Optimization Algorithms in Boiling Water Reactor Lattice Design

by

Chad Burns

SUBMITTED TO THE DEPARTMENT OF NUCLEAR SCIENCE AND ENGINEERING
IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF

BACHELOR OF SCIENCE IN NUCLEAR SCIENCE AND ENGINEERING
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Signature of Author _____

Chad Burns
Department of Nuclear Science and Engineering
Thursday, May 16, 2013

Certified by: _____

Kord Smith
M.I.T. Department of Nuclear Science and Engineering, Professor of the Practice
Thesis Supervisor

Accepted by: _____

Dennis Whyte
Professor of Nuclear Science and Engineering
Chairman, NSE Committee for Undergraduate Studies

OPTIMIZATION ALGORITHMS IN BOILING WATER LATTICE DESIGN

By

Chad Burns

SUBMITTED TO THE DEPARTMENT OF NUCLEAR SCIENCE AND ENGINEERING ON MAY 16TH, 2013
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF SCIENCE IN NUCLEAR SCIENCE AND ENGINEERING

1. Abstract

Given the highly complex nature of neutronics and reactor physics, efficient methods of optimizing are necessary to effectively design the core reloading pattern and operate a nuclear reactor. The current popular methods for optimization are Simulated Annealing and the Genetic Algorithm; this paper explores the potential for a new method called Greedy Exhaustive Dual Binary Swaps (GEDBS). The mandatory trade-off in computation is accuracy for speed; GEDBS is an exhaustive search and tends toward longer runtimes. While GEDBS performed acceptably for the criterion administered in this paper (local peaking and k_∞ on a Boiling Water Reactor (BWR) fuel lattice) the exhaustive nature of GEDBS will inevitably lead to combinatorial explosion for the addition of the potential dozens of factors that commercial application mandates. This issue may be resolved with the addition of metaheuristics to reduce the search space for GEDBS, or by an increasing computation.

Thesis Supervisor: Kord Smith, Ph.D.
Title: KEPCO Professor of the Practice of Nuclear Science and Engineering

Acknowledgements

I would like to take this opportunity to thank the entire department of nuclear science and engineering at MIT. So many have been so generous with their time, expertise, and passion and I am truly grateful for all the department has enabled me to perform.

Special thanks to Prof. Kord Smith, Koroush Shirvan, David Bloore, and Jeremy Roberts for the wisdom and support they have provided in the making of this thesis. They helped create an engaging and challenging project for the culmination of my MIT undergraduate education and offered essential incite from many more years of experience than I could have hoped for. Many thanks to all.

Table of Contents

1. **Abstract** 3
2. **Introduction** 6
3. **Technical Background** 8
   - 3.1. **Exhaustive Optimization Algorithms** 8
   - 3.2. **Stochastic Heuristic Methods** 9
     - 3.2.1. **Simulated Annealing** 10
     - 3.2.2. **The Genetic Algorithm** 14
   - 3.3. **Greedy Methods** 17
   - 3.4. **Greedy Exhaustive Dual Binary Swaps** 18
4. **Methods** 19
   - 4.1. **Fuel & Core Design** 19
   - 4.2. **CASMO-4** 21
   - 4.3. **Application of the GEDBS algorithm** 23
     - 4.3.1. **Metaheuristic modification to GEDBS** 25
5. **Results** 25
6. **Conclusions** 30
7. **References** 32
8. **Appendix A** 34
   - 8.1. **Python omnicont.py code** 34
   - 8.2. **Sample CASMO Submission** 40

2. Introduction

Since the inception of nuclear power engineers and scientists have pursued optimal core design; for longer still mathematicians have long sought methods for finding extrema over a large set of points. With the introduction of high-powered computers, numerical methods now exist to approximate and find the optima of complex functions. These algorithms are applicable to any system that can be quantized and weighted; their application to nuclear simulations has led to dramatic advancements in all aspects of reactor design, from the fuel pins to the core loading pattern.

Within a core design there are hundreds of independent constrained variables that can be altered, sometimes with drastic physical effect. In an average Boiling Water Reactor (BWR) core there are over four hundred fuel bundles, some reactors containing close to one thousand. Each bundle is composed of an 8-by-8 to 10-by-10 grid of fuel pins that are each filled with hundreds of fuel pellets. These pellets are the lowest level alterable structure for building a reactor core. Assuming just one variable per pellet, there is a preposterously large search space of $10^{20,594,603} \approx 10^{10^{7.31}}$ unique configurations. For comparison there are approximately 10^{80} atoms in the observable universe.

Therefore without a supercomputer the size of several universes there is no way to ensure the absolutely optimal core design. It is possible to achieve reasonably close approximations using optimization methods, the most popular of which are the genetic algorithm and simulated annealing. These optimization methods work to find the extrema of a single representative function by various iterative processes. To further reduce computational challenge, discrete levels or averages are adopted within bounding limits.

There are a near infinite number of factors, enrichments, and densities in reactor cores to consider, however all of these aspects are within the greater structure of a fuel bundle. Creating a simulation of fuel bundle arrangement is a computationally intense process that seriously challenges algorithms. The greedy exhaustive dual binary swaps (GEDBS) method was tested on the level of fuel lattice design within a BWR fuel bundle.

The GEDBS is an algorithm which replaces two fuel pins in a lattice with two from an available palette of pin types, measures the configuration and iterates from the result. This method is a more brute force approach than either the genetic algorithm (GA) or simulated annealing (SA). Ultimately the GEDBS will be compared with the effectiveness of SA and GA; this could eventually lead to the creation of even more efficient methods to find maxima and minima.

SA, GA, and GEDBS are all proceed by minimizing/maximizing a single function which encompasses all of the relevant variables to the problem. By simplifying a great deal of complexity to a single function the numerical evaluation of a 'better' solution can be more readily realized by a computer. This function often becomes stagnant around a local extreme rather than its goal of a global extreme. The amount the algorithm is willing to deviate from its current best value will indicate how large a potential barrier the algorithm can traverse, however with this ability to overcome large potential barriers comes inherent chaos and inefficiency. It is the careful balance of these factors that will prove the ultimate effectiveness of any optimization method. For example a fully exhaustive system that checked every possibility could reach the optimal solution, but still continue processing and check the worst solution in the process. A non-exhaustive pattern would iterate from the current best pattern to differing degrees and not waste computation time checking in the area of the worst responses.

3. Technical Background

The complex nature of reactor core design, fuel bundle assembly, and neutronics necessitate fast and accurate optimization methods. It is easy to have a fast program or an accurate program, but a single method that is both fast and accurate is exceedingly difficult to devise. There are several branches of algorithm that have evolved to meet the computational needs of modern engineering, the most popular of which are stochastic algorithms. The oldest and most rudimentary form of search is an exhaustive 'brute force' search, which is significantly slower than stochastics but more accurate.

3.1. Exhaustive Optimization Algorithms

An exhaustive optimization method attempts to find the best selection from a finite pool by explicitly testing each possibility. By testing each point, exhaustive methods guarantee that the absolute best discrete solution will be found. This 100% accuracy comes with an extreme cost in time and processing power. As the size of the search area increases, exhaustive methods tend toward combinatorial explosion. [1]

Combinatorial explosion is the phenomena where adding another unit to search through drastically increases the number of computations necessary. For example in Equation 1, C is the number of steps to complete the search, Z is the number of possible values, and n is the number of searched variables in the system. For a system of 25 blocks of which 2 are selected at a time there are $25^2 = 625$ cases to check. With 26 blocks there are an additional 51 cases to search; for three blocks selected at any one time there would be an increase of 15,000. Equation 2 shows computation steps for a system which does not allow repeated results.

$$C = Z^n \tag{1}$$

$$C = n! \tag{2}$$

For nuclear systems where often there are hundreds of cases to check with dozens of potential variables to sort, a brute force computation method is not viable. [2] With the constant emergence of newer, more powerful processors perhaps this 100% accurate method will eventually become an option, but until the era of quantum supercomputing more clever ways of optimizing are needed.

3.2. Stochastic Heuristic Methods

As opposed to the exhaustive methods which give an optimal answer, stochastics only attempt to return a near-optimal solution. Stochastic methods take orders of magnitude less computational steps and time to complete a simulation compared to exhaustive methods, making them ideal for complex real world application. As the name implies, this class of search works by incorporating all the relevant variables into a single value function, then introducing random/semi-random elements to perturb the value function in a more optimal direction. This element of randomness helps stochastic functions traverse potential barriers that surround local minima.

Since the 1990s, the field of stochastics has grown explosively, its expansion matching that of computing power. In addition to the comparative speed of stochastics, they are also utilized for the following situations: [3]

- No method for solving the problem to optimality is known.

- Although there is an exact method to solve the problem, it cannot be used on the available hardware in an acceptable timeframe.
- The heuristic stochastic method is more flexible than the exact method, allowing, for example, the incorporation of conditions that are difficult to model or the preferential exclusion of conditions that only need limited modeling.
- The stochastic method is used as part of a global procedure that seeks the optimum solution of a problem.

