# CS 486 - Assignment 2

Alex Klen
20372654

January 30, 2015

# 1 Written Questions

a.) The algorithm will explore the states in the following order: $s, h, k, c, a, b, d, m, e, n, g$.

b.) i.) $h$ is admissible because it satisfies the following two criteria. $h(g) = 0$ because the goal is zero steps from itself. For Manhattan Distance, $h(n) \leq h^*(n)$ (a heuristic will give a cost from node $n$ to the goal of at most the cost of the shortest path) because the Manhattan Distance gives the shortest possible number of moves. This is proven below.
Suppose we have a path $p$ from $n$ to $g$ where $|p| < h(n)$. $p$ needs to have at least $dx = |g.x - n.x|$ horizontal moves and at least $dy = |g.y - n.y|$ vertical moves, so $|p| \geq dx + dy$. But Manhattan Distance is exactly this quantity: $h(n) = dx + dy$. Then $|p| \geq dx + dy = h(n)$, a contradiction. Therefore, $h$ is an admissible heuristic function.

ii.) The algorithm will explore the states in the following order: $s, h, k, c, a, b, d, m, g$.

iii.) The algorithm will explore the states in the following order: $s, h, k, c, f, p, q, r, t, g$.
Note that the algorithm might not explore $c$ , since at that step $c$ and $f$ have the same cost plus heuristic value.

# 2 Programming Questions

A. **Informed Search**

a. A state is a partial tour, represented by AStarNode in my implementation. In this case the path A* takes isn't needed to compute the cost of each state, because this information is stored in the state. The reason states need this information is because the heuristic needs to know which cities the tour has visited so far, so a state needs to include the partial tour.

The initial state is simply a partial tour of only city A.

Goal states are any states that has a tour of every city - the length of the tour is the number of cities in the problem.

The operator to get to new states is simply to choose each city not in the current state's tour and append it to the tour.

The cost of a state is simply the sum of distances of consecutive cities. If the tour is partial, the cost does not include a return edge to the start node. If it is complete, however, it is the full cost of the cycle.

b. My heuristic function for an incomplete tour is as follows:

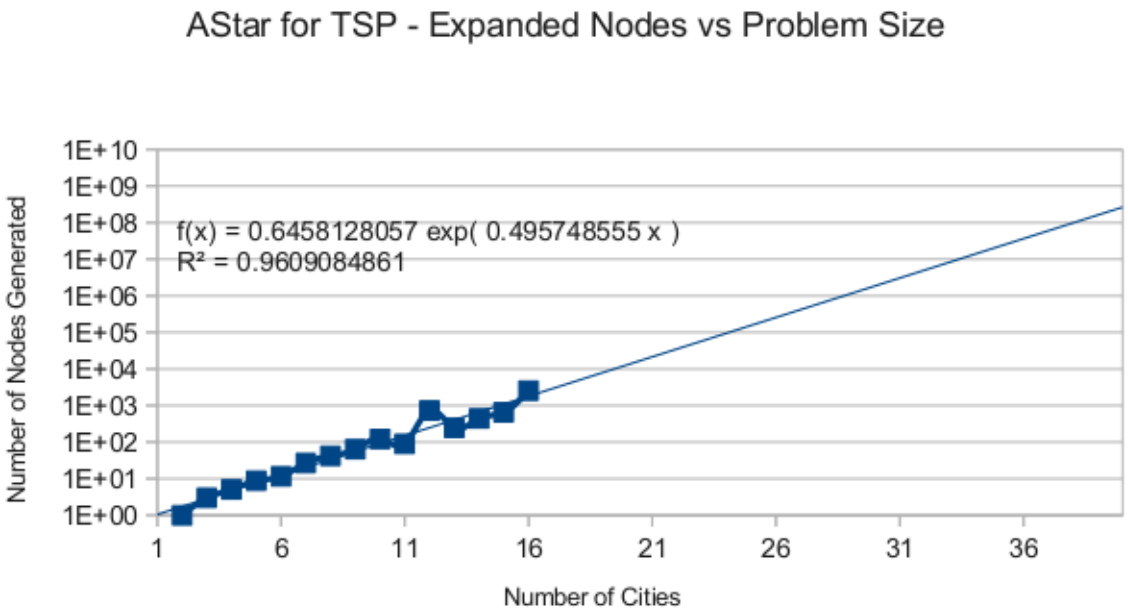
$$\begin{aligned}&\text{Let path} = [p_1, ..., p_k]\\&\text{Let unvisited} = \{\text{cities}\} - \{\text{path}\}\\&h(\text{path}) = \text{dist}(\text{MST}(\text{unvisited})) + \min_{n \in \text{unvisited}} \text{dist}(p_k, n) + \min_{n \in \text{unvisited}} \text{dist}(n, p_1)\end{aligned}$$

