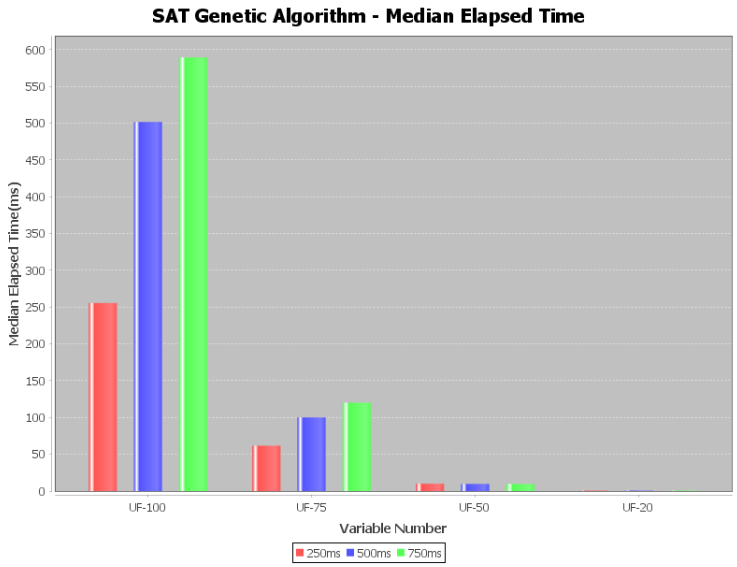
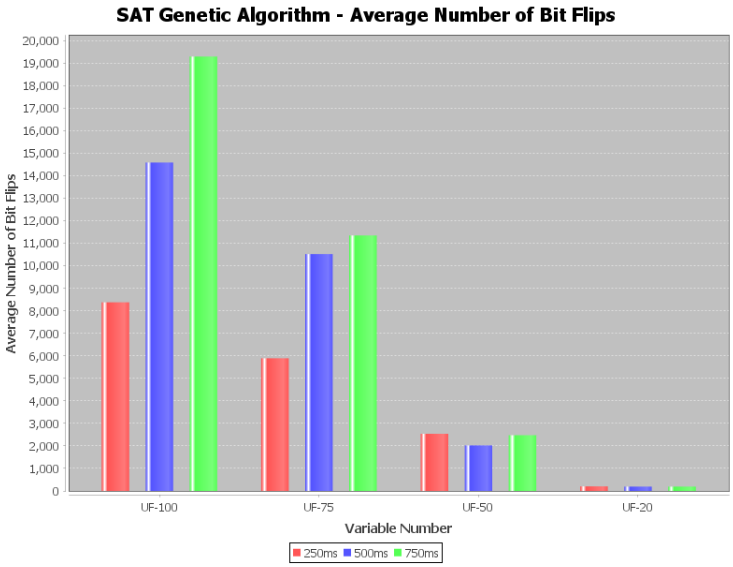
Problem 1

In this problem we were asked to use a genetic algorithm to determine a satisfiable solution for various 3-CNF statements. All of these 3-CNF statements used as benchmarks were satisfiable, our benchmarks only determine if a satisfiable solution can be found with certain time limits. Each statement was contained in its own file. We tested our algorithm on 100 different benchmarks (files) for 20, 50, 75, and 100 variable 3-CNF problems. The algorithm was allowed to run three times for each benchmark each time with a 250ms, 500ms, and 750ms time limit respectively. If an algorithm ran to its alotted time limit it was said to be a failure and was terminated.



Median Elapsed Time: Above is a graph displaying the results of our benchmark evaluations in terms of the median elapsed time. The most interesting result here occured in our UF-100 tests. For the 250ms time limit and 500ms time limit tests the average median time was 250ms and 500ms respectively. Clearly, this means that none were allowed to run to completion and failed to find a satisfiable solution in their time limits. With a 750ms time alottment however, the average runtime was about 580ms implying many, but not all, ran to completion.