3.2.1. Simulated Annealing

Simulated Annealing was originally proposed in a 1983 issue of *Science* relating the then-theoretical method to statistical mechanics. SA was tested with the traditional traveling salesman problem and a computer chip design/wiring problem. The results were very impressive for both; SA became a leading technique for the wiring problem, and while SA was outperformed on the traveling salesman problem by other systems, its results were still highly noted. [4]

Formally, SA is a local search stochastic strategy that seeks the ground state. Over time SA has evolved many different improvements in general and specialty tweaks depending on the application. Recently Fast Simulated Annealing, a semi-local search function for faster convergence in higher dimensional problems, has become popular for faster optimization. [5] Another innovation is Nested Annealing, which is faster for smaller separable value heuristics. [6]

The underlying idea of SA is derived from statistical mechanics via metallurgy. As a liquid freezes the atoms and molecules naturally settle into the most energy efficient configuration. At full solidification the material is in its lowest energy state. [4] Mathematically, a cost or value function is selected, slight perturbations are made to lower the cost function, and finally the method converges to a single solution. For a set of solutions S there is an associated

cost function $T(s)$ where s is a single solution in the set of S. Let s^* and $T(s^*)$ denote the SA algorithm's most recent tested solution and value respectively. During each loop of the process, a slight perturbation is made to s^* such that the immediate neighbors, $s^{\pm}$, are tested. If $T(s^{\pm})$ is at a lower cost than $T(s^*)$ the new solution is accepted. [4] [7] [8]

To account for potential barriers between minima of the function SA has a metric to accept solutions that are not immediately better than the current best. This acceptance distribution is a probability of SA taking the new $T(s^{\pm})$ as the best given the probability defined in Equation 3.

$$P_a(s^{\pm}) = e^{\frac{\left(T(s^{\pm}) - T(s^*)\right)}{c}} \tag{3}$$

c is a predefined control parameter that alters how open the acceptance distribution is. Furthermore, different applications of SA commonly have a generating distribution, which tracks different possible valleys of the cost function to be explored. [7] Figure 1 shows the transition over potential barriers graphically.

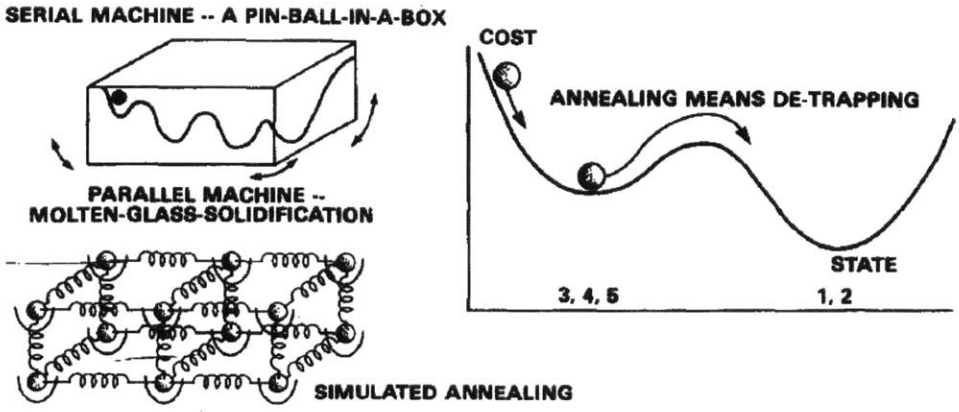

Figure 1– Material science visualization and cost function
These set of graphs show the transition of a complex 3D model of a set of cooling glass atoms to a 3D graph of their energy function to a final 2D simplification of temperature. The jump over the potential barrier in the right graph is due to the flexibility lent by the acceptance distribution. [5]

Finally, there is an overarching cooling scheme or annealing scheme that determines the rate at which $T(s)$ must settle to the minima. By altering this $\frac{dT}{dn}$ function with n as a unit of computation step, SA will go further or shorter down each potential branch of minima. For example if the annealing scheme dictated that T must decrease by at least 1 'degree' over any three steps, then the algorithm may take two steps uphill before coming across a decrease again. The uphill steps are always subject to the acceptance of Equation 3. Figure 2 gives a flow chart of the general SA process.

The results of SA are highly dependent on the topography of the value function in question. For a smooth function with a second derivative that doesn't change sign, SA has no uncertainties about converging on an optimal or near-optimal solution. For a highly erratic function SA will need to take many more steps to converge to an acceptable final solution, since there are more potential barriers to overcome between minima.

The possibilities for convergence of SA are convergence to a local minimum, convergence to a global minimum, and non-convergence; the best and most difficult result is convergence to a global minimum. In order for SA to converge upon the best solution, S^*, there must be a path from the starting solution to S^* where the highest value of the cost function along this path is less than or equal to $T(s^*) + h$. Here h is the height of the starting position relative to S^*. Now let every possible solution along the path communicate with S^* from a value distance of d^*. SA will converge to the global best solution under the conditions in Equation 4, as proposed by Hajek in 1988. [8]

$$\sum_{n=1}^{\infty} \frac{e^{-d^*}}{\frac{dT(n)}{dn}} = \infty \tag{4}$$

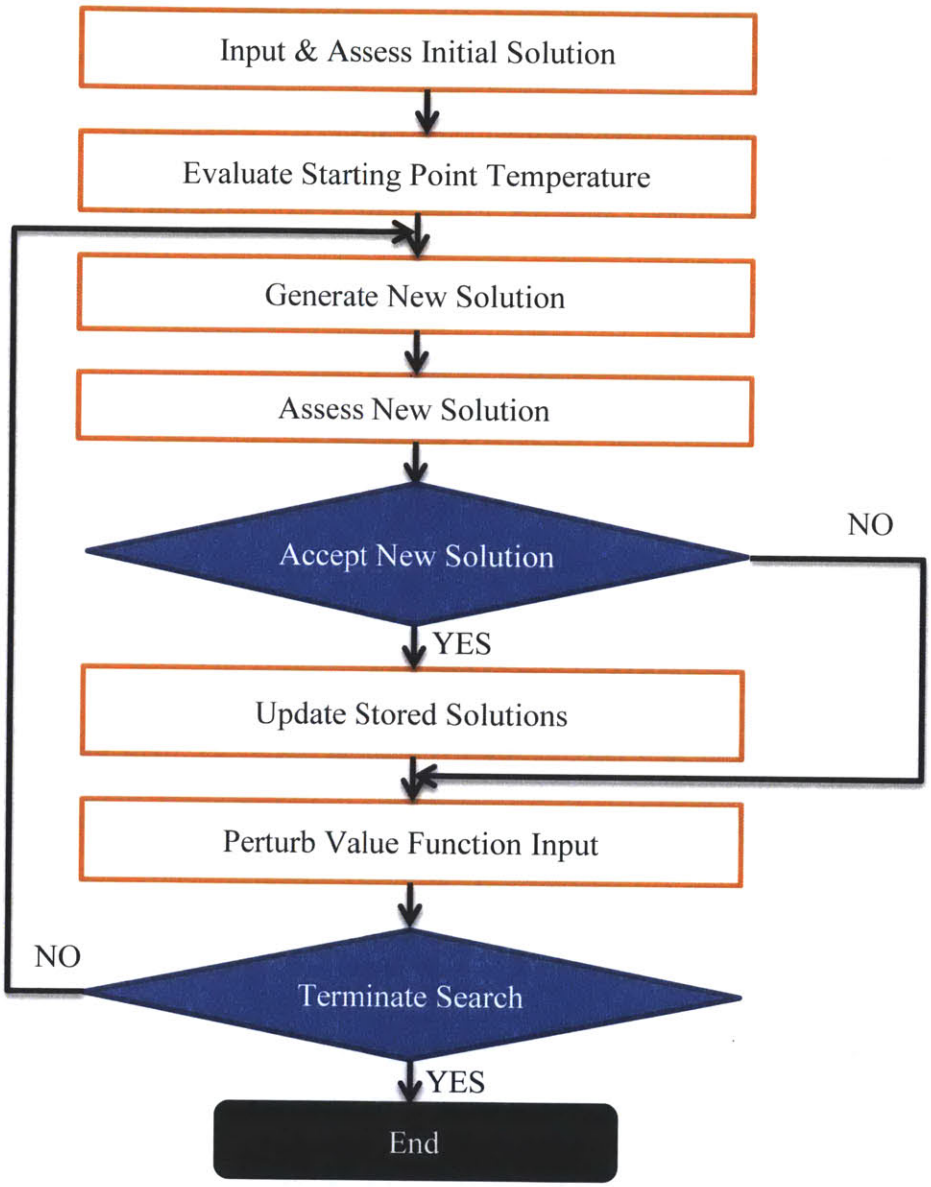

Figure 2– Overview of general SA search process [7]

Theoretically, given a wide acceptance criterion and extremely slow cooling scheme SA should find the global minimum every time, however this method would only converge given near infinite run time, as its computational intensity is comparable to, and in some cases higher than, an exhaustive algorithm. In this application SA would be a slightly directionalized random cost search, highly limited in its precision. As with computational methods in general, there is always an inverse relationship between accuracy and speed.

Finally, it is highly unlikely that SA would fail to converge on a solution, especially given that standard double type variables would need to be identical to at least 15 significant figures [9]. This outcome is only marginally possible for a highly erratic function with many local minima very close to each other at exactly the same value. Even for this rare case, there are simple modifications to the SA annealing schedule that would circumvent this problem.

3.2.2. The Genetic Algorithm

The Genetic Algorithm is a part of the evolutionary strategy field of computation that emerged in the 1960s. Evolutionary strategy is the general field in which solutions to numerical problems are attempted solved by modeling the algorithms after natural biological processes. The GA in particular is modeled after the reproductive cycle of genes and the Darwinian theories of survival of the fittest and adaptive mutations.