The heuristic function is 0 for a complete tour. I did not come up with this heuristic, I found it on page 26 of *Heuristic Search: Theory and Applications* By Stefan Edelkamp, Stefan Schroedl.
It is admissible because of the following observation. The optimal tour $T^* = [p_1..p_k, u_1, u_l, p_1]$ has some path through the remaining unvisited nodes $u_1..u_l$, with the edges $(p_k, u_1)$ and $(u_l, p_1)$. The path $[u_1..u_l]$ is a spanning tree of the unvisited nodes. By definition the minimum spanning tree (MST)

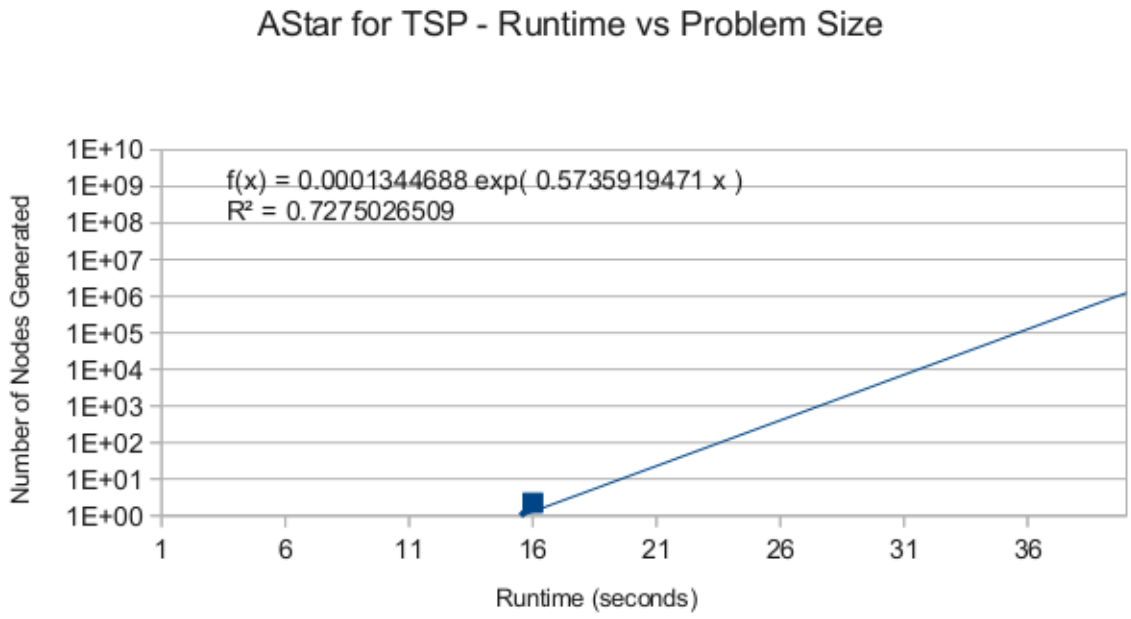
has sum of edge lengths less than or equal to any spanning tree.

$$
\begin{aligned}
\text{dist}\,(T^*) &= \text{dist}\,(p_1..p_k, u_1..u_l, p_1) \\
&= \text{dist}\,(p_1..p_k) + \text{dist}\,(p_k, u_1) + \text{dist}\,(u_1..u_l) + \text{dist}\,(u_l, p_1) \\
&\leq \text{dist}\,(p_1..p_k) + \min_{n \in \text{unvisited}} \text{dist}\,(p_k, n) + \text{dist}\,(\text{MST}\,(\text{unvisited})) + \min_{n \in \text{unvisited}} \text{dist}\,(n, p_1) \\
&= \text{dist}\,(p_1..p_k) + h\,(\text{path}) \\
\therefore\ \text{dist}\,(T^*) - \text{dist}\,(p_1..p_k) &\leq h\,(\text{path})
\end{aligned}
$$

c. The following graph shows the number of A* states created and placed in the priority queue during the search averaged over all 10 given TSPs. The vertical axis uses a logarithmic scale and an exponential trend line is shown.




The trend line shows the approximate number of states generated for a 36-city TSP, which is 36.3 million.
The chart below shows the averaged running times for the same runs.




The trend line shows the approximate runtime a 36-city TSP would be $\approx$ 35 hours.