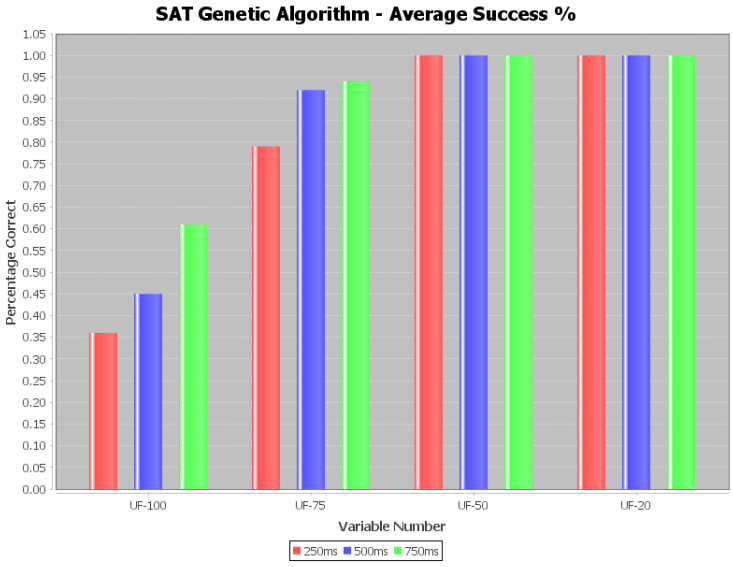
As a trend, the fewer variables a problem had, the less runtime it required. 20 variable problems required median runtimes of less than 5ms on average. 50 variable problems required median runtimes of less than 25ms on average. Based on the results of our benchmark evaluations it seems as though the trend may be that as the number of variables increases the median runtime increases exponentially. This is hard to see in the UF-100 column as the times were explicitly limited to 250 and 500ms and thus solution to completion cannot accurately be described. More benchmark evaluations would be required to prove definitively that this is the relationhip.



Average Number of Bit Flips: Above is the average number of bit flips the algorithm used during the flip heuristic segment of its code for each of our benchmarks. Clearly, the more variables a problem had, the more bitflips it required during execution of the algorithm.

For UF-100 and UF-75, tests with lower time limits yielded on average less bitflips. This is due to the fact that many of these runs did not run until they found a satisfiable state, but were instead cut off by their time limites. This effect is noticable when compared with our UF-50 and UF-20 runs which did not have a correlation between time limit and average bitflips.

In general, as the number of variables in the problem increased the number of bitflips that were required to find a satisfiable solution (or until time ran out) greatly increased.



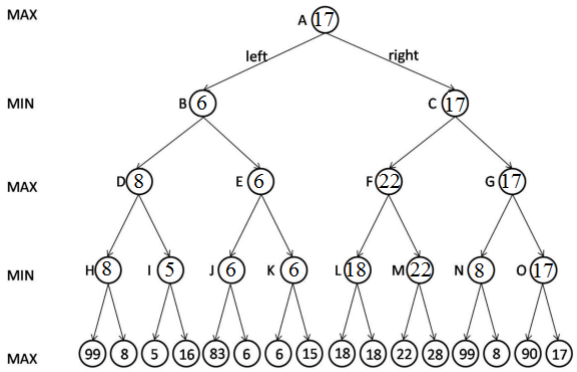
Success Percentage: Above is a chart displaying the percentage across all of our benchmarks that a satisfiable solution for the 3-CNF problem was found.For our UF-20 and UF-50 benchmark evaluations, for all given time limits (250ms, 500ms, and 750ms) every benchmarks that was ran was able to find a satisfiable solution.

As the number of variables in the problem increased however we see that more runs began to reach their time limits. For our UF-75 tests, more than 90% of benchmark evaluations were able to find a satisfiable solution with both a 750ms and 500ms time limit. However, decreasing the time limit to 250ms we began to see our first real drop in percentage successful down to 78%.

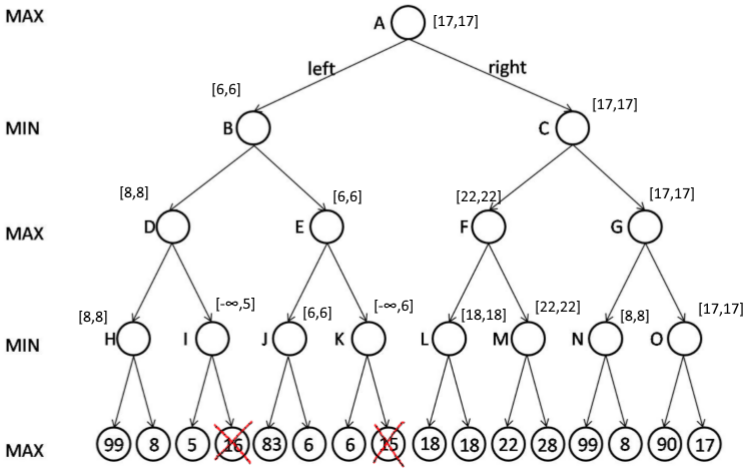
The success rate was worse for our 100 variable benchmarks tests. Naturally, as we reduced the alotted time, less of these runs were able to find a satisfiable solution. Our worst success rate was for UF-100 with an alotted time of 250ms. Here, only 36% of all runs managed to find a satisfiable solution before the time limit. Clearly the results of this section show that as the number of variables in the 3-CNF problem increases, the more time is necessary for the algorithm to find a satisfiable solution.

