Rochester Institute of Technology

Master’s Project Report

Michael Adams

A Parallel Framework for NP Combinatorial Optimization Problems

Department of Computer Science

Chairperson: Alan Kaminsky

Reader: Stanisław Radziszowski

Observer: Leon Reznik
Abstract – NP-hard problems have no known polynomial time algorithms for finding the optimal solution. Many NP-hard problems, however, have useful similarities between them in how they are solved. This paper proposes a parallel framework developed in Java that abstracts these useful similarities for solving NP-hard problems. The framework provides a platform for solving these problems in parallel using multiple solution strategies, without the user needing to implement the strategy or the parallelism. The framework, as well as how to use it, are discussed here in detail. The development process showed that implementation time without using the framework is about twice as long as with using the framework. In addition, running the solution strategies in parallel showed a clear reduction in execution time or an increase in solution quality depending on the strategy.

I. Introduction

Function optimization is a very important topic in Computer Science, and encompasses a wide array of problems and solution strategies. Some of these optimization problems fall into the NP domain, meaning that the only known algorithms for finding the optimal solution take exponential time. That is, the only way to guarantee the optimal solution is to examine all possible solutions. This is only feasible for problems of small sizes. Heuristics can be used to approximate an optimal solution in a significantly shorter period of time. The design and implementation of these algorithms is important for finding a high quality solution to these problems in a reasonable amount of time. Some examples of these heuristics include hill climbing, simulated annealing, genetic algorithms, and tabu search, but there are many, many more in existence [1]. Many of these algorithms have a fairly generic formula for solving the problem. For example, hill climbing starts with a random solution, iteratively mutates that solution, and transitions to any mutations that are a higher quality solution. The algorithm searches to a local optimum, at which point further mutation (at least minor mutations) reduces solution quality.

Many of these optimization problems conform to these general concepts. That is, they must have a representation of a solution, they must have a way to evaluate the quality of that solution, and they must have a way of changing or mutating a solution. This presents an opportunity to exploit this generality. Taking the case of hill climbing again, the actual algorithm does not need to know how to represent a solution or how a solution is represented, nor how to mutate or evaluate a solution. Rather, these concepts can be abstracted, so that the algorithm is working with these concepts, and the problem specific details can be added later. The algorithm can ask other entities to provide it with a random solution, provide an evaluation for a solution, compare two evaluations, and mutate and return a new solution. Then, these entities can be created for a particular problem, and used with the hill climbing algorithm.

This is the foundation of the framework proposed in this paper. The goal was to create a framework that abstracted these concepts and required only problem specific details to be implemented to solve the problem. The framework supports three solution strategies – brute force, hill climbing, and simulated annealing. In addition, because each of these (and many other optimization algorithms) support parallel execution, the framework would be designed to allow sequential or parallel execution. The main focus was shared memory multiprocessor machines, but it can be built upon to allow distributed parallelism as well. Three problems were chosen to demonstrate the effectiveness of the framework – 0-1 knapsack, maximum satisfiability, and minimum vertex cover. Each problem was implemented both with and without the framework.
Section II presents my hypotheses regarding the framework. Section III looks at some related frameworks. Section IV presents each solution strategy, as well as how they operate both sequentially and in parallel. Section V provides a description of each of the example problems. Section VI explains the framework, providing a class diagram and class descriptions for the framework, as well as class descriptions for each of the problem implementations. Section VII discusses the experiments that were run and the machine they were run on. Section VIII presents and analysis the results of the experiments. Section IX provides a developer manual, while section X provides a user manual (including how to run each of the example problems). Section XI lays out potential future work. Section XII presents the lessons learned while completing this project. Lastly, section XIII provides a conclusion.

II. Hypothesis

I hypothesize that the framework I am proposing reduces the time required to implement an NP optimization problem in parallel. I also hypothesize that the framework’s parallelism is beneficial to each search strategy in comparison to a sequential version. For the brute force strategy, I hypothesize that this benefit is a speedup over a sequential execution. For hill climbing, I primarily hypothesize that this benefit is a speedup over a sequential execution. In addition, I hypothesize that hill climbing also provides a higher quality solution than a sequential execution. For simulated annealing, I hypothesize that a higher quality solution is found in the same execution time.

III. Related Work

Mendes et al. proposed a similar framework in 2001 [2]. There are some similarities between them, but I believe that my framework offers more flexibility. Both frameworks are designed to generalize some of the components of NP problems into abstract classes, both are written in Java, and both support multiple solution strategies. Their framework, however, does not support parallel execution and seems more rigid than what I propose here. The three solution strategies they implemented are: genetic algorithm, memetic algorithm, and multiple start. The problems they’ve implemented are: single and parallel machine scheduling, permutational flowshop scheduling, grid matrix layout, and to show that it is useful for more than just the NP domain, continuous function optimization.

The rigidity of their framework was interpreted from section 4 of their paper. The authors mention that the user must define a representation of his problem, e.g. how a particular solution will be represented. They say that valid representations can be arrays of bits, integers, floats, doubles, chars, or mixes of those types. My framework imposes no such restriction. The user may implement their representation with whatever data structures they deem necessary. The authors also mention a method to evaluate a solution, which must return an integer, float, or double. Again, my framework does not impose such a restriction, and an evaluation can also be represented however the user desires.

The main advantage that their framework has over mine is a graphical user interface. Most importantly, it allows the user to see how the algorithm is progressing by displaying better solutions as it finds them, along with how long it took to find each solution. However, this feature leans more toward personal preference of whether or not the user wants a GUI.
Coelho et al. proposed a framework very similar to the one proposed here, but written in C++. OptFrame [3], as they call it, has much of the same functionality and flexibility as the framework I’m proposing, with the addition of distributed memory parallelism. Although their framework does not include a brute force method, it does include hill climbing and simulated annealing. As the authors described it, they’ve implemented very simple versions of the solution strategies, allowing the user to modify it into a smarter version for their specific problem.

OptFrame appears to have a similar design and structure to my framework, with the main difference being implementation language and terminology, and their support for distributed memory parallelism. They also mention a few other frameworks, but these seem much more focused on researching the metaheuristic algorithms themselves rather than providing an easy to use framework for users.

IV. Solution Strategies

The proposed framework supports three different solution strategies: brute force, hill climbing, and simulated annealing. Each algorithm supports sequential and parallel execution.

IV.A – Brute Force

The brute force solution strategy is an exhaustive strategy. It simply searches for the optimal solution by attempting all possible solutions. This guarantees to find the best possible solution, but takes exponential time to do so. As such, it is not suitable for large problem sizes.

Sequential

The sequential brute force simply attempts all possible problem solutions and keeps track of the best one. The possible solutions to iterate over are controlled by the ExhaustiveGenerator class. This class must know the number of possible solutions, and must be able to generate every one of them. This is all processed by a single thread.

Parallel

Figure 1 Parallel Brute Force. The work is split amongst the available threads. Each thread only needs to iterate over #solutions / #threads of the solutions.
The parallel implementation of brute force performs the same, but splits the work amongst all available threads. That is, each thread gets \((\text{solutions} / \text{threads})\) solutions to evaluate, as shown in Figure 1. Each thread has its own ExhaustiveGenerator object to generate its portion of the solutions. Once all threads have finished with their portion of solutions, each thread reduces the best solution it found locally. Thus, the parallel implementation should result in the same solution, but with the benefit of a speedup of approximately \(\text{threads}\).

**IV.B – Hill Climbing**

The hill climbing solution strategy randomly traverses the solution space until it reaches a local optimum [1]. The current best solution is in some way mutated, and if this mutation is a better solution, then it becomes the current best solution. Only solutions that provide an improvement are transitioned to, so it cannot escape a local optimum.

**Sequential**

The sequential hill climbing algorithm starts from a random point in the solution space. Then, for a set number of iterations \(I\), it will perform \(M\) mutations to the current best solution, transitioning to any mutations that improve the solution. The way mutations are made depends on how the MutationGenerator class was implemented. For the three implemented problems, mutations are made by simply flipping a value in the assignment (i.e., include an excluded item or vice versa (knapsack), include an excluded vertex or vice versa (minimum vertex cover), and make a false literal true or vice versa (maximum satisfiability). After all iterations have been performed, the current best solution is the final solution.