GAs were first proposed by John Holland in the 1960s; Holland and his research group developed this proposal into a functional system and in 1975 they published *Adaptation in Natural and Artificial Systems*, presenting the GA as an abstraction for biological evolution to formally study the phenomena of adaptation as it occurs in nature. [10] GAs are generally high

performers in changing conditions, since the solution is derived from a changing series of digital genes, chromosomes, alleles, and offspring.

The GA is a stochastic algorithm that operates off a value function; in the GA context the value function is known uniquely as a fitness function. Each individual in a population (solution in a set) is judged according to the fitness function. As per natural selection the most fit survive while inferior genes pass away. The GA starts with an initial set of individuals who create offspring, the most fit of the offspring and parents are kept and cross-bred with each other again.

It is easy to foresee how this could converge on a local minimum; the operations that make the GA effective are selection, crossover, and mutation. Selection is the process where the GA heuristic recognizes that a particular trait in a solution is highly desirable so the algorithm actively reproduces that trait in other chromosomes. Crossover refers to the reproduction process between two chromosomes; this operator will randomly chose a point in the sequence of variables that make up a solution and exchange everything after this point with another solution. This process is a crude parallel to haploid organisms. Finally mutation randomly changes one of the traits in a chromosome. Mutation can occur at any point, but any individual section has a very small chance of mutation. [10]

To apply these processes to lattice design, GA might *select* a $^{235}_{92}U$ enrichment of 5% to be prolific throughout all the pins and actively try to pass on that trait. A *crossover* might occur between two pins where the density of the $^{238}_{92}U$ is exchanged, and a *mutation* could occur where the gadolinium enrichment in the lattice randomly changes from 3% to 1%. Depending on a variety of other factors, any of these changes could improve or deteriorate the fuel bundle's performance, however since only the positive improvements are kept the generations will

improve on average and converge toward a set of optimal offspring. Once the mutation and interbreeding of these offspring stops yielding improving results, the best solution may be taken from the final optimal set. Figure 3 depicts a flowchart of the GA's application to a "search for solutions" problem.

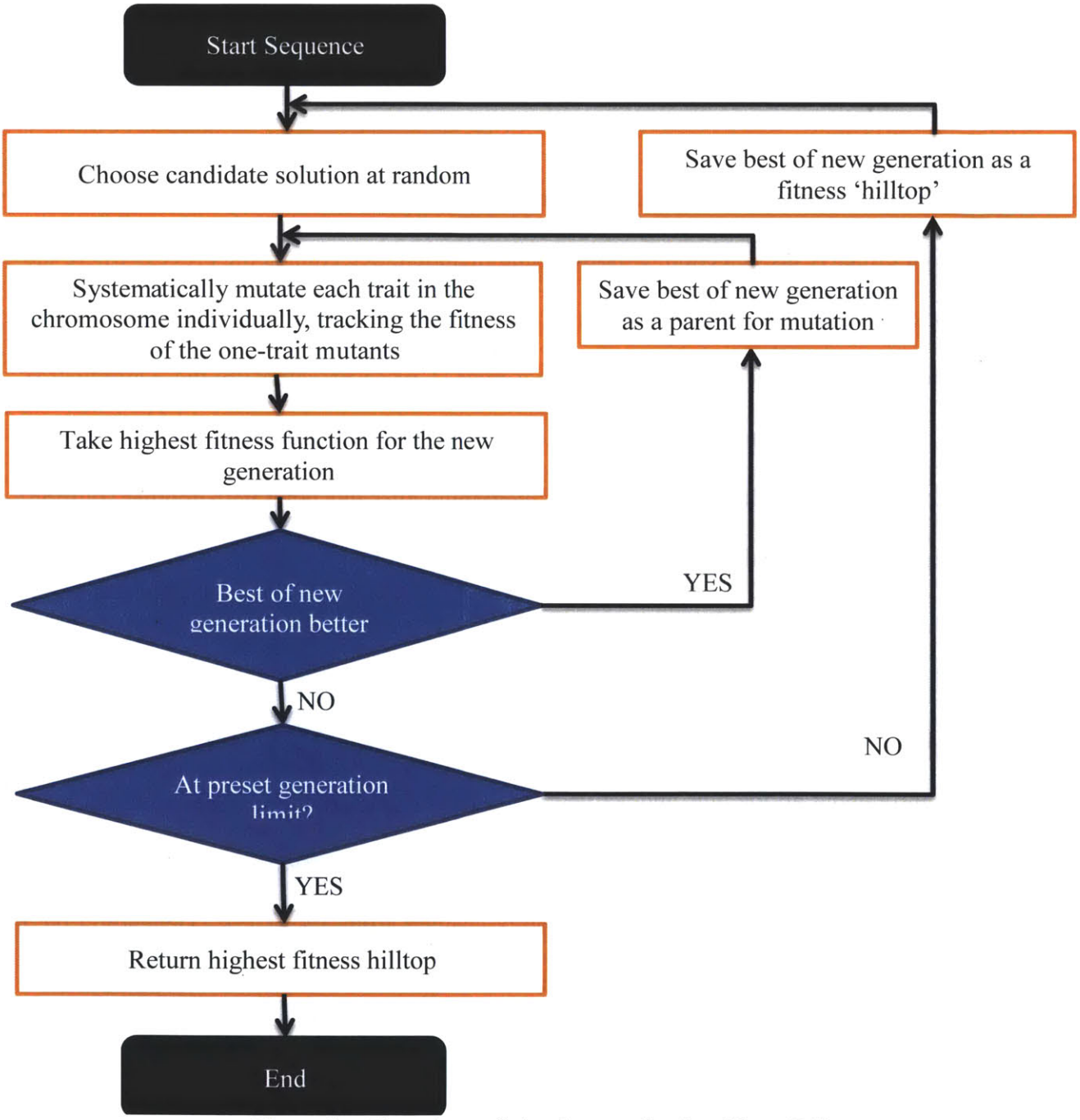

Figure 3 – Flowchart of simple genetic algorithm. **[10]**

Interestingly, it is possible to have GA simulate or behave extremely closely to SA for certain types of problems. If the mutation operator for the GA is taken as the annealing schedule for SA, there is very little difference between the two algorithms. [11] This makes sense because both of these mechanisms are the overarching control method by which the algorithms can bypass potential barriers. If both algorithms can only jump over barriers of the same height or height-width ratio, then it is highly likely they will behave similarly and return greatly similar, if not identical, results.

Because the GA does not mutate an offspring if has equal fitness to its parent, it must always converge. Granted this may not be convergent on the most fit solution, but it will return a local extreme at least. For example if there were a value function that was a flat plane in 3D space parallel to the independent variable axes, SA would not converge since it would change to any solution's equally valuable neighbor. The GA would simply sit at the randomly chosen starting position until the preset generation limit was met. This being said, the expected number of generations for the GA to stochastically converge is given in Equation 5. g_{conv} is the number of generations to converge and l is the length of the chromosomes under evaluation. [12]

$$g_{conv} = l \cdot \ln(l) \tag{5}$$

3.3. Greedy Methods

Another subclass of search/sort algorithm are greedy methods; these methods are not exclusive and any other form of optimization can be forced to behave as a greedy method, although doing so will negate many of the intrinsic advantages of the original method. When

evaluating the value function of a new solution compared to a previous best, the greedy method will immediately switch to the new function as better if and only if the value is higher. The greedy method will make the locally optimum choice at each step. [13]

This will lead to entrapment by local minima very frequently in searches where movement after comparison is not necessitated. For smaller problems that are similar to the traveling salesman case, greedy algorithms can very quickly yield acceptable solutions since the number of computational steps is only equal to the number of elements to sort. Since this runs in constant time, this is the fastest possible method.

SA could be made into a greedy method by modifying Equation 3 such that $P_a = 0$ which is true as $\lim_{c \to -0} e^{\frac{f_1 - f_2}{c}}$. The GA could also be made into a greedy method if instead of storing all the mutated offspring, the algorithm simply accepted the first improved solution and continued to operate. In both cases the ability to jump over potential barriers is reduced. For both greedy GA and SA the only way to find global minima on an erratic function would be by a random starting position or selection to that area, not at all a likely event given the magnitude of search space these methods are often applied to.

3.4. Greedy Exhaustive Dual Binary Swaps

The GEDBS algorithm is not (yet) a formal optimization search method in computation; however it has found use in academic and nuclear industry manual optimization calculations. [14] As the name implies, this is both a greedy method and exhaustive. Since exhaustive methods tend toward long run time with full accuracy and greedy methods tend toward shorter run times with low accuracy the combination of the two yields a theoretical middle ground.

GEDBS changes two elements in a solution before comparing this mutation to its predecessor. In this respect this is akin to performing two mutations of the GA at once. The changes are not random however; GEDBS exchanges a pin in a lattice with a different pin from a palette of possible pin types. For a grid of B independent pins in a fuel lattice and Z available pin types in the palette, the number of steps required by GEDBS is given in Equation 6.

$$n = B^2 \cdot Z^2 \tag{6}$$

Because GEDBS is greedy it is susceptible to the pitfalls of local extrema, however because it alters two elements at once rather than just one it has the capacity to jump over higher potential barriers. It is also guaranteed to jump over any barriers within the current solution's two element range because it is exhaustive; the algorithm completely cycles through all potential switching of any two given elements. If the program were to be fully exhaustive under the same switching sequence the number of steps would behave according to Equation 7. This runtime regime is unreasonably large for any modern computer.

$$n = B^B \cdot Z^B \tag{7}$$

4. Methods

4.1. Fuel & Core Design

Reactor cores are often made for perfect symmetry for ease of design and operational purposes; this concept holds for the smaller scale of lattice design. When designing a new fuel

lattice, symmetry can appreciably reduce the computation power needed as well as make easier conceptualizations for the designer. All lattice simulations performed for this paper were done in half symmetry. Diagonal symmetry is a standard symmetry for BWR lattice design.