Problem 2

A. Below is the minimax search tree with each intermediate value throughout the operation of the algorithm labelled.



B. Below is a representation of which nodes are not visited throughout operation of the minimax algorithm with alpha-beta pruning and the final alpha/beta values at each intermediate node throughout its operation labelled.



The first node that will not be visited by the minimax algorithm using alpha-beta pruning is the terminal node with value 16 that is the right child of node I. At this point in the operation of the algorithm node D has discovered that it can choose a value of 8 from node H. This means that the range of values node D will possibly choose are $[8, \infty]$ (since D is a MAX node and will not accept anything less than 8 since it knows it can receive that from H). When we visit the left child of node I we know that the range of values node I will accept is $[-\infty, 5]$ (since node I is a MIN and will not choose anything larger than the 5 it knows it can get from its left child).

Since node I will offer a maxmium value of 5 to node D, we do not need to visit node I's right child because node D as a MAX node will already know that choosing the value at node H is preferable.

The second node that will not be visisted by the minimax algorithm with alpha-beta pruning is the right child of node K. Assume we are at the step where the MIN at node B knows it can get a value of 8 from node D. After node J's children are visited node E knows that it can get a value of 6 from node J. This means node E's value will be ≥ 6. The MIN at node B knows that it can possibly get less from node E than it can from D, so node E's right subtree is visisted. Once the MIN at node K knows it will return a maximum value of 6, the MAX at node E knows it should choose the 6 from node J. Thus the MAX player at node E can safely choose node J without ever visiting the right terminal child of K.

C. According to the exhaustive minimax algorithm the MAX player will choose 17 at the root state. According to the minimax algorithm employing alpga-beta pruning the MAX player will again choose 17 at the root state. In general the best move computed by the two algorithms is guaranteed to be the same. This is due to the fact that exhaustive minimax and minimax employing alpha-beta pruning employ the same logic at each state. It is simply that employing alpha-beta pruning reduces the number of computations required by the algorithm, it does not change the underlying logic. Therefore both algorithms will yield the same best move to the MAX player at the root state, but minimax employing alpha-beta pruning will visit less nodes to come to this conclusion and therefore be more efficient.

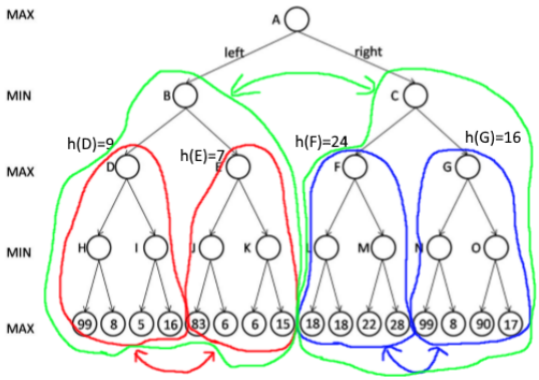
D. In this version of minimax we are using alpha-beta pruning as well as guiding heuristic values. The heuristic values that were given in the problem were as follows: $h(D) = 9$ $h(E) = 7$ $h(F) = 24$ $h(G) = 16$.

Because the heuristic value for node E is lower than the heuristic value for node D, and since their parent node B is a MIN, the subtrees with root D and root E will be swapped. This is because, based on the heuristic values, the algorithm would want to search the subtree of node E first, since B is a minimizer and would would choose the minimal value, it is more likely that we can eliminate nodes that we need to search if we search this subtree first.

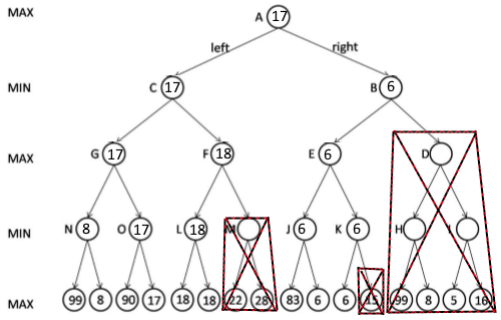
The same is true for nodes F ang G. Node G has a lower heuristic value and since their parent node C is a MIN node, it will choose the minimal value of the two children nodes. Since the heuristic is informing us that G is most likely the node with the minimal value, we will expand G first to see if there are any nodes we can possibly eliminate from our search process.

Node A is a MAX. Based on the information provided by the heuristic we can determine which subtree, B or C, would be better to search first. Since B is a MIN, and we know that $h(D) = 9$ and $h(E) = 7$ we can assume that node B would choose node E. The same logic can be applied to see that node C will choose node G. Now we know that based on the heuristic node B has a value of 7 and node C has a value of 16. Since A is a MAX it should expand node C first, so B and C are swapped.