**Parallel**

The parallel implementation of hill climbing also starts from a random point in the solution space. Each thread starts from this same starting solution, but each takes a different path through the solution space. Each thread performs the same number of iterations \(I\), but performs only \((M / \text{threads})\) mutations, as shown in Figure 2. Each thread has its own MutationGenerator, each of which has a different random number generator, and therefore generates different mutations from the other threads. The hope is that one of the threads takes a luckier path through the solution space, finishing at a higher quality solution than only one thread may have been able to find. Also, because each thread performs a reduced number of mutations, a speedup is also seen.

**IV.C – Simulated Annealing**

The simulated annealing solution strategy attempts to simulate the process of annealing in metallurgy [1],[4],[5]. Annealing is the process of gradually cooling a metal, allowing the molecules to slowly settle into a crystalline structure of least energy. In other words, the molecules (solutions) are highly volatile at first, but over time become less volatile until settling into a local optimum. This volatility is simulated by allowing transitions to states of higher energy (i.e. lower quality solutions). The purpose of this is to allow the algorithm to escape local optima near the beginning of execution (something that standard hill climbing cannot do), transitioning to other areas of the solution space. Lower energy states (higher quality solutions) are always transitioned
Figure 2 Parallel Hill Climbing. Rather than one thread doing all mutations, they are split amongst the available threads. Each has a separate random mutator, generating different mutants.

to, whereas higher energy states (lower quality solutions) are transitioned to based on an acceptance probability. This acceptance probability is dependent on the difference in energy of the two states (i.e. less likely to move to a significantly higher energy, more likely to move to a slightly higher energy), as well as the current temperature of the system, which decreases periodically (lower temperatures result in lower acceptance probabilities). The rate of temperature reduction is defined by the cooling schedule. Once the temperature becomes sufficiently low, the algorithm essentially becomes a hill climb.
**Sequential**

The sequential implementation of simulated annealing simply performs the simulated annealing algorithm on a single thread. A random solution is chosen as the starting best solution. For each temperature step, a number of trials $T$ will be performed. During each trial, the current best solution is mutated (again, dependent on the implementation of MutationGenerator), and potentially transitioned to. There is a success limit $S$. If during the $T$ trials there are $S$ successful transitions (both automatically moving to a higher quality solutions and passing the acceptance probability and moving to lower quality solutions), then the temperature is reduced and the next temperature step is started immediately. If during the $T$ trials there are no successful transitions, then the algorithm terminates and the current best solution is the final solution. Alternatively, after all temperature steps have been completed, the algorithm terminates and the current best solution is the final solution.

**Parallel**

The parallel implementation of simulated annealing performs a unique run of the algorithm on each available thread, as shown in Figure 3. That is, each thread will perform the algorithm as described in the sequential description, but each thread will have a different MutationGenerator, each of which has a unique random number generator. Each thread starts at a unique starting
solution, and each takes a unique path through the solution space. Thus, the parallel implementation will not run any faster, but the hope is that additional threads may get a luckier starting solution and path through the solution space than just one thread.

V. Problem Descriptions

V.A – 0-1 Knapsack

The 0-1 knapsack problem is an NP-complete combinatorial optimization problem [6]. The problem is as follows. Given a list $Q$ of $N$ objects, each of which has a weight $W_i$ and value $V_i$, and also given a “bag” with a weight limit $M$, the goal is to choose a series of 0-1 values for $X_i$, such that

$$
\sum_{i=0}^{N-1} X_i \cdot W_i < M
$$

AND

$$
\sum_{i=0}^{N-1} X_i \cdot V_i \text{ is maximized}
$$

where $X_i$ is either 0 or 1 (hence 0-1 knapsack problem). Zero therefore indicates exclusion of the object and 1 indicates inclusion.

This is much more easily understood through a real world example. Imagine a thief breaks into a house to burgle. He brings with him a burlap sack which can only hold a certain amount of weight before the seams start to tear. The thief’s goal is to fill his burlap sack with the lightest, but most valuable objects. For example, a diamond ring would have a very high value and a very low weight. In contrast, a desk would have relatively low value and a very high weight. The problem is to find the optimal selection of items, such that the thief takes the most valuable items without tearing his burlap sack. Figure 4 depicts an example scenario of this problem.

Figure 4 Illustration of 0-1 Knapsack problem. Optimal selection would be all but the green box$^1$.

---

$^1$ Image from: http://en.wikipedia.org/wiki/Knapsack_problem
V.B – Maximum Satisfiability

The maximum satisfiability problem is also an NP-hard combinatorial optimization problem [7]. Given a Boolean expression \( E \) in conjunctive normal form, find the assignment of literals that maximizes the number of satisfied clauses. The assignment of literals simply refers to assigning true or false to each independent literal. In addition, a solution to the maximum satisfiability problem is also a solution to the satisfiability problem, which simply checks if the Boolean expression can be satisfied by a particular assignment of literals. An example of a conjunctive normal form Boolean expression and an assignment solving maximum satisfiability can be seen in Figure 5.

\[
E = (x_1 \lor \neg x_2 \lor \neg x_3) \land (x_1 \lor x_2 \lor x_4)
\]

Figure 5 Example of a 3CNF Boolean expression. Here, having an assignment where \( x_1 \) is true will satisfy the maximum number of clauses (2 clauses).

V.C – Minimum Vertex Cover

The minimum vertex cover problem is an NP-hard combinatorial optimization problem. Given a graph \( G \), with a set of vertices \( V \), a graph cover is any subset of \( V \) such that all edges in \( G \) are incident to at least one vertex in the subset. Thus, a minimum vertex cover is the smallest subset of vertices such that all edges in \( G \) are incident to at least one vertex in the subset. For example, the graphs in Figure 6a show covers, while the graphs in Figure 6b show minimum vertex covers.

![Figure 6](Image from: http://en.wikipedia.org/wiki/Minimum_vertex_cover)

This can also be represented by a real-world use case. Imagine a building with intersecting hallways (thus, in Figure 6, the edges would be hallways, and the vertices would be intersections of hallways). The problem could then be expressed as finding the minimum number of security cameras such that every hallway is monitored. That is, a security camera would be placed at the intersection of 2 or more hallways, and can be used to monitor any incident hallways.

VI. Framework Overview

Having discussed the above solution strategies and example problems, it can be seen that certain aspects related to using a solution strategy have strong similarities between the problems. These similarities can be abstracted and exploited to allow a generic framework for solving such a problem. The problem-specific details can then be implemented to allow for solving that

---

\(^2\) Image from: http://en.wikipedia.org/wiki/Minimum_vertex_cover
particular problem. This is the foundation of the proposed framework. The framework was developed in Java, using Professor Kaminsky’s Parallel Java\textsuperscript{3} library [8]. Each of the framework classes will be presented in detail, including how each factors in to abstracting these problem similarities.

\textit{VI.A – Framework}

Figure 7 depicts the class diagram for the framework. There are parallel versions of each of the solution strategies, called BruteForceSMP, HillClimbingSMP, and SimulatedAnnealingSMP. The only difference is that they take arrays of assignment generators, cooling schedules, and random number generators in their constructors.

\textbf{Assignment}

This abstract class represents the concept of an assignment for a combinatorial optimization problem. That is, a potential solution for a combinatorial optimization problem involves a particular combination of entities. In addition, each combination needs to be able to be evaluated (i.e. assess the solution quality of each combination). In this class, users must implement a method to evaluate the assignment, as well as a copy method (for parallel reduction). Detailed subclass examples are provided for each problem in the following subsections.

\textbf{Evaluation}

This abstract class represents the concept of an evaluation of an assignment. That is, Assignment’s evaluate method must produce an object of this type. The evaluation holds information regarding the solution quality (again, detailed examples are provided for each example problem in the following subsections). Evaluations must be able to be compared to one another to determine which is the higher quality solution. In addition, evaluations must be able to determine a difference in energy values from other evaluations, for use with the simulated annealing solution strategy. In this class, users must implement a method to compare against other evaluations, as well as method determining the difference in energy values against other evaluations.