Gadolinium, particularly $^{157}_{64}Gd$, is the burnable poison for many commercial nuclear reactors. $^{157}_{64}Gd$ has a thermal neutron cross section of 255,000 barns, larger even than $^{10}_{5}B$ thermal cross section. [15] [16] For the simulations in this paper gadolinium was used as a burnable poison ranging from 0-10% enrichment at 2.5% intervals, this gives five discrete levels of gadolinium enrichment as [0, 2.5, 5, 7.5, 10]. The gadolinium is used to control peaking, axial and radial power profiles, and the overall cycle depletion of the reactor. [16] These concepts are equally applicable to fuel bundle lattices as to the entire core loading.

In cases where gadolinium varied, the uranium enrichment, $\frac{N(^{235}_{92}U)}{N(^{235}_{92}U)+N(^{338}_{92}U)}$, was fixed at 5% with a density of 10.2 g/cm^3. In cases where uranium varied, the enrichment was fixed at 5% while density varied from 0.1 g/cm^3 to 10 g/cm^3 at intervals of 2.475 g/cm^3; there was no gadolinium present for those cases. For both sets of simulations all pins were taken as geometrically uniform.

The k_∞ eigenvalue reflects the criticality condition of a lattice that does not have neutron leakage. For an ideal lattice at steady operating conditions $k_\infty = 1$, any lattice with too large a k_∞ will create a local peak in power that is not tolerable . If a lattice has too low a k_∞ then it is losing more neutrons than it is creating, hindering power production. [16] [17] The upper limit used for simulation was $k_\infty \leq 1.1$.

The radial pin power peaking factor (PPPF) is a measure of local power compared to global energy production. This is monitored to ensure that any one pin in a lattice does not generate more power than its technical specifications permit. The local peaking can be

determined at the level of a single fuel pellet, however all PPPFs generated in GEDBS simulations were monitored at the fuel pin level. The maximum PPPF allowed was set at 1.3. The limits of both k_∞ and PPPF will be reflected in the heuristic value functions of SA, GA, and GEDBS.

All lattice optimization attempts were run at beginning of cycle (BOC) conditions. There was no fuel depletion for lattices. This zero burn up condition is important for how the lattice, and overall reactor, will behave upon the initial startup of a cycle; however realistically there would have to be a full depletion case run to ensure conditions are met continually through the end of cycle.

4.2. CASMO-4

Regardless of which optimization algorithm is used, at some point the new lattice design is analyzed by some code for neutronic evaluation. The code used in all simulations in this paper is the CASMO-4 lattice physics code of Studsvik Scandpower Inc. and run on the Massachusetts Institute of Technology Department of Nuclear Science and Engineering server clusters. [18]

CASMO is a lattice physics code for modeling PWR and BWR heterogeneous fuel designs, such as mixed concentrations of uranium-oxide and burnable poisons. The code is a multi-group two-dimensional transport code written in FORTRAN. CASMO can return an extremely wide array of data regarding almost any part of a core or lattice, however as previously stated the values of interest here are $k_\infty$and PPPF at BOC. Figure 4 shows the flow diagram of CASMO-4's main evaluation process.

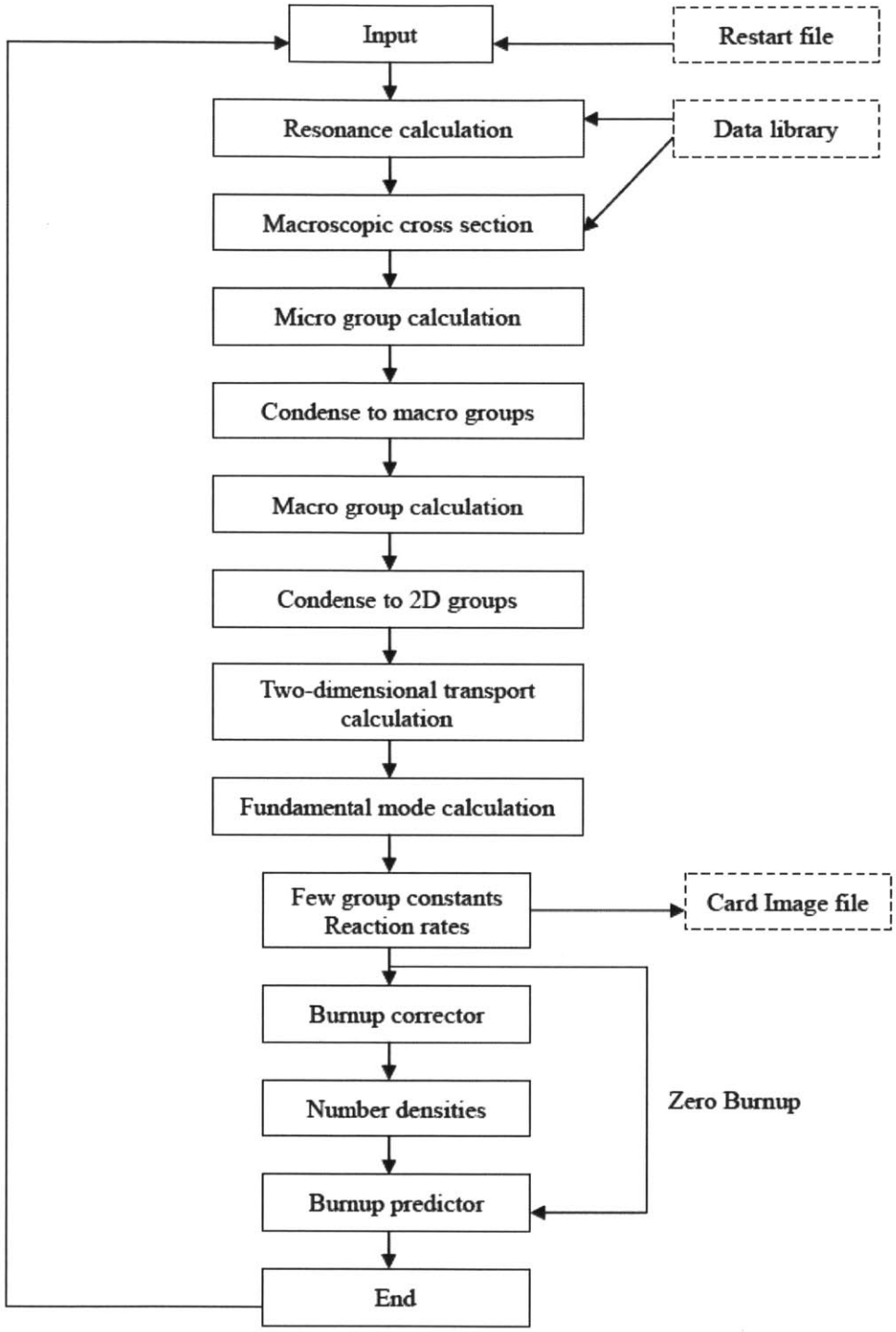

Figure 4 – Flow diagram of CASMO-4 lattice evaluation. [21]

4.3. Application of the GEDBS algorithm

GEDBS was applied to BOC lattice starting conditions under several different circumstances. In all cases there was half lattice symmetry of fifty-five possible locations, less four water tubes, to place a variable number of fuel types. Figure 5 shows a uniform starting map of the input for a palette of five different fuel pin types. The GEDBS algorithm will engage this set by changing the first two fuel pin locations, then the first and third, then first and fourth,

