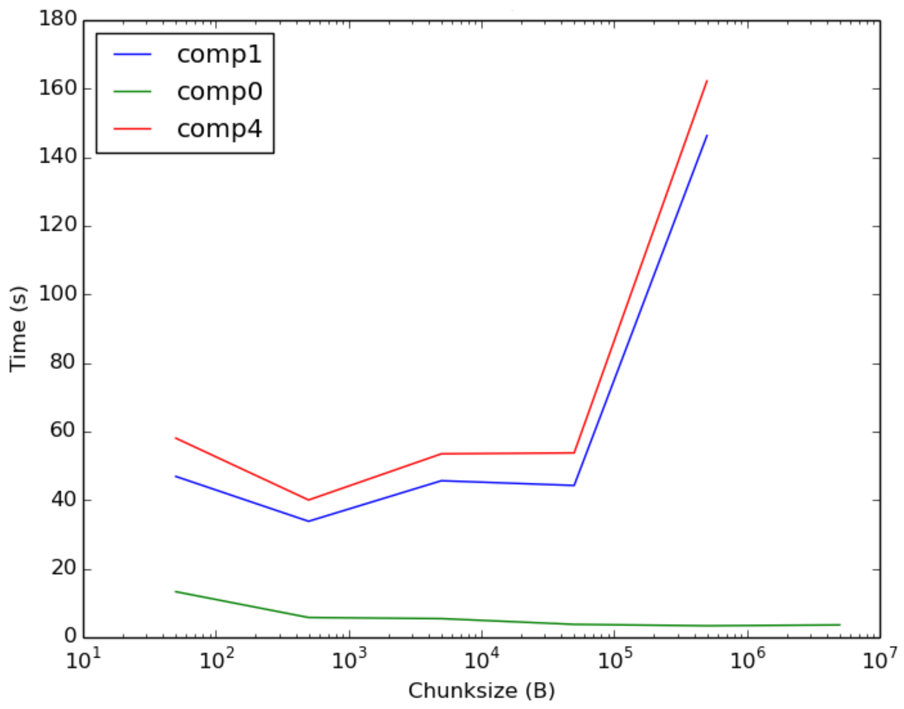

Figure C.1: Post-processing performance for 25 entries of a small-sized database for a variety of compression levels and chunk sizes.

In order to investigate possible chunk size and compression optimizations, a small ($\sim$ MBs) dataset and a large ($\sim$ GBs) dataset were created. The post-processing step was then run on 25 instances in each dataset. The operation was timed using the UNIX `time` command. An initial chunksize for each dataset was chosen to be proportional to the ratio of a normal L2 cache to row size and a compression level of four was selected per suggestions from the PyTables documentation [3]. For the performance study, chunk size and compression level were varied around these recommended values in order to determine if any tuning was available. The results of the study on the small dataset is shown in Figure C.1. The large dataset results is shown in Figure C.2.

Assuming some level of compression, an ideal chunksize range is identified for the small database of between $\sim 10^3 - 10^5$ bytes. Further the small database example confirms that the study's methodology is well founded: an ideal chunksize range is established. A similar optimal chunk size range is found for the large database. However, note that in this exercise, only $\sim 0.25\%$ of instances are post-processed. An optimal performance of > 80 seconds per instance is unacceptable.

A number of strategies exist for trying to increase performance. A classic strategy is pivoting the group-dataset structure such that data queries are made upon an entire group rather than rows in a dataset. In this example, such a pivot involves dividing the single, large dataset into n datasets, where n