\textbf{ExhaustiveGenerator}

This abstract class represents an assignment generator for an exhaustive solution strategy. That is, this class must be able to generate an Assignment object for every possible assignment combination. Because of the exponential nature of an exhaustive solution strategy, the number of possible assignment combinations is limited to $2^{64}$. In this class, users must implement a method to generate an Assignment object, given an index into the solution space.

\textbf{MutationGenerator}

This abstract class represents an assignment generator for a random solution strategy. That is, this class must be able to generate an Assignment randomly from the entire solution space.

\footnote{http://www.cs.rit.edu/~ark/pj.shtml}
addition, it must be able to randomly mutate a given Assignment object, creating a new, mutated Assignment object. In this class, the user must implement a method to do each of these functions.

**Figure 7 Class diagram for the framework.**

**NPOptProblem**

This abstract class represents an NP optimization problem. This class holds the solution strategy that will be used to solve the problem. Users must create a subclass of this class, but aside
from a constructor, nothing needs to be implemented in this class. In my example problems, I chose to implement the setup and running of a problem in the main method of the subclass.

SolutionStrategy

This abstract class represents a solution strategy for combinatorial optimization problems. It has an abstract solve method that must be implemented by the subclassed solution strategy, and returns the best Assignment object found. The user only needs to implement a subclass if they wish to add a new solution strategy.

BruteForce

This class represents the Brute Force solution strategy. BruteForceSMP represents a parallel version of the Brute Force solution strategy. This solution strategy operates as outlined in section IV.A. The user does not need to modify these classes.

HillClimbing

This class represents the Hill Climbing solution strategy. HillClimbingSMP represents a parallel version of the Hill Climbing solution strategy. This solution strategy operates as outlined in section IV.B. The user does not need to modify these classes.

SimulatedAnnealing

This class represents the Simulated Annealing solution strategy. SimulatedAnnealingSMP represents a parallel version of the Simulated Annealing solution strategy. This solution strategy operates as outlined in section IV.C. The user does not need to modify these classes.

CoolingSchedule

This abstract class represents a cooling schedule to be used with Simulated Annealing. It uses the current temperature to calculate an acceptance probability for transitioning to higher energy states. How this temperature is reduced is defined in the subclasses. The user only needs to subclass this if they want to use a cooling schedule other than the two provided.

LinearCoolingSchedule

This class represents a linear cooling schedule. It must have a positive temperature loss value. Each time the cooling schedule reduces the temperature, this temperature loss value is subtracted from the current temperature.

ExponentialCoolingSchedule

This class represents an exponential cooling schedule. It must have a cooling rate value between 0 and 1, non-inclusive. Each time the cooling schedule reduces the temperature, the current temperature is multiplied by the cooling rate value.
VI.B – 0-1 Knapsack

KnapsackAssignment

This subclass represents an assignment for the knapsack problem. The actual assignment is stored in an array of long primitives, where a 0 bit represents the item at that index is excluded, and a 1 bit represents the item at that index is included. To perform an evaluation, the indices are iterated over, and if the bit at that index is a 1, then the item at that index adds its value and weight to running totals for value and weight. Once all indices are iterated over, and all items with a 1 bit at their index have been added to the total, then the resulting total value and weight are used to create a KnapsackEvaluation object.

KnapsackEvaluation

This subclass represents an evaluation of a KnapsackAssignment. The evaluation holds information such as the total value and weight of the combination of items for that particular KnapsackAssignment, as well as the validity of the assignment (whether or not the total weight is under the knapsack capacity). Comparison against another evaluation falls into a few different conditions. If only one evaluation is valid, then the valid one is considered better. If both are valid, then the one with the higher value is considered better. If neither are valid, then the one with the higher density (value divided by weight) is considered better. Determining a difference in energy between two evaluations also falls into different conditions. If both evaluations are valid, the difference in energy is the difference in total values. If not, then the difference in energy is the difference in total weights.

KnapsackExhaustiveGenerator

This subclass represents an exhaustive generator for solving the knapsack problem with an exhaustive solution strategy. It quite simply creates a new KnapsackAssignment object by instantiating it with an array of one long (the index into the solution space).

KnapsackMutationGenerator

This subclass represents a mutation generator for solving the knapsack problem with a random solution strategy. To generate a random assignment, it creates a long array large enough for the number of items (one long for every 64 items), and sets each long to a random bit pattern. To mutate a given assignment, it copies the long array into a new array, and then randomly flips bits throughout the long array. It uses this resultant long array to instantiate a new KnapsackAssignment object.

KnapsackItem

This class represents a knapsack item. Simply holds the value and weight of the item. These items are used by KnapsackAssignment to evaluate a particular assignment.
KnapsackItemList

This class represents a list of knapsack items. In addition to the standard ArrayList functionality, it also keeps track of the total value and weight of items in the list.

KnapsackProblem

This subclass represents the Knapsack problem itself. The main method takes command line arguments for setting up and solving the problem (see section X.A). Aside from the main method, this subclass does not add any functionality to the NPOptProblem abstract class. KnapsackProblemPAR is equivalent, but solves in parallel.

VI.C Maximum Satisfiability

MAXSATAssignment

This subclass represents an assignment for the maximum satisfiability problem. The actual assignment is stored in an array of long primitives, where a 0 bit represents the literal at that index has a false value, and a 1 bit represents the literal at that index has a true value. To perform an evaluation, the clauses are iterated over, and for each clause the literals are iterated over. If any literal in the clause has a 1 bit at its index and the literal is not notted (or has a 0 bit at its index and the literal is notted), then that clause is satisfied and it immediately progresses to the next clause. Once all clauses are iterated over, the total number of satisfied clauses is used to instantiate a MAXSA TEvaluation object.

MAXSA TEvaluation

This subclass represents an evaluation of a MAXSATAssignment. The evaluation holds the number of satisfied and unsatisfied clauses. When comparing against other evaluations, the evaluation with a higher number of satisfied clauses is considered better. The difference in energy between two evaluations is simply the difference in the number of unsatisfied clauses.

MAXSA TExhaustiveGenerator

This subclass represents an exhaustive generator for solving the maximum satisfiability problem with an exhaustive solution strategy. It quite simply creates a new MAXSATAssignment object by instantiating it with an array of one long (the index into the solution space).

MAXSATMutationGenerator

This subclass represents a mutation generator for solving the maximum satisfiability problem with a random solution strategy. To generate a random assignment, it creates a long array large enough for the number of items (one long for every 64 items), and sets each long to a random bit pattern. To mutate a given assignment, it copies the long array into a new array, and then randomly flips bits throughout the long array. It uses this resultant long array to instantiate a new MAXSATAssignment object.
Literal

This class represents a Boolean literal. It holds the index of the literal (for looking up the bit value), as well as whether or not it has been notted.

Clause

This class represents a Boolean clause. It holds a list of Literal objects. Although it subclasses ArrayList, it does not add any additional functionality. It was only subclassed for naming purposes.

Expression

This class represents a Boolean expression. It holds a list of Clause objects, as well as the number of literals (that is, the problem size, not the number of all literals in all clauses).

MAXSATProblem

This subclass represents the maximum satisfiability problem itself. The main method takes command line arguments for setting up and solving the problem (see section X.B). Aside from the main method, this subclass does not add any functionality to the NPOptProblem abstract class. MAXSATProblemPAR is equivalent, but solves in parallel.

VI.C – Minimum Vertex Cover

MVCAssignment

This subclass represents an assignment for the minimum vertex cover problem. The actual assignment is stored in an array of long primitives, where a 0 bit represents the vertex at that index is excluded, and a 1 bit represents the vertex at that index is included. To perform an evaluation, the edge list is cloned into a working list, then the indices are iterated over, and if the bit at that index is a 1, then all edges incident to the vertex at that index are removed from the working list. Once all indices are iterated over, then the size of the working list represents how covered the graph is. The amount of the graph that is covered can be represented by \(1.0 - \frac{\text{size of working list}}{\text{size of original list}}\). This gives the percent of the graph that is covered (1.0 = 100%). This and the vertex count (number of 1 bits in the assignment) are used to instantiate an MVCEvaluation.