The specified swaps are shown below:



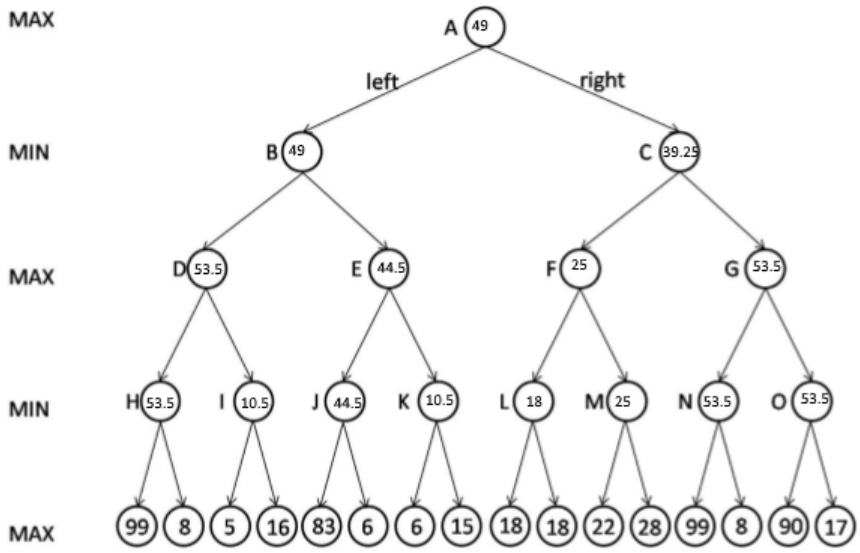
After these swaps are made the minimax algorithm with alpha-beta pruning will run in its normal operation and find that the following nodes will not need to be expanded:



Clearly, using the information provided by the heuristic values, the minimax algorithm using alpha-beta pruning will find that it can eliminate more nodes that its uninformed counterpart and still find an optimal solution.

E. Consider a variation where the MIN player will choose either the left or right child state as its move with a 0.50 probability. Here, the MAX player is uncertain what node to choose because it cannot predict the choice of the MIN player based on the minimum value available to it (as in a typical minimax search tree). However, the MAX player can choose a value in the search tree based on the expected value of the MIN players choice.

   For example, if a MIN player has available to it a node with a utility value of 5 and a node with a utility value of 15, based on this problem we know that it will choose 5 with probability 0.50 and 15 with probability 0.50. Therefore we can say the expected value of the MIN players choice is 10. For each MIN node, we can say that its expected value is $0.50 * \textit{value of left child} + 0.50 * \textit{value of right child}$. The MAX player can then maximize its choice by selecting the MIN node with the higher expected value, and in this way we utilize all of the information available to our MAX player to optimize his choice based on the randomness of the MIN player.



Above is a visualization of the described algorithm where the MIN player stores in its node its expected value from choosing from its two children each with 0.50 probability. The MAX player will then choose the MIN node with the higher expected value.

Problem 3

A. **Variables:** Defined as $n_{i,j}$ where $i \in$ {set of all row numbers} and $j \in$ {set of all column numbers} for each cell n on the board. Essentially, each square on the sudoku board will be a variable, for a total of 81 variables.

Domain: Defined as the set of possible values in sudoku. Here, this set is defined as $\{1, 2, ..., 9\}$

Constraints:

1. All variables in any row must be assigned distinct values.
2. All variables in any column must be assigned distinct values.
3. All variables in any 3x3 region must be assigned distinct values.
4. The assignments given to variables in the start state cannot be changed.

B. **Start State:** The start state is a given board that has variables assigned with certain values. These values are the basis for the constraints for the rest of the problem and may not be changed. Below is an example start state.

	1		4	2				5
		2		7	1		3	9
							4	
2		7	1					6
				4				
6					7	4		3
	7							
1	2		7	3		5		
3				8	2		7	

Successor Function:

1. Select a random unassigned variable
2. Assign a value to this variable that does not violate any constraints
3. The resulting state is the successor

Goal Test: A state is considered to a valid goal state if it is both consistent and complete.

Path Cost Function: The path cost from any one state to its successor state will be 1.

C. Sudoku problems that are easy compared to ones that are hard can be determined based on the relative usefulness of applying the minimum remaining values heuristic to their search states.

Consider a Sudoku board for which there are an equal number of remaining values for each variable based on their constraints. Here, the heuristic will not be of much use to us since it will not help us narrow our search and we will have to backtrack from many more states to find a satisfiable goal state than if we were faced with an easier Sudoku board.