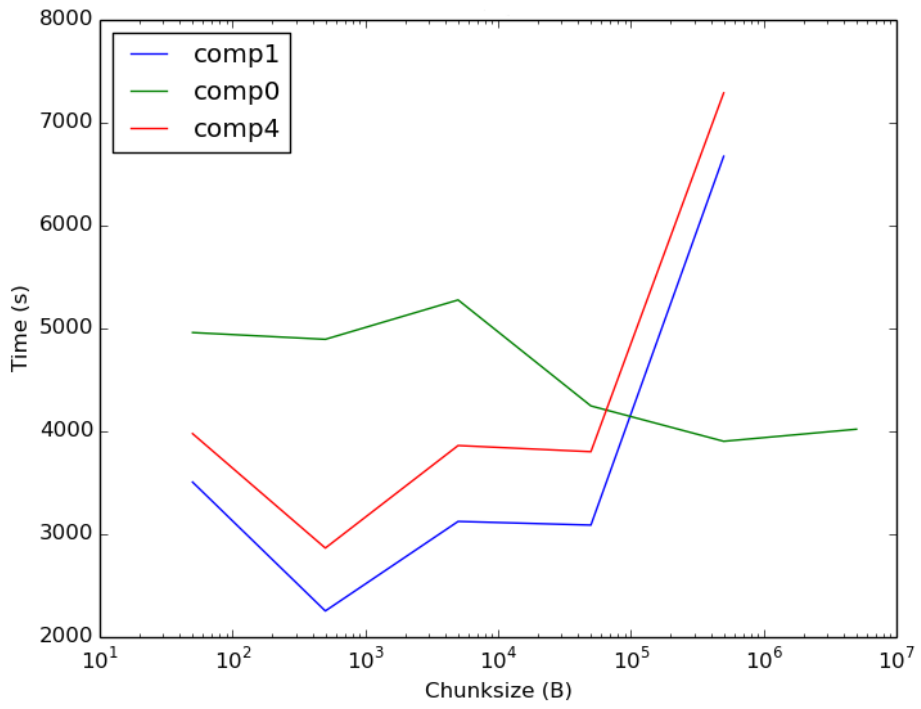

Figure C.2: Post-processing performance for 25 entries of a large-sized database for a variety of compression levels and chunk sizes.

is the number of unique primary keys, i.e., UUIDs.

Accordingly, an additional performance test was conducted with a new database layout. All datasets on which queries are made were pivoted such that new group nodes were added for each UUID, and all data for that UUID was appended to a dataset under the associated group. The post processing step was divided into the read and vector-population operations associated with the exchange family and the the read and vector-population operations associated with a species. The exact same operations were applied to a large database with the column-based layout and a large database with the group-based layout. Specific instances, increasing in size, were identified to be post-processed. The group-based results were compared with the column-based results and are shown in Figure C.3. The speed of each operation was compared directly for both layout strategies. The ratio of the group-strategy running time to the column strategy running time was then plotted. Therefore, a low ratio implies a large time savings, and a ratio close to unity implies almost no time savings. Times were calculated using the IPython magic `%timeit` command.

As can be seen, the group-based strategy performs quite well, over an order of magnitude better than the column-based strategy for species operations. Furthermore, species operations are shown to have a much larger speedup relative to family operations. This artifact is due to the fact that at the time of

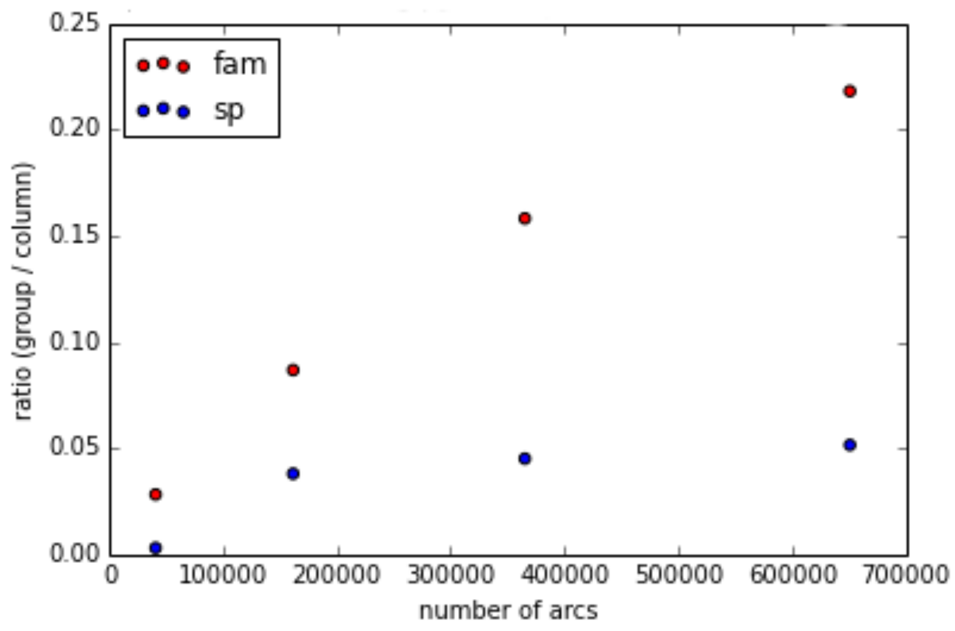

Figure C.3: The ratio of group-based queries to column-based queries as a function of problem size. A lower ratio indicates a faster process time for the group strategy over the column strategy.

this analysis, solution values were stored *only if* they were nonzero. When read, a data structure must be allocated and populated for each non-zero index rather than simply copying a block on data on disc. The writing of family-based solution values has since been updated to also write zero values to avoid this issue.