MVCEvaluation

This subclass represents an evaluation of an MVCAssignment. The evaluation holds the number of vertices used as well as what percent of the graph was covered. Comparison against another evaluation falls into a few different conditions. If only one evaluation is a cover, then the one that is a cover is considered better. If both are covers, then the one with a smaller vertex count is considered better. If neither are covered, then the one with the higher percent covered is considered better. Determining a difference in energy between two evaluations also falls into different conditions. If both evaluations are covers, the difference in energy is the difference in vertex counts. If not, then the difference in energy is the difference in percent of the graph covered.
**MVCExhaustiveGenerator**

This subclass represents an exhaustive generator for solving the minimum vertex problem with an exhaustive solution strategy. It quite simply creates a new MVCAssignment object by instantiating it with an array of one long (the index into the solution space).

**MVCMutationGenerator**

This subclass represents a mutation generator for solving the minimum vertex cover problem with a random solution strategy. To generate a random assignment, it creates a long array large enough for the number of items (one long for every 64 items), and sets each long to a random bit pattern. To mutate a given assignment, it copies the long array into a new array, and then randomly flips bits throughout the long array. It uses this resultant long array to instantiate a new MVCAssignment object.

**Edge**

This class represents a graph edge. It simply holds the indices of the two incident vertices. These are used by MVCAssignment to evaluate a particular assignment.

**EdgeList**

This class represents a list of graph edges. In addition to the standard ArrayList functionality, it also keeps track of the number of vertices in the graph.

**MVCProblem**

This subclass represents the minimum vertex cover problem itself. The main method takes command line arguments for setting up and solving the problem (see section X.C). Aside from the main method, this subclass does not add any functionality to the NPOptProblem abstract class. MVCProblemPAR is equivalent, but solves in parallel.

**VII. Experiments**

All final experiments were conducted on RIT’s hybrid cluster, Tardis\(^4\). Each test was run on one of the backend computers. Each backend computer contains two AMD Opteron 2218 dual core CPUs running at 2.6GHz clock speed, allowing up to 4 threads for execution. Each experiment was performed sequentially and in parallel, with independent parallel experiments for 1, 2, 3, and 4 threads. For each of these, the experiment was run three times, and the smallest execution time was recorded. This execution time was used to determine speedups.

There were two main sets of experiments – small problem size and large problem size. The small problem size experiments were designed such that, using brute force, the shortest would execute in about one minute sequentially, and the longest would execute in about one hour sequentially. Each of these experiments were also run using the hill climbing and simulated annealing algorithms. The results are presented in Table 1, which shows the performance of the algorithms in terms of execution time and parallel speedup.

---

\(^4\) [http://www.cs.rit.edu/~ark/runningpj.shtml#intro](http://www.cs.rit.edu/~ark/runningpj.shtml#intro)
annealing strategies so that their solution quality and speedup against the brute force execution time could be studied. The parameters for hill climbing for the smaller problem sizes were chosen such that execution would take approximately a couple of minutes. The large problem size experiments were chosen such that they could not be brute forced, but only solved using hill climbing and simulated annealing. The experiments were designed such that, using hill climbing, the shortest would execute in about one minute, and the longest would execute in about one hour. Due to simulated annealing’s alternate stop condition of having no successful transitions during \( T \) trials, there was no attempt to create experiments of a certain execution time for either of the problem sizes. Rather, the simulated annealing experiments simply used similar parameters to the hill climbing experiments. The full list of experiments can be seen in Appendix A.

VIII. Results

VIII.A – Implementation Time

All three problems were implemented both with and without the framework to test whether or not the framework made implementation any easier. As seen in Figure 8, implementing these problems without the framework took approximately twice as long as it did using the framework (rounded to the nearest 15 minutes).

Figure 8 Implementation time for each problem, both with and without the framework. In general, using the framework halved the implementation time.

This is due somewhat to having to re-implement the structure and minor methods that are implemented in the abstract classes, but mostly to having to re-implement the solution strategies. Re-implementing the solution strategies not only requires a significant increase in implementation time, but also an understanding of the solution strategy. When implementing problems using the framework, there was no need to actually understand how any of the three solution strategies worked. Granted, successful execution of the strategies may require some knowledge, such as how to choose the right parameters for simulated annealing, but as far as implementation, no knowledge of the strategies is required. This offers a very useful benefit to the user, especially as more solution strategies become available in the framework.
For all charts in subsections VIII.B, VIII.C, and VIII.D, except for the brute force speedup charts in VIII.B, the gray lines represent sequential results (also noted by the label for that line ending in an ‘S’). For the brute force speedup charts in VIII.B, the one gray line represents the ideal speedup trend.

**VIII.B – Brute Force**

Brute force showed some very interesting results. Aside from a few data points, the general trend showed an increase in speedup when increasing the number of threads. However, in many cases this speedup was above or below the ideal speedup. Due to the differences in the amount of work required to evaluate a solution between problems, each problem used a different range of problem size. Knapsack problems ranged from 27 to 33 items, maximum satisfiability problems ranged from 20 to 26 literals, and minimum vertex cover problems ranged from 23 to 29 vertices.

![Knapsack BF Running Time vs. Processors](chart1)

![Knapsack BF Speedup vs. Processors](chart2)

**Figure 9** Knapsack brute force results. In general, there was a reduction in running time with additional processors, and the speedups in many cases were above ideal.

![MAXSAT BF Running Time vs. Processors](chart3)

![MAXSAT BF Speedup vs. Processors](chart4)

**Figure 10** MAXSAT brute force results. In general, there was a reduction in running time with additional processors. Speedups for more than 1 thread were all less than ideal, however.
Figure 11 MVC brute force results. In general, there was a reduction in running time with additional processors. There was more variation in speedup, but again, with more than 1 thread, all speedups were less than ideal.

I believe that the main causes of these discrepancies are the JDK being used and the machine potentially being worked on. Upon completing the source code, I ran preliminary tests on both my personal machine and RIT’s paradox SMP computer. With each machine, I tried running with JDK 1.5 (as recommended on Professor Kaminsky’s Parallel Java library website) as well as JDK 1.7. JDK 1.7 not only reduced execution time, but also displayed near ideal speedups. However, at the time of final testing, the machine that was available to me (the tardis hybrid cluster) was not configured to run JDK 1.7. In addition, although the job queue was being used, the system administrators had been notified of the JDK issue and may have been working on it during certain experiment executions. In some cases, there were significant differences in execution times of the three independent timings.

Although some of the speedups were less than ideal, these results do support the hypothesis to an extent.

VIII.C – Hill Climbing

Smaller Problems

Each of the brute force problem cases were also tested with hill climbing. This problem set was mainly designed to show quality as a function of the brute force solution. It also shows, as will the larger problem set, the reduction in running time as the number of processors increases. Both the knapsack and maximum satisfiability problems show fairly consistent results, while minimum vertex cover had much more variance. The knapsack problem actually showed no improvement in solution quality when increasing the processor count, but the other two problems had a few cases where solution quality improved. In one of the minimum vertex cover problems, going from 1 to 2 threads increased the solution quality by over 20%. This is mainly due to the solutions having very small values for these small problem sizes (i.e. there may only be a 1 vertex difference, but this represents a large percent when the solutions contain only a dozen or so vertices). Due to the differences in the amount of work required to evaluate a solution between problems, different numbers of iterations and mutations were used for each problem (but for each
problem, the same number of iterations and mutations were used for each problem size). Knapsack used 10000 iterations and mutations, maximum satisfiability used 1000 iterations and mutations, and minimum vertex cover used 1000 iterations and 10000 mutations. When mutating solutions, all problems made two changes to the solution (i.e. flipped two bits).

![Knapsack HC Running Time vs. Processors](image1)

**Figure 12** Knapsack hill climbing results for small problem sizes. In general, there was a reduction in running time with additional processors. In this particular example, increasing threads did not provide an increase in quality.

![Knapsack HC Solution Quality vs. Processors](image2)

The minimum vertex running time results were rather interesting, in particular the two problems that took longer on one thread of the parallel algorithm than the sequential algorithm. Again, I believe this may have the same reasoning as the brute force results. There was again a drastic difference in one of the three independent timings. By chance, the single thread executions for two experiments did not have one drastically shorting running time, and therefore are shown to take longer than the sequential. However, aside from the one lucky timing, the sequential and

![MAXSAT HC Running Time vs. Processors](image3)

**Figure 13** MAXSAT hill climbing results for small problem sizes. In general, there was a reduction in running time with additional processors. There are also some clear quality improvements. In once case, only with 4 processors was the optimal solution found.