An easier Sudoku board can be defined as one for which there is a varied range of minimum remaining values for each variable. This will be an easier game because the minimum remaining values heuristic will direct us to expand states with the least minimum remaining values first. Since the distribution of minimum remaining values for an easy board is more varied, there will always be a good estimate of what state to expand next and this will lead us more directly to the goal state.

D. Pseudocode for Local Search

Algorithm 1 Sudoku Local Search Algorithm

```
1: procedure SudokuSolver(Board board)
2:     board ← randomizeCompleteState(board)
3:     while !completeAndSatisfied(board) do
4:         nextBoard ← updateState(board)
5:         if evaluateSatisifiability(nextBoard) > evaluateSatisifiability(board) then
6:             board ← nextBoard
7:     return board
```

Due to its random nature this problem will perform worse on easier problems and better on hard problems than its incremental formulation counterpart.

When local search using incremental formulation is used with a good heuristic it will be able to solve easier problems much faster because the heuristic will be very good at directing the search to the goal state. Compared to complete state formulation, which randomizes one variable in a complete state as a successor state, the incremental method will be much more directed and be able to find a solution much faster. Whereas because of the randomness of complete state it will most likely take longer to find a solution.

With harder Sudoku problems however, the incremental formulation will not find a solution as fast as the complete state formulation. The heuristic that the incremental formulation uses will not be very useful for hard problems because it will still have to search a lot of the search tree to make up for what cannot be informed by the heuristic. The complete state formulation however, since it changes states randomly, will not have this problem, and will probabilistically complete the algorithm in faster time than the incremental formulation.

Problem 4

For Superman to be defeated, it has to be that he is facing an opponent alone and his opponent is carrying Kryptonite. Acquiring Kryptonite, however, means that Batman has to coordinate with Lex Luthor and acquire it from him. If, however, Batman coordinates with Lex Luthor, this upsets

Wonder Woman, who will intervene and fight on the side of Superman.

S = Superman Defeated
A = Facing Opponent Alone
K = Opponent Carrying Kryptonite
B = Batman Coordinates with Lex Luthor
W = Wonder Woman Upset

A. $S \implies A \wedge K$

$K \implies B$

$B \implies W$

$W \implies \neg A$

B. $S \implies A \wedge K \equiv \neg S \vee (A \wedge K) \equiv (\neg S \vee A) \wedge (\neg S \vee K)$

$K \implies B \equiv \neg K \vee B$

$B \implies W \equiv \neg B \vee W$

$W \implies \neg A \equiv \neg W \vee \neg A$

C. In order to prove that superman cannot be defeated we must use our knowledge base to prove that superman defeated is unsatisfiable.

Given: $KB = (\neg S \vee A) \wedge (\neg S \vee K) \wedge (\neg K \vee B) \wedge (\neg B \vee W) \wedge (\neg W \vee \neg A)$

Prove: $KB \vDash \neg S$

Show: $KB \wedge S$ is unsatisfiable

$KB \wedge S = \underline{(\neg S \vee A)} \wedge (\neg S \vee K) \wedge (\neg K \vee B) \wedge (\neg B \vee W) \wedge \underline{(\neg W \vee \neg A)} \wedge S$

$KB \wedge S = \underline{(\neg S \vee \neg W)} \wedge (\neg S \vee K) \wedge (\neg K \vee B) \wedge \underline{(\neg B \vee W)} \wedge S$

$KB \wedge S = \underline{(\neg S \vee \neg B)} \wedge (\neg S \vee K) \wedge \underline{(\neg K \vee B)} \wedge S$

$KB \wedge S = \underline{(\neg S \vee \neg K)} \wedge \underline{(\neg S \vee K)} \wedge S$

$KB \wedge S = (\neg S \vee \neg S) \wedge S$

$KB \wedge S = \neg S \wedge S$

$KB \wedge S = \varnothing$

Above we can see that $KB \wedge S$ reduces to an empty set and is thus unsatisfiable $\therefore KB \vDash \neg S$. ■