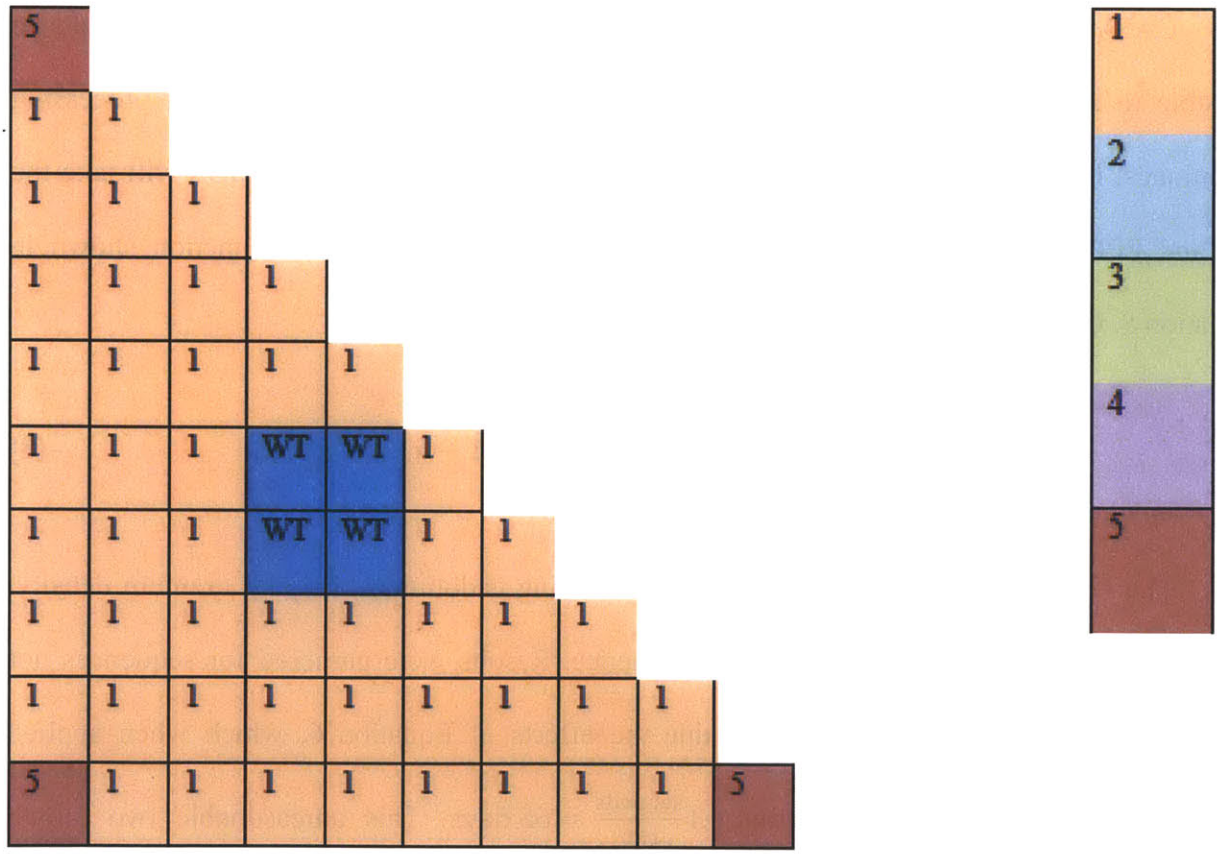

Figure 5 – Half lattice map with accompanying five pin type palette. The tubes labeled "WT" are water tubes not available for pin exchange. This is the starting position for all simulations.

and so forth until all possible dual combinations have been exhausted. The lattice map in Figure 5 shows the starting configuration for both the gadolinium variable and uranium variable simulations.

At each evaluation point, the python script running GEDBS will submit a job request to CASMO. The GEDBS script can be viewed in its entirety in section 8.1 of Appendix A. A sample job submission, which is a close ASCII interpretation of Figure 5, is in Section 8.2. Since this is a lattice of a BWR, there are water tubes at spaces number 18, 19, 24, and 25. This numbering system starts at zero in the upper left and works across then down. The water tubes are skipped in the GEDBS process.

CASMO will take on average 3 seconds per submission to return results. It would be possible to speed this processing time up significantly by incorporating parallel processing techniques; however such endeavors are outside the scope of this study. GEDBS will take the k_∞ and PPPF from CASMO's results and apply them to the objective function shown in Equation 8. GEDBS will continue iterating to maximize this value.

$$V(k_\infty, P_{PPF}) = 4 \cdot (1.3 - P_{PPF}) + 2 \cdot (1.1 - k_\infty) \tag{8}$$

The GEDBS algorithm was applied for a varying gadolinium, varying uranium density, and both varying uranium and gadolinium sequence. Results were garnered for sequences not involving both uranium and gadolinium due the effects of Equation 6, which when applied yielded a runtime of: $(55^2 \cdot 25^2)\text{runs} \cdot 3\frac{\text{seconds}}{\text{run}} \approx 65$ days. This unreasonable two month runtime is a result of the combinatorial explosion of the palette factor Z, which increased from 5 for only one changing variable to 5^2 for two changing variables.

4.3.1. Metaheuristic modification to GEDBS

A metaheuristic modification was applied to some sweeps of the GEDBS code. After finding that some pins never changed upon reaching a certain type, the python code could apply a metaheuristic and not evaluate changes in that pin to save computation time. The results from this metaheuristic application were compared to several results from sequences that did not apply the metaheuristic and found no difference in the end result. This metaheuristic cannot be applied upon starting the GEDBS algorithm, as many unsuccessful attempts to change the pin must occur before the metaheuristic recognizes it as optimally set.

5. Results

In the scenario where gadolinium varied, the final lattice solution is shown in Figure 6.

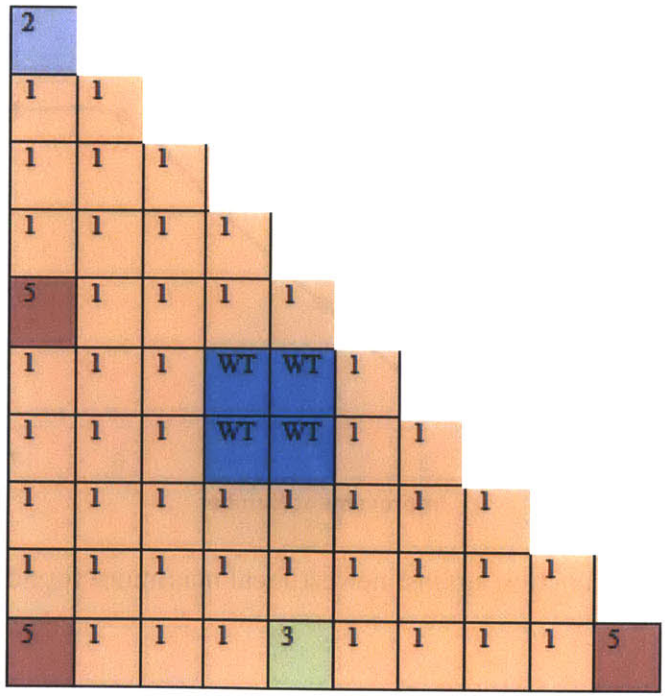

Figure 6- Final lattice configuration from gadolinium variable simulation.

The 0% gadolinium are pins 1 and the 10% are pins labeled 5; water tubes are zeros. The intermediate pins are at intervals of 2.5%. This result confirms one's intuition, the highest gadolinium concentration pins are placed at the corners to reduce the high pin powers resulting from thermal flux peaking in the water gaps. The selection is relatively symmetric about the central axes and homogenous in the center of the bundle.

Figure 8 and Figure 7 give graphic data on the convergence of GEDBS. Figure 7 shows only the maximums found in the iteration process; not every individual point of iteration. The value function starts off slow moving, then makes nearly linear improvements until asymptotically leveling off again. It took three sweeps of the GEDBS algorithm to converge. We can see from Figure 8 that the distribution of local maxima solutions that there is a bimodal distribution, with a strong trade-off between k_∞ and PPPF.

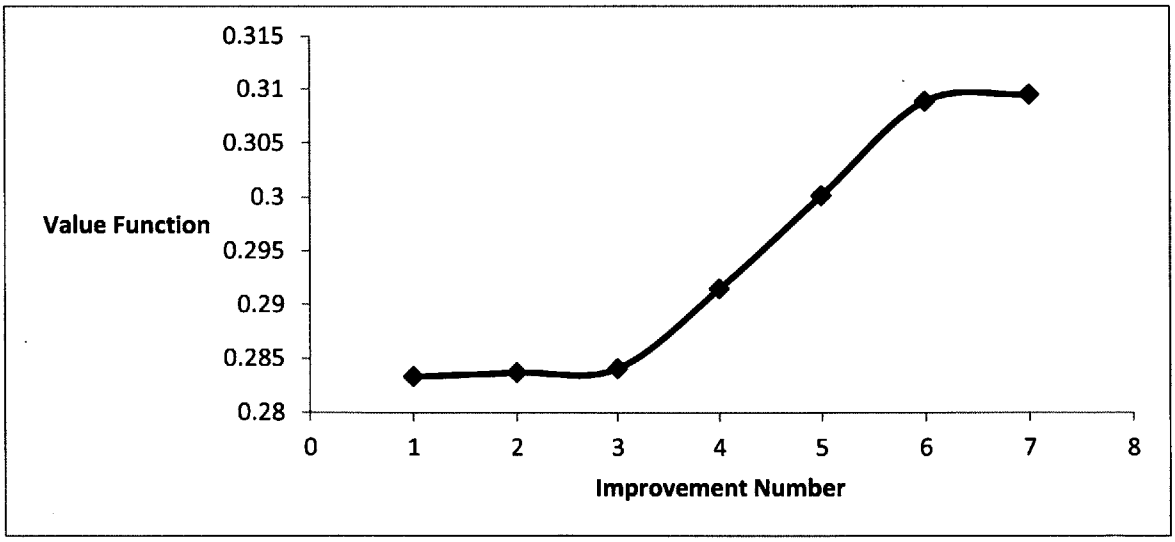

Figure 7 – Plot of value function against newest local maximum found.