![MAXSAT HC Solution Quality vs. Processors](image4)
single thread executions were almost the same length. It would be interesting to rerun the tests on a machine that is not potentially being working on by system administrators.

Figure 14 MVC hill climbing results for small problem sizes. Although less consistent with previous graphs, there was still a general reduction in running time with additional processors. There were also some quality improvements when moving from 1 to 2 threads.

Larger Problems

The larger problems were designed to show the framework’s ability to handle large problem sizes that could not be brute forced. All of the problems used a problem size of N=100. The only thing that changed between runs was the number of mutations performed per iteration. Due to the differences in the amount of work required to evaluate a solution between problems, different numbers of iterations and mutations were used for each problem. Knapsack used 10000 iterations and mutation numbers of 3000, 6000, 12000, 24000, 48000, 96000, and 192000. Maximum satisfiability used 500 iterations and mutation numbers of 800, 1600, 3200, 6400, 12800, 25600, and 51200. Minimum vertex cover used 100 iterations and mutations numbers of 1000, 2000,

Figure 15 Knapsack hill climbing results for large problem sizes. There is a general reduction in running time with additional processors. There was also a slight quality improvement in some cases going from 2 to 3 processors.
Figure 16 MAXSAT hill climbing results for large problem sizes. There is a general reduction in running time with additional processors. There was no improvement in solution quality, however.

Figure 17 MVC hill climbing results for large problem sizes. As with the small problem sizes, the reduction in speed is less consistent. Additionally, in one case, the quality actually decreased with 4 processors.

Because the optimal solution could not be determined, the quality charts here used the best solution found as the optimal solution. This was meant to show that increasing the number of processors would gradually increase the quality while still keeping it on a percentage scale for easy comparison. In the following charts, ‘NM’ stands for number of mutations.

The running time results were fairly consistent with the results of the small problem sizes. However, there were fewer and less drastic quality improvements. In the case of minimum vertex cover with 1000 mutations, the quality actually decreased with 4 processors. This is due to each processor looking at a reduced number of mutations. It must have been the case that one of the threads in the 3 processor run found the higher quality solution somewhere between 250 and 333 mutations. This would not have been the case if each processor simply ran a unique copy of the
sequential algorithm. However, it was only the one case, the loss in quality was not too drastic, and the way it is implemented does provide a noticeable reduction in running time.

In both the small and large problem sets, the parallel hill climbing algorithm did show a general speedup when increasing the number of processors, confirming the primary hypothesis. Because there were many cases which did show a quality improvement, the secondary hypothesis is supported to some extent. However, due to the case where quality decreased, I will add that it is problem dependent.

**VIII.D - Simulated Annealing**

**Smaller problems**

Each of the brute force problem cases were also tested with simulated annealing. This problem set was mainly designed to show quality as a function of the brute force solution. The results for simulated annealing were very promising, both in terms of execution time and solution quality. Each problem had at least one example of improving solution quality with additional processors. In addition, aside from maximum satisfiability, there was not much variance in execution time. Even then, the largest spread was only ~4 seconds. With 4 threads, the solution quality was above 91% in all cases. For most cases, simulated annealing resulted in an equal or higher quality solution compared to the hill climbing algorithm, but executed in significantly less time (just over 7 seconds in the longest case - roughly a third of the shortest hill climbing execution). All of the problems used an exponential cooling schedule, with a cooling rate of 0.9. Due to the differences in the amount of work required to evaluate a solution between problems, different numbers of temperature steps were used for each problem (but for each problem, the same number of temperature steps were used for each problem size). Knapsack used 100000 temperature steps with a starting temperature of 50, maximum satisfiability used 1000 temperature steps with a starting temperature of 500, and minimum vertex cover used 1000 temperature steps with a starting temperature of 50. When mutating solutions, all problems made two changes to the solution (i.e. flipped two bits).

![Figure 18 Knapsack simulated annealing results for small problem sizes. Running time in general stayed approximately the same. There were also improvements to quality with additional processors.](image)
The variance in running time that does exist is likely due to how the simulated annealing algorithm terminates. One termination condition is that all temperature steps have been iterated over, which would be the full execution time of the algorithm. There is an early stop condition, however, if the algorithm fails to improve the solution quality any further. As mentioned in section IV.C, if, during a temperature step, the algorithm goes through $\tilde{T}$ trials without a single successful transition, the algorithm terminates. When running in parallel, each thread has a unique mutation generator and random number generator. Because of this, when increasing the number of threads, the new thread may not reach this early stop condition as quickly as the original threads, increasing the overall execution time.
Larger Problems

The larger problems were designed to show the frameworks ability to handle large problem sizes that could not be brute forced. All of the problems used a problem size of N=100. The only thing that changed between runs was the number of temperature steps performed. As a reminder, the number of temperature steps for each problem here is equal to the number of mutations for each of the hill climbing problems (e.g. knapsack simulated annealing starts with 3000 temperature steps because the hill climbing started with 3000 mutations). When mutating solutions, all problems made three changes to the solution (i.e. flipped three bits).

Because the optimal solution could not be determined, the quality charts here used the best solution found as the optimal solution. This was meant to show that increasing the number of processors would gradually increase the quality while still keeping it on a percentage scale for easy comparison. In the following charts, ‘TS’ stands for temperature steps.

Figure 21 Knapsack simulated annealing results for large problem sizes. Once again, the execution time is very small and shows little variance. There were also some clear improvements to solution quality.

Figure 22 MAXSAT simulated annealing results for large problem sizes. Some small variations in running times. All cases found a better solution going from 1 to 2 threads.
The results were not quite as impressive as the smaller problem size results. Similar to the larger problem size hill climbing results, there were fewer and less drastic improvements to quality. Still, the execution time was typically shorter than the hill climbing, and significantly shorter for the larger parameter sizes, due to simulated annealing’s early termination condition. Interestingly, all but one of the minimum vertex cover problem sizes had a drastic increase in running time when going from 1 to 2 threads, followed by a drastic decrease in running time going from 2 to 3 threads. The run that did not significantly increase running time at 2 threads just happened to have one lucky run that took ~150 seconds less time than the other two. Some variation in execution times is understandable, but this much variation is boggling. The only potential causes that come to mind are JDK issues, JIT compiler issues, or that the machine was being accessed during certain executions.

In both the small and large problem sets, there was at least one example per problem that showed an improvement in quality with additional processors. Although the execution time was not the same for all problems, this is understandable due to simulated annealing’s alternate stop condition. Knowing this, the results for simulated annealing have confirmed its hypothesis.

There seemed to be a pattern of “one lucky timing” in a surprising number of the experiments. I believe that this has skewed some of the results of my experiments. As previously stated, compiling and running with JDK1.7 on the paradox machine consistently showed lower running times and much more ideal speedups. It is possible that anyone rerunning my tests in their own environment may see different results.

IX. Developer Manual

To compile the framework, Professor Kaminsky’s Parallel Java library must be included in your classpath. Once it is included, all of the source files can be compiled using the javac command.
To compile a problem that you have implemented using the framework, ensure that Professor Kaminsky’s Parallel Java library is included in your classpath, as well as the parallel framework library (either the supplied jar, or the compiled classes from the above step). With both of these in your classpath, simply use the javac command to compile your source code.

X. User Manual

To implement a problem, simply follow the documentation and examples provided. The user must implement subclasses for Assignment, Evaluation, ExhaustiveGenerator, MutationGenerator, and NPOptProblem, as well as any additional classes required by the problem. Running the problem will depend on how the NPOptProblem subclass has been implemented. In the three example problems, command line arguments can be passed to this subclass’s main method to create and solve the particular problem.