Problem 5

A. Travelling Salesman Problem using A* search.

B. For this problem we used the local search method Simulated Annealing to solve the Travelling Salesman Problem.

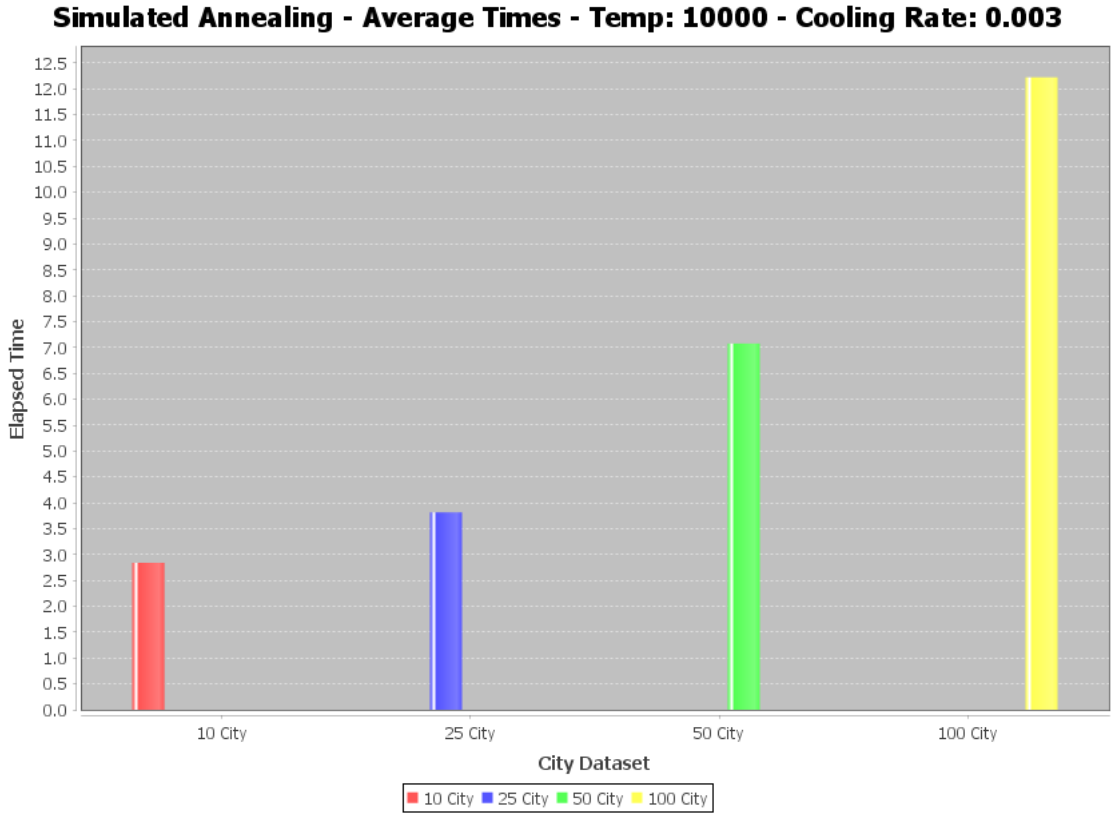
We stored the initial randomized complete state path as an array of City objects of size *numCities*+ 1 with the first element in the array being the start city and the last element in the array being the goal city. Both of these cities were the first city in the data file (this would be the travelling salesman's start point and end point).

During each randomization of a new state we randomly flipped two cities in the path (that were not the start or goal city) and returned that new state to our algorithm. This method will make sure that the next state is both a complete state and does not change the travelling salesman's start or end points.

Based on the parameters of the simulated annealing algorithm this new state is either accepted because it improves upon the previous state, or with probability $e^{\frac{\Delta currentFitness\ -\ nextFitness}{temperature}}$. This process is repeated and the temperature is lowered based on the cooling rate. Once the temperature reaches below 1 the algorithm is considered complete and the path found is returned.