For the purposes of this study, the dataset-group pivot served the required purpose. Post-processing now performs satisfactorily for the operations needed and the database sizes experienced. However, if future performance issues arise, other strategies may be investigated. Perhaps the most fruitful of these will be returning to the single dataset layout and using PyTable's indexing feature.

D Cyclopts Command Line Interface

Cyclopts provides a rich command line interface (CLI) for instance generation, local execution, and remote execution. The CLI includes a number of useful utilities, however this section will only present those required for running the full Cyclopts workflow, both local and remote. The full set of CLI options is presented in Listing D.1.

Listing D.1: All available Cyclopts CLI options (the result of `cyclopts -h`).

```
usage: Cyclopts [-h]
              {convert,exec,pp,condor-submit,condor-collect,
              condor-rm,cde,combine,col2grp,dump}
              ...

positional arguments:
  {convert,exec,pp,condor-submit,condor-collect,condor-rm,cde,
   combine,col2grp,dump}
    convert Convert a parameter space defined by an input run
                  control file into an HDF5 database for a Cyclopts
                  execution run.
    exec Executes a parameter sweep as defined by the input
                  database and other command line arguments.
    pp Post process input and output.
    condor-submit Submits a job to condor, retrieves output when it has
                  completed, and cleans up the condor user space after.
    condor-collect Collects a condor submissions output.
    condor-rm Removes processes on condor for a user.
    cde Updates the Cyclopts CDE tarfile on a Condor submit
                  node.
    combine Combines a collection of databases, merging their
                  content.
    col2grp Moves input and output databases from id-column form
                  to id-group form.
    dump Dumps information about an instance database.
```

```
optional arguments:
  -h, --help show this help message and exit
```

D.1 Local Execution

When working locally, the primary workflow is `cyclopts convert`, followed by `cyclopts exec`, finishing with `cyclopts pp`. `cyclopts convert` converts a user-provided definition of a parameter space into an instance database. `cyclopts exec` then executes some or all of those instances, resulting in a solution database. Finally, `cyclopts pp` post-processes the instance and solution data. The options for each are described in Listings D.2, D.3, and D.4, respectively.

Listing D.2: CLI options for `cyclopts convert`.

```
usage: Cyclopts convert [-h] [--cycrc CYCRC] [--profile] [--proffile ↩
    PROFFILE]
                      [--species_module SPECIES_MODULE]
                      [--species_class SPECIES_CLASS] [--rc RC] [--db DB]
                      [-n NINST] [--count] [-v] [--debug] [-u UPDATE_FREQ]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                      $HOME/.cyclopts.rc useful for declaring global
                      family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --species_module SPECIES_MODULE
                      The module for the problem species
  --species_class SPECIES_CLASS
                      The problem species class
  --rc RC The run control file to use that defines a continguous
                      parameter space.
  --db DB The HDF5 file to dump converted parameter space points
                      to. This file can later be used an input to an execute
                      run.
  -n NINST, --ninstances NINST
                      The number of problem instances to generate per point
                      in parameter space.
  --count Only read in the run control file and count the number
                      of possible samplers that will be created.
  -v, --verbose Print verbose output during the conversion process.
  --debug Use objgraph and pdb to debug the conversion process.
  -u UPDATE_FREQ, --update-freq UPDATE_FREQ
                      The instance frequency with which to update stdout.
```