X.A - 0-1 Knapsack

To run my implementation of the 0-1 Knapsack problem, either KnapsackProblem (sequential) or KnapsackProblemPAR (parallel) must be run. The command line arguments are the same for both classes. For the parallel run, it will use all available threads for execution unless otherwise indicated with the –Dpj.nt flag. Both the Parallel Java library as well as the parallel framework library must be included in your classpath. The command should be run as follows:

```
java KnapsackProblem <N> <LV> <UV> <LW> <UW> <capacity> <numIterations> <numMutations> <mutations> <startTemp> <modes> <seed>
```

where

- `<N>` = Number of knapsack items
- `<LV>` = Lower bound value for item values
- `<UV>` = Upper bound value for item values
- `<LW>` = Lower bound value for item weights
- `<UW>` = Upper bound value for item weights
- `<capacity>` = Weight capacity of the knapsack
- `<numIterations>` = Number of iterations/temp steps to perform (hc/sa)
- `<numMutations>` = Number of mutations to perform (hc)
- `<mutations>` = Number of changes to make when mutating assignments
- `<startTemp>` = Starting temperature for cooling schedule (sa)
- `<modes>` = Three 0/1 characters indicating strategy selection (‘101’ = bf and sa)
- `<seed>` = Seed for pseudo random number generator

The problem is generated as follows. The seed will be used to create a random number generator. N KnapsackItem objects are created sequentially. This is done by using the random number generator to generate two random doubles, one for value (in the range [LV, UV]) and one for weight (in the range [LW, UW]). These items are added to a KnapsackItemList, along with the knapsack capacity. This item list is then used in evaluation.
X.B – Maximum Satisfiability

To run my implementation of the Maximum Satisfiability problem, either MAXSATProblem (sequential) or MAXSATProblemPAR (parallel) must be run. The command line arguments are the same for both classes. For the parallel run, it will use all available threads for execution unless otherwise indicated with the –Dpj.nt flag. Both the Parallel Java library as well as the parallel framework library must be included in your classpath. The command should be run as follows:

```
java MAXSATProblem <N> <C> <LL> <UL> <numIterations> <numMutations> <mutations> <startTemp> <modes> <seed>
```

where

- `<N>` = Number of Boolean literals
- `<C>` = Number of Boolean clauses to create
- `<LL>` = Lower bound number of literals per clause
- `<UL>` = Upper bound number of literals per clause
- `<numIterations>` = Number of iterations/temp steps to perform (hc/sa)
- `<numMutations>` = Number of mutations to perform (hc)
- `<mutations>` = Number of changes to make when mutating assignments
- `<startTemp>` = Starting temperature for cooling schedule (sa)
- `<modes>` = Three 0/1 characters indicating strategy selection (‘101’ = bf and sa)
- `<seed>` = Seed for pseudo random number generator

The problem is generated as follows. The seed will be used to create a random number generator. C Clause objects are created sequentially. For each clause, a number of literals in the range [LL, UL] are added to the clause. For each literal, the random number generator generates a random number between 0 and N-1 (the index of the literal), and a Boolean value for whether the literal is notted in that clause or not. Each clause is added to an Expression object, and once all clauses are added, the expression is used in evaluations.

X.C - Minimum Vertex Cover

To run my implementation of the Minimum Vertex Cover problem, either MVCProblem (sequential) or MVCProblemPAR (parallel) must be run. The command line arguments are the same for both classes. For the parallel run, it will use all available threads for execution unless otherwise indicated with the –Dpj.nt flag. Both the Parallel Java library as well as the parallel framework library must be included in your classpath. The command should be run as follows:

```
java MVCProblem <N> <LE> <UE> <numIterations> <numMutations> <mutations> <startTemp> <modes> <seed>
```

where

- `<N>` = Number of graph vertices
- `<LE>` = Lower bound number of outgoing edges per vertex
- `<UE>` = Upper bound number of outgoing edges per vertex
<numIterations> = Number of iterations/temp steps to perform (hc/sa)
<numMutations> = Number of mutations to perform (hc)
<mutations> = Number of changes to make when mutating assignments
<startTemp> = Starting temperature for cooling schedule (sa)
<modes> = Three 0/1 characters indicating strategy selection ('101' = bf and sa)
<seed> = Seed for pseudo random number generator

The problem is generated as follows. The seed will be used to create a random number generator. For each value 0 to N-1 representing a vertex, a number of Edge objects in the range [LE, UE] are created for that vertex. The other incident vertex is a randomly generated number in the range 0 to N-1, excluding the current value (i.e. a vertex cannot have an edge to itself). Each edge is added to an EdgeList object, which is then used in evaluations.

XI. Future Work

There are two main areas of interest in regards to future work. Firstly, I would like to expand the framework to support cluster and hybrid parallelism. Currently, the framework only supports shared memory multiprocessor systems. A cluster version would support a network of single processor computers connected over a network. A hybrid version would support a network of shared memory multiprocessor computers connected over a network. Considering the minimal amount of interprocess communication, I believe that these additional parallel platforms would be very useful. The actual implementation to support these would not be very time consuming, but due to resource and time availability, they could not be included in this work.

The other area of interest would be supporting additional solution strategies. The framework currently supports brute force, hill climbing, and simulated annealing. However, there are many other applicable solution strategies that could be added. One such example would be a genetic algorithm.

Additionally, it would be useful to experiment with even larger problem sizes, more on par with known benchmarks. This would involve problems where N is in the thousands. Problems of this size can have great variation in their assignments and mutations, and could potentially show more improvements in solution quality.

XII. Lessons Learned

Most of what was learned during this project was not new knowledge, but reinforcement against bad habits. Specifically, I was forced to deal with self-motivation and proper planning. Unlike normal classwork, the project does not have a due date and does not get periodically graded. There were no authoritative consequences for procrastination. Thus, I had to find my own sources of motivation. This was easy during stages such as implementing the framework, because I am passionate about coding, but much more difficult during phases I enjoy less, such as writing reports. My main drive during the more difficult phases was the fact that I’ve already received and accepted a job offer, and I wanted to ensure that I had completed the project before starting the job.

Because of this pressure to finish before starting a job, proper planning was very important to reaching my goal. For the most part, my plans were well laid out and I accomplished them satisfactorily and on time. I did, however, fail to properly plan for a major bug (the parallel hill
climbing solution strategy was producing different results on different runs despite using the same arguments). This was a failure on two levels. Firstly, I did not perform sufficient preliminary tests to catch this bug sooner. This resulted in me having to rerun all of the hill climbing tests after the bug was fixed. Secondly, I did not include bug fixing in my schedule, which should have been included. Luckily, I was ahead of schedule for most of the project, so this did not cause me to fall behind schedule.

XIII. Conclusion

A parallel framework for NP combinatorial optimization problems developed in Java has been presented. Although other frameworks are available, most of them are focused on researching the heuristics and metaheuristics themselves. Also, the more up-to-date ones are implemented typically in C/C++. The framework proposed here has been shown to provide a clear reduction in implementation time when implementing new problems. For each new solution strategy added, this time reduction with increase further. In addition, the framework allows the user to implement a problem and solve it using three solution strategies without needing knowledge of the solution strategies (although, some knowledge is useful for parameter selection when executing). This is an important benefit to the framework.

The other hypotheses were not supported as strongly. The brute force algorithm did typically show a speedup, but there were some sporadic cases including far above and far below ideal speedup. There were also outliers that actually ran slower with additional processors. One of the main problems was that, in many cases, there would be “one lucky timing” amongst the three timing runs. This one timing would be drastically lower than the other two. Thus, experiments that did not get a lucky timing appeared to be much slower. It is likely that, given many runs, it would eventually have a shorter execution time. This was very unfortunate, as it skewed some of the results and has seemingly no explanation. That said, based on preliminary tests in alternate environments, I believe that better results can be achieved with the correct environment. Regardless, I believe there was enough evidence to support the claim that the parallel brute force algorithm provides a speedup.

The hill climbing results certainly showed a reduction in execution time, with minor variation, thereby confirming its primary hypothesis. However, there were mixed results regarding the secondary hypothesis. In many cases, increasing the number of threads did improve solution quality, but there were also many cases that did not improve quality, and even one where quality was decreased. This was due to having a very small number of mutations per thread. Based on this, it seems that using the parallel version with very small mutation numbers can be detrimental. However, for large numbers of mutations, it seems to confirm the hypothesis to some extent.