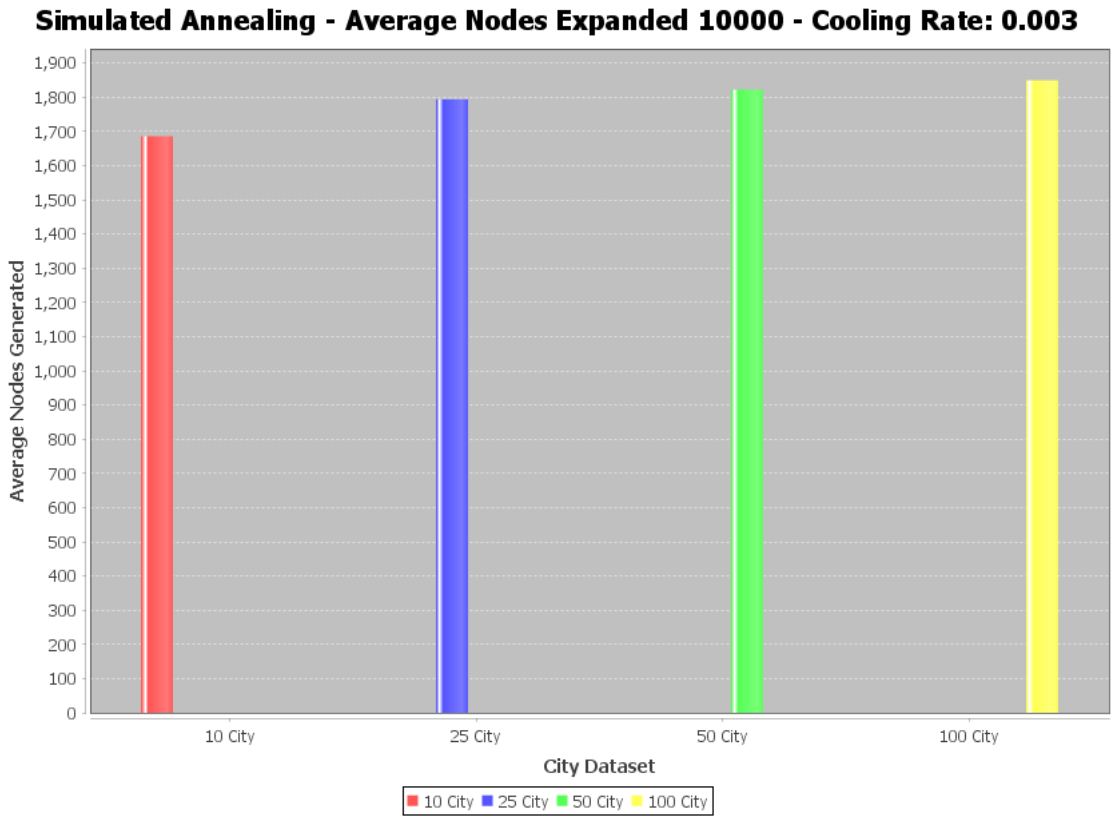
Using simulated annealing we were able to find a solution for each set of cities however if we set the cooling rate too low we would easily run out of memory (this happened at about a cooling rate on the order of 10^{-6}).

Below is the chart for the average solution time for each benchmark set of cities using a temperature of 10,000 and a cooling rate of 0.003.

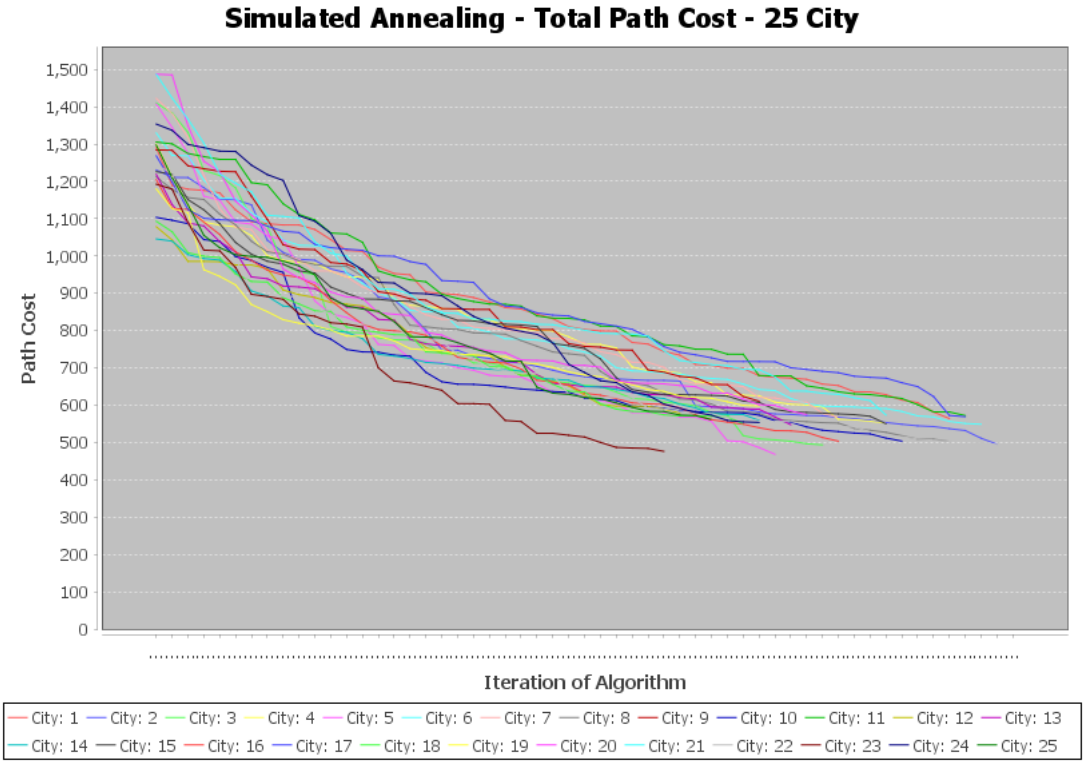


Generally as the number of cities included in the problem increased the time it took to find a solution also increased.