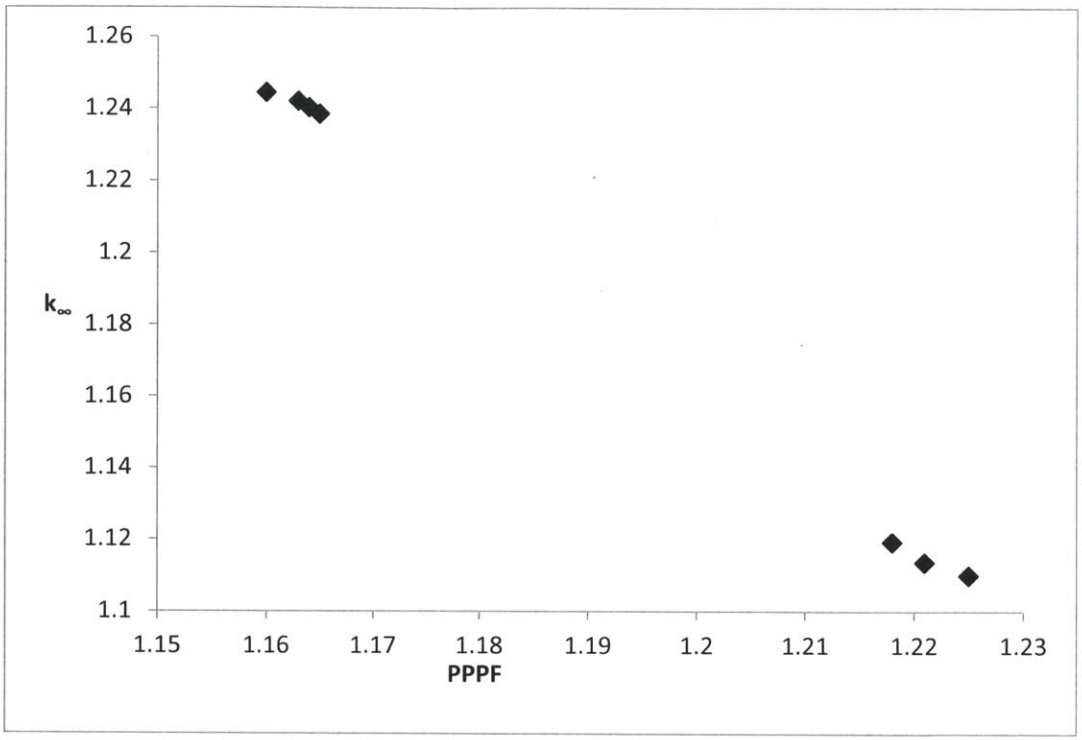

Figure 8 – PPPF vs. k_∞.

In the case where uranium density was varied the system converged in quite a different manner. The final result is given below in Figure 9.

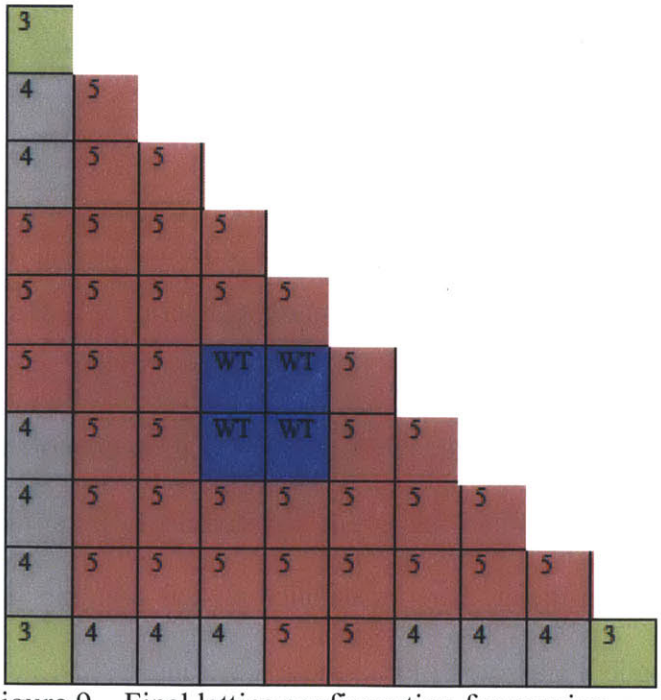

Figure 9 – Final lattice configuration for varying uranium simulation.

Here the highest enrichment is pin #5 and the lowest is pin #1; the intermediates are populated as previously noted. This is an expected result, since again there is a symmetric bundle with the lowest reactivity pins along the outside of a homogenous lattice. Figure 10 shows that the uranium variable case started out much further from its final solution, however GEDBS closed in on its solution quickly and then spent several thousand iterations hovering in the proximity of the purported optimum. Note that unlike Figure 7, Figure 10 shows all iterations of the algorithm, not just the local maximums. It took seven sweeps of the GEDBS algorithm to converge on the solution shown in Figure 9.

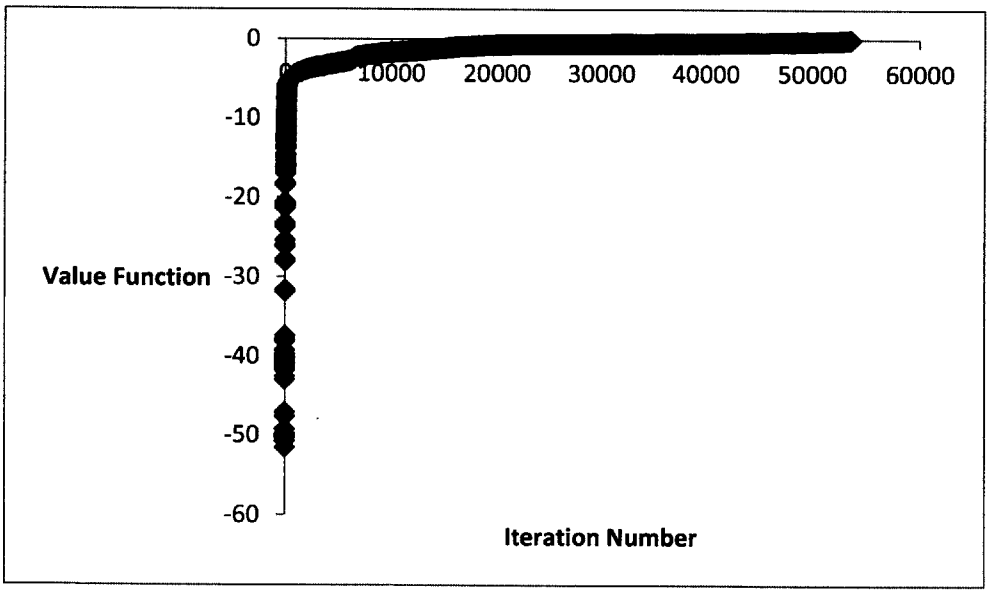

Figure 10 – Value function against iteration number of GEDBS.

Figure 11 shows that the distribution of PPPF against k_∞ is much more dispersed than in the variable gadolinium scenario. The PPPF starts out extraordinarily large due to the starting configuration, the pins on the perimeter have essentially all the uranium and therefore are creating all the power, resulting in a massive PPPF in those locations.

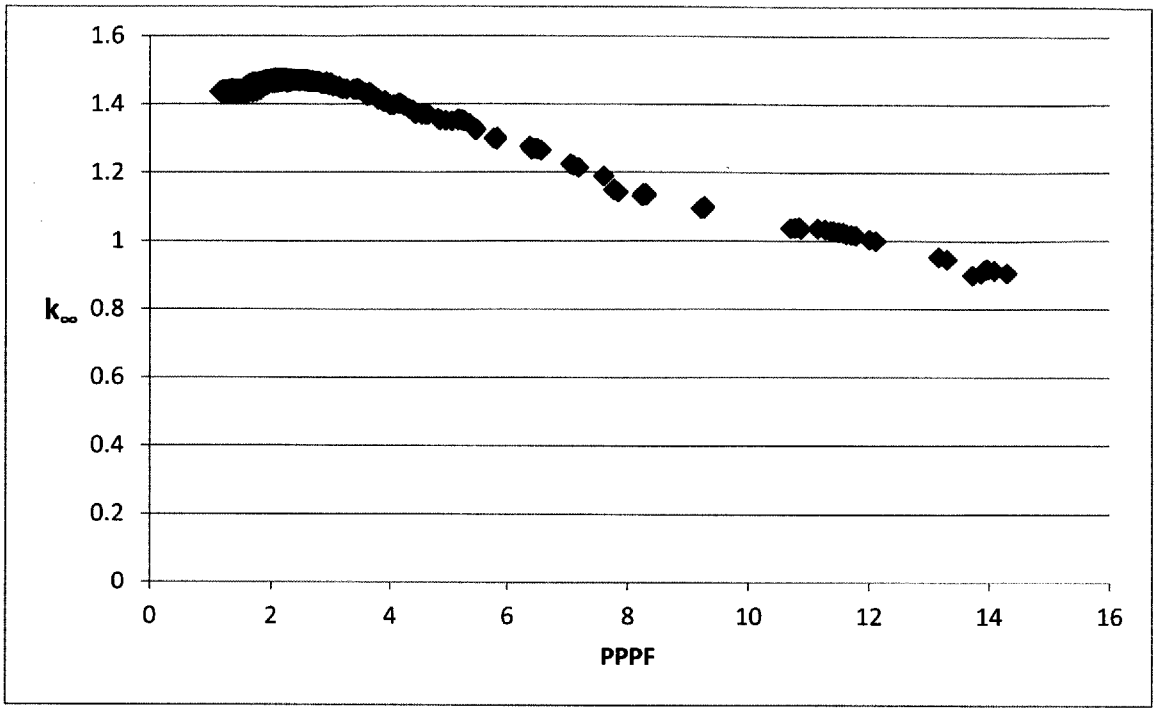

Figure 11 - $\boldsymbol{K}_\infty$ vs. PPPF.

While both gadolinium variable and uranium variable cases reached different numerical values, the qualitative nature of the respective solutions are quite similar. The starting lattice of all 1s with 5s at the corners was used for both; this meant that the uranium variable sequence had close to the worst possible starting position. This is consistent with Figure 10, where the value function starts an extra order of magnitude away from the solution compared to Figure 7.