Lastly, the simulated annealing results confirmed its hypothesis by showing an increase in solution quality in many cases. Although in some cases this quality improvement was very slight, there is still enough evidence in support of the hypothesis. In addition, simulated annealing typically found a solution equal to or better than hill climbing in a significantly shorter period of time. It was also capable of finding the global optimum in many cases, in much less time than brute force took.

References


### Appendix A – Experiment Runs

Each of the following experiments was run using JDK 1.5. Each experiment was repeated three times, and the lowest execution time was recorded. For the parallel runs, only the 1 thread command line is shown. The runs for higher thread counts were identical, but with a different value for –Dpj.nt=1. As a reminder, the meaning of the parameters can be seen in section X for each problem.

#### Knapsack

##### Brute Force

```plaintext
<table>
<thead>
<tr>
<th>Command</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>java KnapsackProblem</td>
<td>27 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>27 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>28 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>28 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>29 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>29 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>30 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>30 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>31 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>31 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>32 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>32 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>33 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>33 50 100 20 30 375 1 1 1 1 100 324984135476843</td>
</tr>
</tbody>
</table>
```

##### Hill Climbing (Small)

```plaintext
<table>
<thead>
<tr>
<th>Command</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>java KnapsackProblem</td>
<td>27 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>27 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>28 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>28 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>29 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>29 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>30 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>30 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>31 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>31 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>32 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>32 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java KnapsackProblem</td>
<td>33 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
<tr>
<td>java -Dpj.nt=1 KnapsackProblemPAR</td>
<td>33 50 100 20 30 375 10000 10000 2 50 010 324984135476843</td>
</tr>
</tbody>
</table>
```
Simulated Annealing (Small)

```
java KnapsackProblem 27 50 100 20 30 375 100000 1 2 50 001 324984135476843
java -Dpj.nt=1 KnapsackProblemPAR 27 50 100 20 30 375 100000 1 2 50 001 324984135476843
java KnapsackProblem 28 50 100 20 30 375 100000 1 2 50 001 324984135476843
java -Dpj.nt=1 KnapsackProblemPAR 28 50 100 20 30 375 100000 1 2 50 001 324984135476843
java KnapsackProblem 29 50 100 20 30 375 100000 1 2 50 001 324984135476843
java -Dpj.nt=1 KnapsackProblemPAR 29 50 100 20 30 375 100000 1 2 50 001 324984135476843
java KnapsackProblem 30 50 100 20 30 375 100000 1 2 50 001 324984135476843
java -Dpj.nt=1 KnapsackProblemPAR 30 50 100 20 30 375 100000 1 2 50 001 324984135476843
java KnapsackProblem 31 50 100 20 30 375 100000 1 2 50 001 324984135476843
java -Dpj.nt=1 KnapsackProblemPAR 31 50 100 20 30 375 100000 1 2 50 001 324984135476843
java KnapsackProblem 32 50 100 20 30 375 100000 1 2 50 001 324984135476843
java -Dpj.nt=1 KnapsackProblemPAR 32 50 100 20 30 375 100000 1 2 50 001 324984135476843
java KnapsackProblem 33 50 100 20 30 375 100000 1 2 50 001 324984135476843
java -Dpj.nt=1 KnapsackProblemPAR 33 50 100 20 30 375 100000 1 2 50 001 324984135476843
```

Hill Climbing (Large)

```
java KnapsackProblem 100 50 100 20 30 375 100000 3000 3 50 010 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 100 20 30 375 100000 3000 3 50 010 9846324179143
java KnapsackProblem 100 50 100 20 30 375 100000 6000 3 50 010 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 100 20 30 375 100000 6000 3 50 010 9846324179143
java KnapsackProblem 100 50 100 20 30 375 100000 12000 3 50 010 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 100 20 30 375 100000 12000 3 50 010 9846324179143
java KnapsackProblem 100 50 100 20 30 375 100000 24000 3 50 010 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 100 20 30 375 100000 24000 3 50 010 9846324179143
java KnapsackProblem 100 50 100 20 30 375 100000 48000 3 50 010 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 100 20 30 375 100000 48000 3 50 010 9846324179143
java KnapsackProblem 100 50 100 20 30 375 100000 96000 3 50 010 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 100 20 30 375 100000 96000 3 50 010 9846324179143
java KnapsackProblem 100 50 100 20 30 375 100000 192000 4 50 010 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 100 20 30 375 100000 192000 4 50 010 9846324179143
```
Simulated Annealing (Large)

```
java KnapsackProblem    100 50 20 30 1350 3000 1 3 50 001 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 20 30 1350 3000 1 3 50 001 9846324179143
java KnapsackProblem    100 50 20 30 1350 6000 1 3 50 001 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 20 30 1350 6000 1 3 50 001 9846324179143
java KnapsackProblem    100 50 20 30 1350 12000 1 3 50 001 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 20 30 1350 12000 1 3 50 001 9846324179143
java KnapsackProblem    100 50 20 30 1350 24000 1 3 50 001 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 20 30 1350 24000 1 3 50 001 9846324179143
java KnapsackProblem    100 50 20 30 1350 48000 1 3 50 001 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 20 30 1350 48000 1 3 50 001 9846324179143
java KnapsackProblem    100 50 20 30 1350 96000 1 3 50 001 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 20 30 1350 96000 1 3 50 001 9846324179143
java KnapsackProblem    100 50 20 30 1350 192000 1 4 50 001 9846324179143
java -Dpj.nt=1 KnapsackProblemPAR 100 50 20 30 1350 192000 1 4 50 001 9846324179143
```

Maximum Satisfiability

Brute Force

```
java MAXSATProblem 20 1000 3 3 1 1 1 1 1 100 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR 20 1000 3 3 1 1 1 1 1 100 16874301079873
java MAXSATProblem 21 1000 3 3 1 1 1 1 1 100 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR 21 1000 3 3 1 1 1 1 1 100 16874301079873
java MAXSATProblem 22 1000 3 3 1 1 1 1 1 100 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR 22 1000 3 3 1 1 1 1 1 100 16874301079873
java MAXSATProblem 23 1000 3 3 1 1 1 1 1 100 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR 23 1000 3 3 1 1 1 1 1 100 16874301079873
java MAXSATProblem 24 1000 3 3 1 1 1 1 1 100 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR 24 1000 3 3 1 1 1 1 1 100 16874301079873
java MAXSATProblem 25 1000 3 3 1 1 1 1 1 100 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR 25 1000 3 3 1 1 1 1 1 100 16874301079873
java MAXSATProblem 26 1000 3 3 1 1 1 1 1 100 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR 26 1000 3 3 1 1 1 1 1 100 16874301079873
```
Hill Climbing (Small)

```
java MAXSATProblem    20 1000 3 3 1000 1000 2 500 0 10 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  20 1000 3 3 1000 1000 2 500 0 10 16874301079873

java MAXSATProblem    21 1000 3 3 1000 1000 2 500 0 10 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  21 1000 3 3 1000 1000 2 500 0 10 16874301079873

java MAXSATProblem    22 1000 3 3 1000 1000 2 500 0 10 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  22 1000 3 3 1000 1000 2 500 0 10 16874301079873

java MAXSATProblem    23 1000 3 3 1000 1000 2 500 0 10 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  23 1000 3 3 1000 1000 2 500 0 10 16874301079873

java MAXSATProblem    24 1000 3 3 1000 1000 2 500 0 10 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  24 1000 3 3 1000 1000 2 500 0 10 16874301079873

java MAXSATProblem    25 1000 3 3 1000 1000 2 500 0 10 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  25 1000 3 3 1000 1000 2 500 0 10 16874301079873

java MAXSATProblem    26 1000 3 3 1000 1000 2 500 0 10 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  26 1000 3 3 1000 1000 2 500 0 10 16874301079873
```