Listing D.3: CLI options for `cyclopts exec`.

```
usage: Cyclopts exec [-h] [--cycrc CYCRC] [--profile] [--proffile PROFFILE]
                   [--family_module FAMILY_MODULE]
                   [--family_class FAMILY_CLASS] [--db DB]
                   [--solvers [SOLVERS [SOLVERS ...]]]
                   [--instids [INSTIDS [INSTIDS ...]]] [--rc RC]
                   [--outdb OUTDB] [--conds CONDS] [-v]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                      $HOME/.cyclopts.rc useful for declaring global
                      family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --family_module FAMILY_MODULE
                      The module for the problem family
  --family_class FAMILY_CLASS
                      The problem family class
  --db DB An HDF5 Cyclopts database (e.g., the result of
                      'cyclopts convert').
  --solvers [SOLVERS [SOLVERS ...]]
                      A list of which solvers to use.
  --instids [INSTIDS [INSTIDS ...]]
                      A list of instids (as UUID hex strings) to run.
  --rc RC The run control file, which allows idetification of a
                      subset of input to run.
  --outdb OUTDB An optional database to write results to. By default,
                      the database given by the --db flag is use.
  --conds CONDS A dictionary representation of execution conditions.
                      This CLI argument can be used instead of placing them
                      in an RC file.
  -v, --verbose Print verbose output during execution.
```

Listing D.4: CLI options for `cyclopts pp`.

```
usage: Cyclopts pp [-h] [--cycrc CYCRC] [--profile] [--proffile PROFFILE]
                 [--family_module FAMILY_MODULE]
                 [--family_class FAMILY_CLASS]
                 [--species_module SPECIES_MODULE]
                 [--species_class SPECIES_CLASS] [--indb INDB]
                 [--outdb OUTDB] [--ppdb PPDB] [--verbose_freq VERBOSE_FREQ↩
                     ]
                 [--limit LIMIT]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                      $HOME/.cyclopts.rc useful for declaring global
                      family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --family_module FAMILY_MODULE
                      The module for the problem family
  --family_class FAMILY_CLASS
                      The problem family class
  --species_module SPECIES_MODULE
                      The module for the problem species
  --species_class SPECIES_CLASS
                      The problem species class
  --indb INDB An HDF5 Cyclopts input database (e.g., the result of
                      'cyclopts convert').
  --outdb OUTDB An HDF5 Cyclopts output database (e.g., the result of
                      'cyclopts exec').
  --ppdb PPDB An HDF5 Cyclopts post processed database (can be
                      combined with others via 'cyclopts combine'.
  --verbose_freq VERBOSE_FREQ
                      Stdout is informed of progress at the given processed
                      instance frequency.
  --limit LIMIT Post process only X instances (used for
                      profiling/testing).
```

A diagram explaining the role of the CLI workflow with respect to Cyclopts object tree (as seen in Figure 3.6) is shown below in Figure D.1.

