CS360 – Homework #6

Search-Based Planning

1) Consider the formulation of the Tower of Hanoi puzzle as a planning problem (solution of Homework 4, problem 2). Using the delete relaxation, calculate the length of the action sequence for achieving each predicate that appears in the goal state from the start state, then calculate the heuristic value of the start state as the maximum of these values.

There is only one operator, namely, `Move(X, Y, Z)`. With the delete relaxation, we obtain:

```
Action Move(X,Y,Z):
Preconditions = {Clear(X), On(X,Y), Clear(Z), Smaller(X,Z)};
Effects = {Clear(Y), On(X,Z)};
```

We use the following recursion to compute $h_s(p)$ for all propositions p that appear in the goal state (Slide 6 of the Search-Based planning slide set):

h_s(set of predicates s') $= max_{predicate\ p\ in\ s'} h_s(p)$.
h_s(predicate p) $= 0$ if p appears in the state s,
h_s(predicate p) $= 1 + min_{relaxed\ operator\ o\ with\ p\ in\ add\ list} h_s$(precondition list of o), otherwise.

That is, if a proposition p appears in the start state s, $h_s(p) = 0$. The following propositions appear in the start state (for brevity, we skip the propositions of the form `Smaller(X,Y)`):

```
Clear(D1), On(D1, D2), On(D2, D3), On(D3, P1),
Clear(P2), Clear(P3).
```

Let P_i denote the set of propositions p with $h_s(p) = i$. That is, P_0 contains exactly the propositions in the start state. Note that P_0 contains `On(D1, D2)` and `On(D2, D3)`, but not `On(D3, D3)`. We now incrementally compute P_1, P_2, . . . , until we discover the P_i that contains `On(D3, D3)`.

We start with P_1. This is the set of propositions that can be added by an operator whose preconditions are satisfied by P_0. There are only two operators whose preconditions are satisfied by the propositions in P_0, namely, `Move(D1, D2, P2)` and `Move(D1, D2, P3)`. The add effects of these two operators have the propositions `On(D1, P2)`, `On(D1, P3)`, `Clear(D2)`. Since none of them are in P_0 and all preconditions of the operators that add them are in P_0, all of them must be in P_1 (because of the recursive rules listed above). Therefore, $P_1 = \{\texttt{On(D1,P2)}, \texttt{On(D1,P3)}, \texttt{Clear(D2)}\}$. Below, we list $P_0 \cup P_1$.

```
Clear(D1), On(D1, D2), On(D1, P2), On(D1, P3),
Clear(D2), On(D2, D3), On(D3, P1),
Clear(P2), Clear(P3).
```

We now identify P_2. This is the set of propositions that can be added by an operator whose preconditions are satisfied by $P_0 \cup P_1$. There are two new operators whose preconditions are satisfied by $P_0 \cup P_1$, namely `Move(D2, D3, P2)` and `Move(D2, D3, P3)`, which add the propositions `On(D2, P2), On(D2, P3), Clear(D3)` (note that, at this point, any operator that moves D1 does not add any new propositions, so we skip those operators). These propositions do not appear in $P_0 \cup P_1$ but can be added by operators whose preconditions are in $P_0 \cup P_1$. Therefore, $P_2 = \{\texttt{On(D2,P2)}, \texttt{On(D2,P3)}, \texttt{Clear(D3)}\}$. Below, we list $P_0 \cup P_1 \cup P_2$.

```
Clear(D1), On(D1, D2), On(D1, P2), On(D1, P3),
Clear(D2), On(D2, D3), On(D2, P2), On(D2, P3),
Clear(D3), On(D3, P1),
Clear(P2), Clear(P3).
```

$P_0 \cup P_1 \cup P_2$ satisfy the preconditions of `Move(D3, P1, P3)`, which adds `On(D3, P3)`. Therefore, `On(D3, P3)` $\in P_3$ and, consequently $h_s(\texttt{On(D3,P3)} = 3$.

Taking the maximum of 0, 0 and 3, we obtain 3 as the heuristic value of the start state.

Local Search

2) In the N-Queens problem, we want to place N queens on an $N \times N$ board with no two queens on the same row, column, or diagonal. Come up with a value function and use hill climbing to try to solve the problem by minimizing this value function, starting with the configuration given below. Generate the successors of a state by moving a single queen vertically.

	A	B	C	D
1				
2	Q1		Q3	
3		Q2		Q4
4				

The value function is the number of pairs of queens that can attack each other, assuming no other queens are on the board (that is, three queens in the same row count as three pairs, not two). In the initial state, we have the pairs (Q1, Q2), (Q1, Q3), (Q2, Q3), (Q2, Q4), (Q3, Q4), making its value 5.

Moving Q1 to A1 removes the pairs (Q1, A2) and (Q1, Q3), decreasing the value by 2. Moving it to A3 removes the pair (Q1, Q3), but adds the pair (Q1, Q4). Moving it to A4 does not change the pairs. A symmetric argument can be made for Q4.

Moving Q2 to B1 does not change the pairs. Moving it to B2 removes the pair (Q2, Q4). Moving it to B4 removes all three pairs that Q2 appears in. A symmetric argument can be made for Q3.

The lowest value we can get with one move is 2, either by moving Q2 to B4 or Q3 to C1. Moving Q2 to B4, we get:

	A	B	C	D
1				
2	Q1		Q3	
3				Q4
4		Q2		

There are many possible moves in this state, but moving Q3 to C1 finds the solution (that is, a state with the lowest possible value of 0), so we can skip the other moves:

	A	B	C	D
1			Q3	
2	Q1			
3				Q4
4		Q2		

3) How would you approach the Traveling Salesman Problem if we wanted to find a good (but not necessarily the best) solution to it using hill climbing? Explain also how A* search can be used for this purpose.

We start with a random solution that visits all cities. The value function is the total distance traveled by the salesman. We can generate the successors of a state by picking any two cities on the tour and switching the order in which they are visited. Another method of generating successors might be to pick any subpath of the tour and invert the order the cities that are visited in this subpath. Finally, the 2-opt and 3-opt algorithms are more complex but very well performing hill-climbing algorithms for the Traveling Salesman Problem.

For A*, each state of the search corresponds to a path that visits a subset of the cities. Since the starting location of the salesman is not important, we can arbitrarily choose a city A for their start location, and the start state contains the degenerate path (A), that is the zero-cost path from A to A. The successors of a state are generated by finding the next city to visit, given all the cities visited so far. The cost of adding a city to the path is the distance from the last city on the path to the new city. When we are completing the tour by adding the only remaining city to the path, we also add to the cost the distance between this only remaining city to city A. The set of goal states contains all tours, that is, paths that start in A, visit all cities once, and then return to A.

4) What are the advantages/disadvantages of local search methods (such as hill climbing and simulated annealing) compared to A*? For which kind of optimization problems should local search be preferred?

Local search methods are typically much faster than A*, although they do not provide any optimality guarantees. They also use little memory, since they only need store a single state. Local search methods are best applied to computationally hard (e.g. NP-hard) optimization problems. One thing to note is that local search algorithms search in the space of candidate solutions. For instance, if we apply local search to find short paths on graphs, we need to generate an initial path (possibly by searching the graph with depth-first search or even suboptimal versions of A* search), and then start optimizing this path. In this scenario, local search is a poor substitute for A* search.