Table 1 – Summary of final convergence of GEDBS algorithm.

	k_∞	PPPF	Value function
Gadolinium Variable	1.11927	1.218	0.30946
Uranium Variable	1.4351	1.154	-0.066

To compare these results to the more traditional SA and GA schemes, another python script was written primarily by Jeremy Roberts implementing the SA scheme for changing

gadolinium and uranium [19]. This SA python code managed to effectively find an optimum at $k_\infty = 1.09$, PPPF = 1.19 giving a final value function of 0.48. This value function is higher than either of the results seen by GEDBS, and both k_∞ and PPPF are within their respective limits. It should be noted that the SA code did solve for two variables at once and was applied to more than just BOC conditions, however the capacity to alter both k_∞ and PPPF simultaneously lent SA more flexibility within the value function. Nonetheless the results of SA, while applied slightly differently, still outperformed those of GEDBS.

After realizing these results it was not necessary to test GEDBS against a GA, since GA can be made to imitate SA results as discussed in Section 3.2.2. This would only have become a comparison of SA to GA, which is not the directive of this paper. Since the GA results can be made to approximate the results of SA, the question of whether GA is superior in this application to GEDBS could simply devolve into how much the GA operators were akin to the SA cooling schedule.

6. Conclusions

Given the superior results of the SA/GA over GEDBS, there are clearly some improvements that need to be made to the GEDBS model to make it more competitive with the well-established optimization methods. A primary restraint to the GEDBS method is the runtime dictated by Equation 6. Due to the exhaustive nature of GEDBS there are an extreme number of iterations that need to be run.

This problem could be alleviated by advances in computing power. A large number of sweeps at a minor amount of time per sweep would lead to a reduced runtime. Furthermore,

with a runtime on the order of minutes rather than days troubleshooting and improvement of the GEDBS algorithm would also be greatly expedited.

If GEDBS were working on a certain type of problem for an extended period, or was always applied to a specific scenario more metaheuristic modifications could be made. As shown by the differences between Figure 10 and Figure 7 the starting position can make a very big difference in the number of sweeps GEDBS needs to perform to come to convergence. If a metaheuristic method of selecting a generally beneficial starting position were applied, this could save a significant number of iterations. For example, given any particular lattice optimization problem the metaheuristic could preemptively place the higher reactivity pins starting in the center and occupy the corner positions with lower reactivity pins. This would not help the GEDBS algorithm perform any better in a mathematically significant manner, only reduce the necessary runtime of its implementation to certain cases.

For real lattice design for use in an operational reactor there will be many more than two variables to alter. As previously discussed, adding new variables to examine exponentially increases the search space. There are dozens of variables that are federally mandated to be tracked in industry. Therefore it can be concluded that without drastic improvements to computing power the GEDBS algorithm is not ready for industry application.

7. References

[1] W. E. Combs and J. Andrews, "Combinatorial Rule Explosion Eliminated by a Fuzzy Rule Configuration," *IEEE Transactions on Fuzzy Systems,* pp. 1-11, 1998.

[2] R. Groner, M. Groner and W. F. Bischof, Methods of Heuristics, Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.

[3] R. Marti and G. Reinelt, The Linear Ordering Problem: Exact and Heuristic Methods in Combinatorial Optimization, Springer, 2011.

[4] Kilpatrick, Gelatt and Vecchi, "Optimization by Simulated Annealing," *Science,* pp. 671-680, 13 May 1983.

[5] H. Szu and R. Hartlet, "Fast Simulated Annealing," *Physics Letters,* pp. 157-162, 8 June 1987.

[6] S. Rajasekaran, "On the Convergence Time of Simulated Annealing," Department of Computer & Information Science at University of Pennslyvania, Philadelphia, PA, 1990.

[7] F. Busetti, "Simulated Annealing Overview," 2003. [Online]. Available: www.geocities.com/francorbusetti/saweb.pdf .

[8] D. Bertsimas and J. Tsitsiklis, "Simulated Annealing," *Statisical Science,* vol. 8, no. 1, pp. 10-15, 1993.

[9] H. Gerber, "First One Hundred Zeros of J(x) Accurate to 19 Significant Figures," *Mathematics of Computation,* pp. 319-322, Apr 1964.

[10] M. Melanie, An Introduction to Genetic Algorithms, Cambridge, MA: Bradford Book of The MIT Press, 1996.

[11] W. Weishui and X. Chen, "Convergence theorem of genetic algorithm," *IEEE International Conference on Systems, Man, and Cybernetics,* pp. 1676-1681, 1996.

[12] D. Thierens and D. Goldberg, "Convergence Models of Genetic Algorithm Selection Schemes," *PPSN III,* pp. 119-129, 1994.

[13] S. Dasgupta, C. Papadimitriuo and U. Vazirani, Algorithms, Californina: McGraw-Hill, 2007.

[14] K. Smith, Interviewee, *Initial Interview Regarding Optimization Problems.* [Interview]. 16 October 2012.

[15] Korea Atomic Energy Research Institute, "Table of Nuclides," 2000. [Online]. Available: http://atom.kaeri.re.kr/.

[16] R. A. Knief, Nuclear Engineering: Theory and Technology of Commerical Nuclear Power, Mechanicsburg, PA: American Nuclear Society, Inc, 2008.

[17] E. E. Lewis, Nuclear Reactor Physics, Oxford, UK: Academic Press, 2008.

[18] J. Rhodes, K. Smith and D. Lee, "CASMO-5 Developments and Applications," in *ANS Topical Meeting on Reactor Physics*, Vancouver, BC, Canada, 2006.

[19] J. Roberts, *python pyOpt.py,* Cambridge, MA, 2012.

[20] Z. Gong, K. Wang and D. Yao, "An Interval Bound Algorithm of optimizing reactor core loading pattern by using reactivity interval schema," *Annals of Nuclear Energy,* vol. 38, no. 12, pp. 2787-2796, December 2011.

[21] M. S. Kazimi and E. Pilat, *CASMO-4,* Cambridge, MA: 22.351 Systems Analysis of the Nuclear Fuel Cycle.