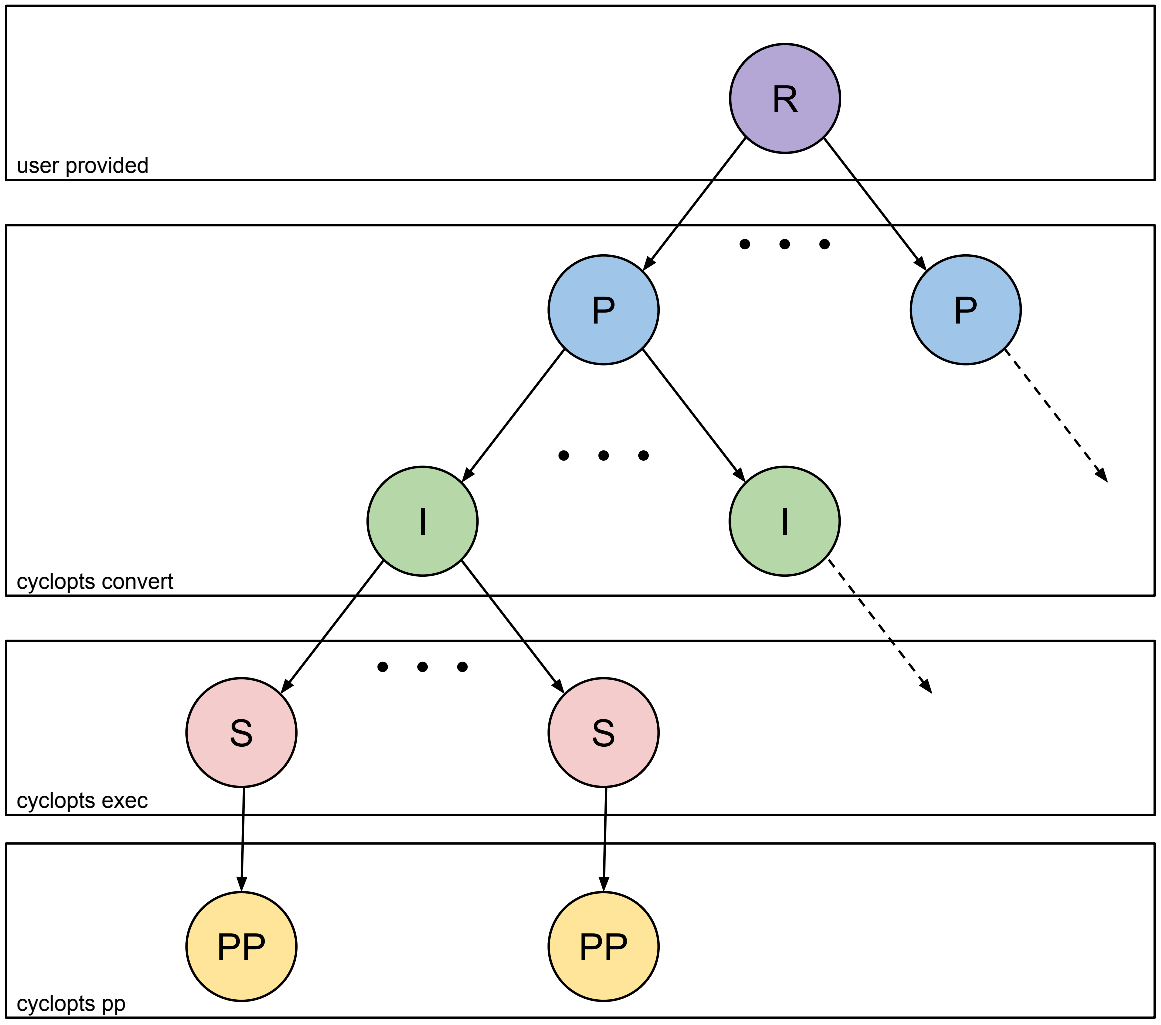

Figure D.1: The Cyclopts object tree structure is shown with boxes around each group of objects that are created given a CLI call. Note that the root node is determined from user-provided input.

D.2 Remote Execution

In order to execute Cyclopts on a Condor system, the submit node must contain the Cyclopts environment. That operation is supported by the `cyclopts cde` CLI, presented in Listing D.5. A job, the input of which is an instance database, can be submitted using `cyclopts condor-submit`. Upon completion, results can be collected with `cyclopts condor-collect`. The arguments for both are shown in Listings D.6 and D.7.

Listing D.5: CLI options for `cyclopts cde`.

```
usage: Cyclopts cde [-h] [--cycrc CYCRC] [--profile] [--proffile PROFFILE]
                  [--family_module FAMILY_MODULE]
                  [--family_class FAMILY_CLASS] [--source-path PREFIX]
                  [-u USER] [-t HOST] [--no-clean] [--keyfile KEYFILE]
                  [--fname FNAME]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                      $HOME/.cyclopts.rc useful for declaring global
                      family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --family_module FAMILY_MODULE
                      The module for the problem family
  --family_class FAMILY_CLASS
                      The problem family class
  --source-path PREFIX The path to cyclopts source.
  -u USER, --user USER The cde user name.
  -t HOST, --host HOST The remote cde submit host.
  --no-clean Do not clean up files.
  --keyfile KEYFILE An ssh public key file.
  --fname FNAME The function to wrap with cde.
```

Listing D.6: CLI options for `cyclopts condor-submit`.

```
usage: Cyclopts condor-submit [-h] [--cycrc CYCRC] [--profile]
                           [--proffile PROFFILE]
                           [--family_module FAMILY_MODULE]
                           [--family_class FAMILY_CLASS] [--rc RC]
                           [--db DB] [--instids [INSTIDS [INSTIDS ...]]]
                           [--solvers [SOLVERS [SOLVERS ...]]] [--count]
                           [-u USER] [-t HOST] [--keyfile KEYFILE]
                           [-d REMOTEDIR] [-k {dag,queue}] [--log]
                           [-p PORT] [--nodes [NODES [NODES ...]]] [-v]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                      $HOME/.cyclopts.rc useful for declaring global
                      family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --family_module FAMILY_MODULE
                      The module for the problem family
  --family_class FAMILY_CLASS
                      The problem family class
  --rc RC The run control file, which allows idetification of a
                      subset of input to run.
  --db DB An HDF5 Cyclopts database (e.g., the result of
                      'cyclopts convert').
  --instids [INSTIDS [INSTIDS ...]]
                      A list of instids (as UUID hex strings) to run.
  --solvers [SOLVERS [SOLVERS ...]]
                      A list of which solvers to use.
  --count Only count instances to be run.
  -u USER, --user USER The condor user name.
  -t HOST, --host HOST The remote condor submit host.
  --keyfile KEYFILE An ssh public key file.
  -d REMOTEDIR, --remotedir REMOTEDIR
                      The remote directory (relative to ~/cyclopts-runs) on
                      the submit node in which to run cyclopts jobs.
  -k {dag,queue}, --kind {dag,queue}
                      The kind of condor submission to use.
  --log Whether to keep a log of worker queue data.
  -p PORT, --port PORT The port to use for a condor queue submission.
  --nodes [NODES [NODES ...]]
                      The execute nodes to target.
  -v, --verbose Print output during the submisison process.
```