**A* Code**

My code is implemented in Haskell and included in separate files with this report.
See the included files *Search.hs* for the generic searching algorithm, *TSP.hs* for my TSP representation, and *Main.hs* for the main program entry point.
Then see *AStar.hs* for a generic A* implementation and *TSPAStar.hs* for the code that applies it to TSP.

B. **Local Search**

a. My neighbourhood operator is to swap the endpoints of two edges in an existing tour. This is the same as reversing a range in a permutation of cities. More concretely, the neighbours for $T = [p_1..p_n, p_1]$ are defined as follows:

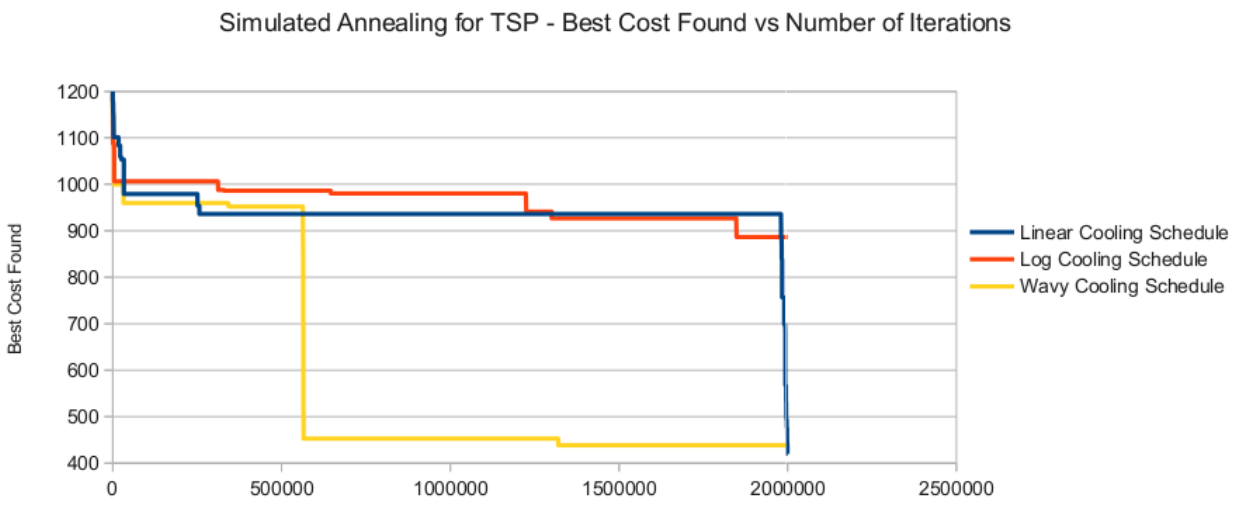
$$\begin{aligned}&\text{neighbours} = \{\}\\&\text{for } i <- [1..n-3]\\&\ \text{for } j <- [i+1..n-2]\\&\ \ \text{neighbours} = \text{neighbours} \cup \{[p_1..p_{i-1}, p_j, p_{j-1}..p_{i+1}, p_i, p_{j+1}..p_n]\}\end{aligned}$$

b. The three cooling schedules I used are shown below. These are scaled to run for 1000 time steps.

1. Linear: $5 * (1000 - x)$
2. Logarithmic: $5 * (300 * \log(-x/100 + 30))$
3. Oscillatory: $5 * (501 - x/4 + (1000 - x/2)/2 * \cos(x/5) * \sin(x/12))$

I think using some form of oscillating schedule would be best, although the formula above isn't necessarily a great one. The idea is that the algorithm will be in an exploitation phase when the temperature drops and it will find a local minimum, and then as the temperature rises it will be in an exploration phase where it can escape valleys and pass into other ones.

c.




The cost of the best solution found was 420.07. One thing to note is that the linear cooling schedule quickly finds a local minimum and then never finds a better solution after that because it cannot escape a local minimum. The logarithmic schedule is at a higher temperature for a longer period of time and so makes progress later than the linear one, and the oscillatory schedule makes some progress even later.

d. Simulated Annealing is complete, assuming that you start at a goal state and your operator moves to other valid goal states. Since it only looks at goal stats it will output a goal state after running, but it may not be optimal.

e. Simulated Annealing is not optimal because it isn't guaranteed to find an optimal goal state. Its performance depends on the cooling schedule, the local search operator, and randomness. It might get caught in a local minima and output a sub-optimal solution.

**Simulated Annealing Code**

See *SimulatedAnnealing.hs* for a generic Simulated Annealing implementation *TSPSimulatedAnnealing.hs* for the code that applies it to solve TSP.