Simulated Annealing (Small)

```
java MAXSATProblem    20 1000 3 3 1000 1 2 500 001 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  20 1000 3 3 1000 1 2 500 001 16874301079873

java MAXSATProblem    21 1000 3 3 1000 1 2 500 001 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  21 1000 3 3 1000 1 2 500 001 16874301079873

java MAXSATProblem    22 1000 3 3 1000 1 2 500 001 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  22 1000 3 3 1000 1 2 500 001 16874301079873

java MAXSATProblem    23 1000 3 3 1000 1 2 500 001 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  23 1000 3 3 1000 1 2 500 001 16874301079873

java MAXSATProblem    24 1000 3 3 1000 1 2 500 001 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  24 1000 3 3 1000 1 2 500 001 16874301079873

java MAXSATProblem    25 1000 3 3 1000 1 2 500 001 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  25 1000 3 3 1000 1 2 500 001 16874301079873

java MAXSATProblem    26 1000 3 3 1000 1 2 500 001 16874301079873
java -Dpj.nt=1 MAXSATProblemPAR  26 1000 3 3 1000 1 2 500 001 16874301079873
```
### Hill Climbing (Large)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Execution Time</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 500 800 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 500 800 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 500 1600 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 500 1600 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 500 3200 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 500 3200 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 500 6400 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 500 6400 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 500 12800 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 500 12800 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 500 25600 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 500 25600 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 500 51200 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 500 51200 3 500 010</td>
<td>984735498716654</td>
<td></td>
</tr>
</tbody>
</table>

### Simulated Annealing (Large)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Execution Time</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 800 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 800 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 1600 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 1600 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 3200 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 3200 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 6400 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 6400 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 12800 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 12800 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 25600 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 25600 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java MAXSATProblem</td>
<td>100 2000 3 3 51200 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
<tr>
<td>java -Dpj.nt=1 MAXSATProblemPAR</td>
<td>100 2000 3 3 51200 1 3 500 001</td>
<td>984735498716654</td>
<td></td>
</tr>
</tbody>
</table>
Minimum Vertex Cover

Brute Force

java MVCProblem 23 1 2 1 1 1 1 100 987635198473
java -Dpj.nt=1 MVCProblemPAR 23 1 2 1 1 1 1 100 987635198473

java MVCProblem 24 1 2 1 1 1 1 100 987635198473
java -Dpj.nt=1 MVCProblemPAR 24 1 2 1 1 1 1 100 987635198473

java MVCProblem 25 1 2 1 1 1 1 100 987635198473
java -Dpj.nt=1 MVCProblemPAR 25 1 2 1 1 1 1 100 987635198473

java MVCProblem 26 1 2 1 1 1 1 100 987635198473
java -Dpj.nt=1 MVCProblemPAR 26 1 2 1 1 1 1 100 987635198473

java MVCProblem 27 1 2 1 1 1 1 100 987635198473
java -Dpj.nt=1 MVCProblemPAR 27 1 2 1 1 1 1 100 987635198473

java MVCProblem 28 1 2 1 1 1 1 100 987635198473
java -Dpj.nt=1 MVCProblemPAR 28 1 2 1 1 1 1 100 987635198473

java MVCProblem 29 1 2 1 1 1 1 100 987635198473
java -Dpj.nt=1 MVCProblemPAR 29 1 2 1 1 1 1 100 987635198473

Hill Climbing (Small)

java MVCProblem 23 1 2 1000 10000 2 500 010 987635198473
java -Dpj.nt=1 MVCProblemPAR 23 1 2 1000 10000 2 500 010 987635198473

java MVCProblem 24 1 2 1000 10000 2 500 010 987635198473
java -Dpj.nt=1 MVCProblemPAR 24 1 2 1000 10000 2 500 010 987635198473

java MVCProblem 25 1 2 1000 10000 2 500 010 987635198473
java -Dpj.nt=1 MVCProblemPAR 25 1 2 1000 10000 2 500 010 987635198473

java MVCProblem 26 1 2 1000 10000 2 500 010 987635198473
java -Dpj.nt=1 MVCProblemPAR 26 1 2 1000 10000 2 500 010 987635198473

java MVCProblem 27 1 2 1000 10000 2 500 010 987635198473
java -Dpj.nt=1 MVCProblemPAR 27 1 2 1000 10000 2 500 010 987635198473

java MVCProblem 28 1 2 1000 10000 2 500 010 987635198473
java -Dpj.nt=1 MVCProblemPAR 28 1 2 1000 10000 2 500 010 987635198473

java MVCProblem 29 1 2 1000 10000 2 500 010 987635198473
java -Dpj.nt=1 MVCProblemPAR 29 1 2 1000 10000 2 500 010 987635198473
Simulated Annealing (Small)

```java
java MVCProblem    23 1 2 1000 1 2 50 001 987635198473
java -Dpj.nt=1 MVCProblemPAR  23 1 2 1000 1 2 50 00 1 987635198473
java MVCProblem    24 1 2 1000 1 2 50 001 987635198473
java -Dpj.nt=1 MVCProblemPAR  24 1 2 1000 1 2 50 00 1 987635198473
java MVCProblem    25 1 2 1000 1 2 50 001 987635198473
java -Dpj.nt=1 MVCProblemPAR  25 1 2 1000 1 2 50 00 1 987635198473
java MVCProblem    26 1 2 1000 1 2 50 001 987635198473
java -Dpj.nt=1 MVCProblemPAR  26 1 2 1000 1 2 50 00 1 987635198473
java MVCProblem    27 1 2 1000 1 2 50 001 987635198473
java -Dpj.nt=1 MVCProblemPAR  27 1 2 1000 1 2 50 00 1 987635198473
java MVCProblem    28 1 2 1000 1 2 50 001 987635198473
java -Dpj.nt=1 MVCProblemPAR  28 1 2 1000 1 2 50 00 1 987635198473
java MVCProblem    29 1 2 1000 1 2 50 001 987635198473
java -Dpj.nt=1 MVCProblemPAR  29 1 2 1000 1 2 50 00 1 987635198473
```

Hill Climbing (Large)

```java
java MVCProblem    100 3 5 100 1000 3 500 010 126354798613
java -Dpj.nt=1 MVCProblemPAR  100 3 5 100 1000 3 500 010 126354798613
java MVCProblem    100 3 5 100 2000 3 500 010 126354798613
java -Dpj.nt=1 MVCProblemPAR  100 3 5 100 2000 3 500 010 126354798613
java MVCProblem    100 3 5 100 4000 3 500 010 126354798613
java -Dpj.nt=1 MVCProblemPAR  100 3 5 100 4000 3 500 010 126354798613
java MVCProblem    100 3 5 100 8000 3 500 010 126354798613
java -Dpj.nt=1 MVCProblemPAR  100 3 5 100 8000 3 500 010 126354798613
java MVCProblem    100 3 5 100 16000 3 500 010 126354798613
java -Dpj.nt=1 MVCProblemPAR  100 3 5 100 16000 3 500 010 126354798613
java MVCProblem    100 3 5 100 32000 3 500 010 126354798613
java -Dpj.nt=1 MVCProblemPAR  100 3 5 100 32000 3 500 010 126354798613
java MVCProblem    100 3 5 100 64000 3 500 010 126354798613
java -Dpj.nt=1 MVCProblemPAR  100 3 5 100 64000 3 500 010 126354798613
```
Simulated Annealing (Large)

```
java MVCProblem    100 3 5 1000 1 3 500 001 126354798613
java -Dpj.nt=1 MVCProblemPAR 100 3 5 1000 1 3 500 001 126354798613

java MVCProblem    100 3 5 2000 1 3 500 001 126354798613
java -Dpj.nt=1 MVCProblemPAR 100 3 5 2000 1 3 500 001 126354798613

java MVCProblem    100 3 5 4000 1 3 500 001 126354798613
java -Dpj.nt=1 MVCProblemPAR 100 3 5 4000 1 3 500 001 126354798613

java MVCProblem    100 3 5 8000 1 3 500 001 126354798613
java -Dpj.nt=1 MVCProblemPAR 100 3 5 8000 1 3 500 001 126354798613

java MVCProblem    100 3 5 16000 1 3 500 001 126354798613
java -Dpj.nt=1 MVCProblemPAR 100 3 5 16000 1 3 500 001 126354798613

java MVCProblem    100 3 5 32000 1 3 500 001 126354798613
java -Dpj.nt=1 MVCProblemPAR 100 3 5 32000 1 3 500 001 126354798613

java MVCProblem    100 3 5 64000 1 3 500 001 126354798613
java -Dpj.nt=1 MVCProblemPAR 100 3 5 64000 1 3 500 001 126354798613
```