Listing D.7: CLI options for `cyclopts condor-collect`.

```
usage: Cyclopts condor-collect [-h] [--cycrc CYCRC] [--profile]
```

```
                            [--proffile PROFFILE] [--outdb OUTDB] [-u USER]
                            [-t HOST] [--keyfile KEYFILE] [-l LOCALDIR]
                            [-d REMOTEDIR] [--clean]

optional arguments:
  -h, --help show this help message and exit
  --cycrc CYCRC A global run control file, defaults to
                      $HOME/.cyclopts.rc useful for declaring global
                      family/species information.
  --profile Enable profiling.
  --proffile PROFFILE Name of profiling filename if profile is set.
  --outdb OUTDB An HDF5 Cyclopts output database (e.g., the result of
                      'cyclopts exec').
  -u USER, --user USER The condor user name.
  -t HOST, --host HOST The remote condor submit host.
  --keyfile KEYFILE An ssh public key file.
  -l LOCALDIR, --localdir LOCALDIR
                      The local directory in which to place resulting files.
  -d REMOTEDIR, --remotedir REMOTEDIR
                      The remote directory (relative to the users home
                      directory) in which output files from a run are
                      located.
  --clean Clean up the submit node after.
```

References

[1] Advanced reactors information system (aris). `https://aris.iaea.org/sites/core.html`. Accessed: 2014-02-02.

[2] Intel xeon processor e5-2690. `http://ark.intel.com/products/64596/Intel-Xeon-Processor-E5-2690-20M-Cache-2_90-GHz-8_00-GTs-Intel-QPI`. Accessed: 2015-01-24.

[3] Optimization Tips for pytables. `http://pytables.github.io/usersguide/optimization.html`. Accessed: 2014-12-18.

[4] Rokkasho start up delayed to 2016. `http://www.world-nuclear-news.org/WR-Rokkasho-start-up-delayed-to-2016-0311144.html`. Accessed: 2015-02-26.

[5] World statistics: Nuclear energy around the world. `http://www.nei.org/Knowledge-Center/Nuclear-Statistics/World-Statistics`. Accessed: 2015-02-02.

[6] 2009. International project on innovative nuclear reactors and fuel cycles (INPRO). Progress Report, IAEA.

[7] 10 CFR 40.A. 1985. Nuclear Regulatory Commission.

[8] Alted, Francesc, Ivan Vilata, et al. 2002–. PyTables: Hierarchical datasets in Python.

[9] Andrianova, E. A., V. D. Davidenko, and V. F. Tsibul'skii. 2008. Desae program for systems studies of long-term growth of nuclear power. *Atomic energy* 105(6):385–390.

[10] Bairiot, H, P Blanpain, D Farrant, T Ohtani, V Onoufriev, D Porsch, R Stratton, C Brown, P Deramaix, I Golovnin, et al. 2003. Status and advances in mox fuel technology. *Technical Report Series-International Atomic Energy Agency* 415:1–179.

[11] Behnel, S., R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. 2011. Cython: The best of both worlds. *Computing in Science Engineering* 13(2):31 –39.

[12] Bertel, Evelyne, and Thierry Dujardin. 2007. Management of recyclable fissile and fertile materials. *NEA News* 25(2).

[13] Blue Ribbon Commission. 2012. Reactor and fuel cycle technology subcommittee report to the full commission. Tech. Rep., Washington D.C.

[14] Boucher, L., and J. P Grouiller. 2006. "COSI": the complete renewal of the simulation software for the fuel cycle analysis. In *Fuel cycle and high level waste management*, vol. 1. Miami, FL, United States: ASME, New York, NY, USA.