8. Appendix A

8.1. Python omnicont.py code

```
import os
import numpy as np
import random

class palette(object):
    def __init__(self):
        #maximum enrichment of uranium
        self.uEnrich = 5.0
        self.uDensity = 10.2
        #min enrichment, global variables as they are needed throughout the palette
        self.minGd = 0
        self.maxGd = 10
      self.deltaGd = 2.5
      self.pinPal = []
      self.pins = []
        #For reasons unknown, there is a floating point error that necessitates multiplying everything
by 10 to use integers.
        global pinPalette
        pinPalette = []
      def makePalette(self): # makes a palette of possible pin types
            gd = self.minGd
            while gd<=self.maxGd:
                  self.pinPal.append((self.uEnrich, gd))
                  gd+=self.deltaGd
            i = 0
            for tup in self.pinPal:
                  self.pins.append("FUE " + str(i+1) + " " + str(self.uDensity) + "/" + str(tup[0]) +"
64016="+str(tup[1]) )
                  i+=1
            print self.pins
            return self.pins


class Lattice:
      #def __init__(self, burnup = 0, oldArray =
[1,2,4,3,5,4,3,4,4,5,3,4,4,4,4,3,4,5,2,2,4,3,4,4,2,2,5,4,3,5,4,4,5,4,4,4,2,4,5,4,4,5,4,5,4,1,2,4,4,4,4,4,4,2,1]
):
      def __init__(self, burnup = 0, oldArray =
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,1,1,1,1,2,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
, fedPalette = []):
            self.upTemp = "blank"
            self.downTemp = ""
            self.octant = []
        self.oldArray = oldArray #optionally accepts an old array
        self.burnup = burnup
        self.pinPalette = fedPalette
```

```
    def upperTemplate(self):
            self.upTemp = "TTL * base_case\n"                +\
            "TFU=900 TMO=560 VOI=40\n"              +\
            "PDE 54 'KWL'\n\n"                +\
            "BWR 10 1.3 13.4 0.19 0.71 0.72 1.33/0.3048 3.8928\n"+\
            "PIN 1 0.4400 0.4470 0.5100\n"          +\
            "PIN 2 1.1700 1.2400 /'MOD' 'BOX' //4\n"      +\
            "LPI\n"                    +\
            "1\n"                    +\
            "1 1\n"                    +\
            "1 1 1\n"                    +\
            "1 1 1 1\n"                    +\
            "1 1 1 1 1\n"                    +\
            "1 1 1 2 2 1\n"                    +\
            "1 1 1 2 2 1 1\n"                    +\
            "1 1 1 1 1 1 1 1\n"                    +\
            "1 1 1 1 1 1 1 1 1\n"                    +\
            "1 1 1 1 1 1 1 1 1 1\n\n"
            return self.upTemp

    def lowerTemplate(self): #create the LFU
        if self.pinPalette == []:
            p = palette()
            self.pinPalette = p.makePalette()
            lineNum = 1
            colNum = 1
            i = 0
            j = 0
            while i < len(self.pinPalette): #create list of fuel pins
            self.downTemp = self.downTemp + self.pinPalette[i] + " \n"
            i+=1
            if (len(self.pinPalette)== i): #add an extra line between FUE and the pins
                self.downTemp = self.downTemp + " \nLFU \n"

            while j < len(self.oldArray): #create triangle of pin types
            self.downTemp = self.downTemp + str(self.oldArray[j]) + " "
            if colNum==lineNum: #steps to next line if the length of the triangle equals the width
                self.downTemp = self.downTemp + " \n"
                colNum=0
                lineNum+=1
            colNum+=1
            j+=1
            if len(self.oldArray) == j:
                self.downTemp += "\nDEP -" + str(self.burnup) + "\n\nSTA\nEND"
        return self.downTemp, self.pinPalette


class createInput:
    def __init__(self, inArray, permPalette=[]):
        self.inArray = inArray
        self.pinPalette = permPalette
        self.output = ""
        self.writeFile = "BWRin"
        self.writeOut(self.output)

    def create(self): #feed arrays to lattice or create from empty if not applicable.
```

```
            #The upper lattice never changes, so no need to keep generating it.

            lat = Lattice(0,self.inArray, self.pinPalette)
            top = lat.upperTemplate()
            bottom = lat.lowerTemplate()
            self.output = top + bottom[0] #join the top and bottom halves to create a whole input file
            self.pinPalette = bottom[1]

      def writeOut(self, output):
            self.create()
            self.file = open(self.writeFile + '.inp','w')
            self.file.write(self.output)
            self.file.close()
      def getPalette(self):
            return self.pinPalette

class subJob:
      def __init__(self, file = "blank"):
            self.file = file
            if self.file == "blank" :
                  c = createInput()
                  self.file = 'BWRin'
            self.runCASMO()

      def runCASMO(self):
            # remove old copies of out and cax
            os.system('rm -Rf '+self.file+'.out '+self.file+'.log '+self.file+'.cax outout ')
            # run casmo
            #print " running: casmo4e " +file + ".inp\n"
            os.system('casmo4e '+self.file+'.inp > output')
            # remove outputs and cax for not, not needed

class readCasmo:
      def __init__(self, file = "output"):
            self.file = file
            self.burnup = [] #create variables for all data we want to extract
            self.kinf = []
            self.pppf = []
            #self.read()

      def read(self): #read the output file and return the desired variables
            f = open(self.file, 'r')
            lines = f.readlines()
            startScan = 45
            words = lines[startScan].split()
            #print words
            self.burnup.append(float(words[6]))
            self.kinf.append(float(words[7]))
            self.pppf.append(float(words[10]))
            return self.burnup,self.kinf,self.pppf

class runSim:
      def
__init__(self,array=[5,5,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
5,1,1,1,1,1,1,1,1,5],slowPos=0,slowVal=5,fastPos=1,fastVal=5,bestArray=[5,5,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,0,0,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,5,1,1,1,1,1,1,1,1,5],best=-1000):
```

```
            #self.burnUp = 0
            self.kinf = 0
            self.kinfBest = 0
            self.best = [] #store best array
            self.pppf = 0
            self.kinfMax = 1.13
            self.kinfMin = 1
            self.pppfMax = 1.35
            self.vialbe = False
            self.waterTube = [18,19,24,25] # water tubes dont change
            self.array = array
            self.permPalette = []
            self.optimize(array,slowPos,slowVal,fastPos,fastVal,bestArray,best)
        def isViable(self, kinf,pppf):
            if (kinf > self.kinfMin) and (kinf < self.kinfMax) and (pppf < self.pppfMax):
                self.viable = True
            else: self.viable = False
        def optimize(self,array,slowPos,slowVal,fastPos,fastVal,bestArray,best):
            #slowPos = 0 #Position of slow loop - i
            #fastPos = 0 #Position of fast loop - j
            c=createInput(self.array,self.permPalette) #create the input file
            self.permPalette = c.getPalette()
            subJob('BWRin')
            r = readCasmo()
            results=r.read()
            self.best = bestArray
            self.bestScore = best
            self.isViable(results[1],results[2])
            while (fastPos in self.waterTube) or (slowPos==fastPos): fastPos+=1 #avoid changing water
tubes or the other variable pin.
            while slowPos <len(self.array): #Slow position loop
                if slowVal == 0: slowVal = 5 #Value inserted into slow loop
                while slowPos in self.waterTube: slowPos+=1 # don't change watertubes
                slowChanged = False #track if large loop has made a better iteration
                while slowVal > 0 and slowVal <= len(self.permPalette):
                    self.array[slowPos] = slowVal
                    if fastPos == 55: fastPos =0
                    while fastPos < len(self.array): #Fast Position loop
                        fastChanged = False
                        #tempj = self.array[fastPos]
                        fastVal = len(self.permPalette)
                        while fastVal <= len(self.permPalette) and fastVal>0: #Fast value loop
                            self.array[fastPos] = fastVal #randomly change second variable pin
                            self.file = open('record.txt','a')
                            self.file.write( str(slowPos) +', '+str(slowVal)+', '+str(fastPos)+',
'+str(fastVal)+', '+ str(self.array) + str(4*(1.3-results[2][0])+2*(1.11-results[1][0]))+"  "+str(results[2][0])+"
"+str(results[1][0])+ '\n')
                            print slowPos, slowVal,fastPos,fastVal
                            if self.array != self.best:
                                c=createInput(self.array,self.permPalette)
                                subJob('BWRin')
                                r = readCasmo()
                                results=r.read()
                                #compare the score of different groups and choose the 'better' one
                                if (4*(1.3-results[2][0])+2*(1.11-results[1][0])) > self.bestScore:
                                    self.bestScore = (4*(1.3-results[2][0])+2*(1.11-results[1][0]))
```

```
                        self.best=self.array[:]
                        fastChanged = True
                        slowChanged = True
                        self.file.write( "BEST:" + str(self.best)+str(self.bestScore)+ '\n')
                    elif (4*(1.3-results[2][0])+2*(1.11-results[1][0])) == self.bestScore:
                        self.isViable(results[1],results[2])
                        if self.isViable:
                            self.bestScore = (4*(1.3-results[2][0])+2*(1.11-results[1][0]))
                            self.best = self.array[:]
                            fastChanged = True
                            slowChanged = True
                fastVal-=1
            if not fastChanged: self.array[fastPos] = self.best[fastPos]
            fastPos+=1
            while (fastPos in self.waterTube) or (slowPos==fastPos): fastPos+=1 #avoid
changing water tubes or the other variable pin.
            slowVal -=1
        if not slowChanged: self.array[slowPos] = self.best[slowPos]
        slowPos+=1
        print 'end slow loop'
        #print self.best

    print self.best

class textreader:
    def __init__(self, file = "record.txt"):
        self.file = file
        self.pickup()
    def pickup(self):
        array = ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p',
'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p']
        bestArray =[]
        best = 0
            f = open(self.file, 'r')
            lines = f.readlines()
            if len(lines)==2:
            del lines[1]
            for i in range(0,len(lines),1):
            if lines[i][0:4] =="BEST":
                bestArray = lines[i][5:170]
                best =  float(lines[i][170:len(lines[i])])
        lastLine = lines[len(lines)-1]
        j =0
        k = 0
        bookmarks = [0,0,0,0]
        while lastLine[j] != '[':
            if (lastLine[j]) == '0' or (lastLine[j]) == '1'or (lastLine[j]) == '2'or (lastLine[j]) == '3'or
(lastLine[j]) == '4'or (lastLine[j]) == '5'or (lastLine[j]) == '6' or (lastLine[j]) == '7' or (lastLine[j]) == '8' or
(lastLine[j]) == '9': bookmarks[k]=str(bookmarks[k]) + lastLine[j]
            elif lastLine[j] == ',':
                bookmarks[k] = int(bookmarks[k])
                k+=1
            j+=1
        k=0
        while lastLine[j] != ']':
```

```
                if (lastLine[j]) == '0' or (lastLine[j]) == '1'or (lastLine[j]) == '2'or (lastLine[j]) == '3'or
(lastLine[j]) == '4'or (lastLine[j]) == '5'or (lastLine[j]) == '6' or (lastLine[j]) == '7' or (lastLine[j]) == '8' or
(lastLine[j]) == '9': array[k]=str(array[k]) + lastLine[j]
                elif lastLine[j] == ',':
                        array[k] = array[k].replace('p',")
                        array[k+1] = array[k+1].replace('p',")
                        k+=1
                j+=1
        for q in range(0,len(array),1):
                array[q] = int(array[q])
        bestArray = eval(bestArray)
        return bookmarks[0],bookmarks[1],bookmarks[2], bookmarks[3],array,bestArray,best


class overseer:
    def __init__(self, cont = 'yes'):
        if cont =='yes':self.cont = True
        else:self.cont = False
        self.run()
    def run(self):
        if self.cont:
            t = textreader()
            SD = t.pickup()[:]
            array =SD[4]
            slowPos = SD[0]
            slowVal=SD[1]
            fastPos=SD[2]
            fastVal = SD[3]
            bestArray =SD[5]
            best = SD[6]
            runSim(array,slowPos,slowVal,fastPos,fastVal,bestArray,best)
        else:
            if os.path.exists('record.txt'):
                os.remove('record.txt')
                file('record.txt','w').close()
            runSim()


overseer('yes')
```

8.2. Sample CASMO Submission

```
TTL * base_case
TFU=900 TMO=560 VOI=40
PDE 54 'KWL'

BWR 10 1.3 13.4 0.19 0.71 0.72 1.33/0.3048 3.8928
PIN 1 0.4400 0.4470 0.5100
PIN 2 1.1700 1.2400 /'MOD' 'BOX' //4
LPI
1
1 1
1 1 1
1 1 1 1
1 1 1 1 1
1 1 1 2 2 1
1 1 1 2 2 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1

FUE 1 10.2/5.0 64016=0
FUE 2 10.2/5.0 64016=2.5
FUE 3 10.2/5.0 64016=5.0
FUE 4 10.2/5.0 64016=7.5
FUE 5 10.2/5.0 64016=10.0

LFU
2
1 1
1 1 1
1 1 1 1
5 1 1 1 1
1 1 1 0 0 1
1 1 1 0 0 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
2 3 1 1 5 1 1 1 1 4

DEP -0

STA
END
```