Below is the number of nodes generated across all benchmark evaluations:



Overall the number of nodes expanded seemed to be relatively constant across each of the benchmark evaluations. This is mostly because it doesn't really matter how large the cities are for simulated annealing in this regard. Each successive state still only flips two cities on the path so regardless of how large the city-groups are finding the next successive state is very similar.



Above is an example of the path cost for each of our 25 city benchmark evaluations during each iteration of the simulated annealing algorithm. In the beginning iterations, when the temperature is very high there are bery steep drops in path cost. This is because the algorithm is more likely at this point to take random walks then to only select better states. This can either increase or decrease the total path cost but has the potential to take the solution across a plateau or over a local extrema. Towards the end of execution we see that the path costs begin to level off as the algorithm is less likely to accept worse states and only chooses states that improve the current state. Since at this point the current state is very close to its final solution it is likely that we no longer have to worry about plateaus and local extrema (since the random walks mixed with hill-climbing in the simulated annealing algorithm would get us around these probabilistically) the algorithm slowly hill-climbs to the probabilistic complete solution.

C. Simulated annealing is a useful technique for problems that contain a lot of plateaus and local extrema in the evaluation of their objective functions. Consider the problems associated with hill-climbing in these scenarios. The hill-climbing methods will get 'stuck' in a local extrema or not be able to traverse a plateau without random walks. This leads to sub-optimal solutions unless we randomly restart hill-climbing which is inefficient.

   Whereas hill-climbing offers no real solution to these issues, simulated annealing is an elegant probabilistically complete solution that tackles these issues in a balanced way. In the beginning the algorithm is very likely to accept successor states with low objective function evaluations than its current state. This means that there will be more random walks in the beginning of simulated annealing which will allow local extrema to be overcome.

   This lenient acceptance of "worse" states slowly tapers off as the temperature decreases until we move into the more hill-climbing stage that only accepts better solutions. Because of the randomness of this method it is only probabilistically complete but compared to hill-climbing in problems with many local extrema it is almost certainly guaranteed to perform better in finding a better solution.

It is implicit in the design of simulated annealing that the value of the objective function has many local extrema that need to be overcame which would cause algorithms like hill-climbing to fail to find a useful solution. The leniency in accepting a new state in the beginning of the algorithm allows us to overcome these issues. However if these issues are not present then simulated annealing is a worse alternative as compared to hill climbing because it will accept worse states whereas hill-climbing would guide us directly to the optimal solution.

Problem 6

A. Using the minimum value of $h_1(n)$ and $h_2(n)$ for each state:

$$\text{Given that } h_1(n) \leq h^*(n) \text{ is always true,}$$
$$Min(h_1(n), \infty) \leq h^*(n)$$
$$\text{Therefore } h_1(n) = minimum(h_1(n), h_2(n)) \text{ is admissible.}$$
$$\text{consistent?}$$

B. Using the maximum value of $h_1(n)$ and $h_2(n)$ for each state:

$$\text{Given that } h_1(n) \leq h^*(n)$$
$$h_2(n) \leq h^*(n)$$
$$Max(h_1(n), h_2(n)) \leq h^*(n)$$
$$\text{Therefore } h_4(n) = maximum(h_1(n), h_2(n)) \text{ is admissible.}$$
$$\text{consistent?}$$

C. The defined heuristic function

$$h_3(n) = w \times h_1(n) + (1 - w)h_2(n), \text{ where } 0 \leq w \leq 1$$

is admissible only when w is a value less than 0.5, any value larger than 0.5 will cause $h_1(n)$ to be multiplied by a value larger than $h_2(n)$. Since the the bounds are from 0 to 1 the heuristic is NOT admissible.

consistent?

D. Considering the informed, best-first search algorithm with an object function of $f(n) = (2-w) \times g(n) + w \times h(n)$. It is guaranteed to work for $0 \leq w \leq 1$ since being multiplied by g(n) which is a constant implies no effect on which order chosen paths are arranged. However, if $w > 1$, then the goal state may have an overestimated distance which will make the heuristic not admissible and so not optimal.

Given the information above we can assess the function for when the values of w are 0, 1, and 2.

$w = 0$:

When $w = 0$ it will make $f(n) = 2g(n)$ and causes the algorithm to perform like a Uniform Cost Search since there is no weight assigned it will find the most optimal path however, it will not be efficient.

$w = 1$:

When $w = 1$ it will make $f(n) = g(n) + h(n)$ which causes the algorithm to perform like A^* and as with $w = 0$ is guaranteed to find the optimal path however, is more efficient.

$w = 2$:

When $w = 2$ it will make $f(n) = 2h(n)$ which causes the algorithm to perform like Greedy Best first search.