[15] Bui, Peter, Dinesh Rajan, Badi Abdul-Wahid, Jesus Izaguirre, and Douglas Thain. 2011. Work queue+ python: A framework for scalable scientific ensemble applications. In *Workshop on python for high performance and scientific computing at SC11*.

[16] Cantor, Georg. 1890. Ueber eine elementare frage der mannigfaltigketislehre. *Jahresbericht der Deutschen Mathematiker-Vereinigung* 1:72–78.

[17] Carlsen, Robert. 2014. CyAn - Cyclus Analysis Tools.

[18] Chatfield, D. C., J. C. Hayya, and T. P. Harrison. 2007. A multi-formalism architecture for agent-based, order-centric supply chain simulation. *Simulation Modelling Practice and Theory* 15(2):153–174.

[19] Cochran, R.G., E.E. Lewis, N. Tsoulfanidis, and W.F. Miller. 1990. *The nuclear fuel cycle: analysis and management*. American Nuclear Society.

[20] Cook, Stephen A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual acm symposium on theory of computing*, 151–158. STOC '71, New York, NY, USA: ACM.

[21] Dantzig, George B. 1951. Maximization of a linear function of variables subject to linear inequalities. *New York*.

[22] Durpel, Van Den, A. Yacout, D. Wade, T. Taiwo, and U. Lauferts. 2009. DANESS v4.2: Overview of capabilities and developments. In *Proceedings of global 2009*. Paris, France.

[23] Ferris, Michael C, Olvi L Mangasarian, and Stephen J Wright. 2008. *Linear programming with matlab*, vol. 7. Society for Industrial and Applied Mathematics.

[24] Forrest, John, et al. 2014. COIN-OR Open Solver Interface. `https://projects.coin-or.org/Osi`.

[25] Gidden, M. 2013. An agent-based modeling framework and application for the generic nuclear fuel cycle. Prelim, University of Wisconsin, Madison.

[26] Gidden, Matthew. 2014. Cyclopts. `http://mattgidden.com/cyclopts/`.

[27] Gidden, Matthew, R. Carlsen, A. Opotowsky, O. Rakhimov, A. Scopatz, and P. Wilson. 2014. Agent-based dynamic resource exchange in cyclus. In *Proceedings of PHYSOR*. Kyoto, Japan.

[28] Gidden, Matthew, and Paul Wilson. 2013. An agent-based framework for fuel cycle simulation with recycling. In *Proceedings of GLOBAL*. Salt Lake City, UT, United States.

[29] Goldfarb, Donald. 1994. On the complexity of the simplex method. In *Advances in optimization and numerical analysis*, 25–38. Springer.

[30] Guerin, L., and M. Kazimi. 2009. Impact of alternative nuclear fuel cycle options on infrastructure and fuel requirements, actinide and waste inventories, and economics. Technical Report MIT-NFC-TR-111, MIT Center for Advanced Nuclear Energy Systems (CANES), Cambridge, MA, United States.

[31] Guerin, L., L. Van Den Durpel, B. Dixon, L. Boucher, and M. Kazimi. 2009. A benchmark study of computer codes for system analysis of the nuclear fuel cycle. Tech. Rep. MIT-NFC-TR-105, MIT.

[32] Guo, Philip J. 2011. CDE: Run any Linux application on-demand without installation. In *Proceedings of the 25th international conference on large installation system administration*. LISA'11, Berkeley, CA, USA: USENIX Association.

[33] Heinonen, Olli J. 2010. Safeguards in action: Iaea at rokkasho, japan. In *International atomic energy agency*.

[34] Holmgren, J., P. Davidsson, J. A. Persson, and L. Ramstedt. 2007. An agent based simulator for production and transportation of products. In *The 11th world conference on transport research, berkeley, USA*, 8–12.

[35] International Atomic Energy Agency (IAEA). 2007. *Liquid Metal Cooled Reactors: Experience in Design and Operation*.

[36] Jacobson, J. J., et al. 2009. *Vision user guide - vision (verifiable fuel cycle simulation) model*. Idaho National Lab, inl/ext-09-16645 ed.

[37] Julka, N., R. Srinivasan, and I. Karimi. 2002. Agent-based supply chain management-1: framework. *Computers & Chemical Engineering* 26(12):1755–1769.

[38] Kantorovich, Leonid Vitalievich. 1940. A new method of solving some classes of extremal problems. In *Dokl. akad. nauk sssr*, vol. 28, 211–214.

[39] Kok, Kenneth D. 2009. *Nuclear engineering handbook*. CRC Press.

[40] Law, Averill M., and David M. Kelton. 1999. *Simulation modeling and analysis*. 3rd ed. McGraw-Hill Higher Education.

[41] Levin, Leonid A. 1973. Universal sequential search problems. *Problemy Peredachi Informatsii* 9(3): 115–116.

[42] Luedtke, James. 2010. Class notes, integer programming (ISYE 720).

[43] Mazumdar, R., L.G. Mason, and C. Douligeris. 1991. Fairness in network optimal flow control: optimality of product forms. *Communications, IEEE Transactions on* 39(5):775–782.

[44] McCarty, Nolan, and Adam Meirowitz. 2007. *Political game theory: an introduction*. Cambridge University Press.

[45] Merriam-Webster Online. 2014. Merriam-Webster Online Dictionary.

[46] Murray, Paul. Personal communication, NWTRB meeting, Arlington, VA.

[47] Nagurney, Anna, June Dong, and Ding Zhang. 2002. A supply chain network equilibrium model. *Transportation Research Part E: Logistics and Transportation Review* 38(5):281 – 303.

[48] Oliver, Kyle M. 2009. Geniusv2: Software design and mathematical formulations for multi-region discrete nuclear fuel cycle simulation and analysis. Ph.D. thesis, University of Wisconsin-Madison.

[49] Rineiski, Andrei, Makoto Ishikawa, Jinwook Jang, Prabhakaran Mohanakrishnan, Tim Newton, Gerald Rimpault, Alexander Stanculescu, and Victor Stogov. 2011. Reactivity coefficients in bn-600 core with minor actinides. *Journal of nuclear science and technology* 48(4):635–645.

[50] Schneider, Erich A., Charles G. Bathke, and Michael R. James. 2005. NFCSim: A dynamic fuel burnup and fuel cycle simulation tool. *Nuclear Technology* 151(1):35–50.

[51] Schweitzer, Tyler Martin. 2008. Improved building methodology and analysis of delay scenarios of advanced nuclear fuel cycles with the verifiable fuel cycle simulation model (VISION).

[52] Scopatz, Anthony. 2013. XDress. `https://s3.amazonaws.com/xdress/index.html`.

[53] Shropshire, D. E., K. A. Williams, W. B. Boore, J. D. Smith, B. W. Dixon, M. Dunzik-Gougar, R. D. Adams, and D. Gombert. 2009. Advanced fuel cycle cost basis. Tech. Rep.

[54] Busquim e Silva, R., M.S. Kazimi, and P. Hejzlar. 2008. A system dynamics study of the nuclear fuel cycle with recycling: Options and outcomes for the US and brazil. Tech. Rep. MIT-NFC-TR-103, MIT Center for Advanced Nuclear Energy Systems (CANES), Cambridge, MA, United States.

[55] Song, H., Chen-Ching Liu, and J. Lawarree. 2002. Nash equilibrium bidding strategies in a bilateral electricity market. *Power Systems, IEEE Transactions on* 17(1):73–79.

[56] Strotz, R. H. 1953. Cardinal utility. *The American Economic Review* 384–397.

[57] Swaminathan, J., S. Smith, and N Sadeh. 1998. Modeling supply chain dynamics: A multiagent approach. *Decision Sciences* 29(3):607–632.

[58] Thain, Douglas, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience* 17(2-4):323–356.

[59] The HDF Group. 1997-NNNN. Hierarchical Data Format, version 5. Http://www.hdfgroup.org/HDF5/.

[60] Vlissides, John, R Helm, R Johnson, and E Gamma. 1995. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley* 49.

[61] Wilson, P.P.H., M. Gidden, K. Huff, and R. Carlsen. 2013. Cycamore : The cyclus additional modules repository. Http://cyclus.github.com/.

[62] Wolsey, Laurence. 1998. *Integer programming*. 1st ed. Hoboken, NJ: Wiley-Interscience.

[63] Yacout, AM, JJ Jacobson, GE Matthern, SJ Piet, DE Shropshire, and CT Laws. 2006. Vision–verifiable fuel cycle simulation of nuclear fuel cycle dynamics. *Waste Management*.

[64] Van der Zee, D. J., and J. Van der Vorst. 2005. A modeling framework for supply chain simulation: Opportunities for improved decision making. *Decision Sciences* 36(1):65–95.