COMP 440 Homework 1

Tony Chen(xc12) and Adam Wang(sw33)

August 2016

1 Modeling sentence segmentation as search

- Suppose the sentence length is L, we define the state space model as following:
  - $S = s_0 \cup s_1 \cup s_2 \cup ... \cup s_L$, where, for $i = 0, 1, 2, ...L$, s_i represents the state that exactly i first characters have been segmented into words.
  - *Action* is defined as: for every word w in D that is a prefix of the substring that starts at the i-th character in the sentence, $a_{s_i,w}$ is the action to jump to state $s_{s_i+len(w)}$.
  - $Successor(s_i, a_{i,w}) = s_{i+len(w)}$
  - $Cost(s_i, a_{i,w}) = 1$
  - $s_{start} = s_0$
  - $isGoal(s_i) = (s_i == s_L)$
- *Since the paths of unweighted graph of states are directed and never going "backward", there's no cycle in the graph.
  - Yes. BFS will be able to traverse the graph layer by layer from s_0 and once it reaches s_L, the current distance (number of layers traversed) is the minimum cost path. Time complexity will be $O(b^s)$ where b is the average number of neighbours and s is the number of layers from s_0 to s_L.
  - No. DFS will not guarantee the distance we have when we find s_L to be the minimum cost path.
  - Yes. UFS will be able to find a least cost path from s_0 to s_L in $O(L)$.
  - Yes. A* with a consistent heuristic will be able to find a least cost path from s_0 to s_L in $O(L)$.
  - Yes. Bellman-Ford will be able to find a least cost path from s_0 to s_L in $O(L^2|A|)$, where $|A|$ is the number of edges.
- Modify *Cost* function so that $Cost(s_i, a_{i,w}) = len(w) - 1$.
  BFS won't work now since it only works on unweighted graph. UFS, A*, and Bellman-Ford will still work since they all can generalize on weighted graph.
- Modify S so that for a state $s_{i,last}$ where $i > 0$, exactly i first characters have been segmented and the last segmented word is *last*.
  Modify *Cost* function so that $Cost(s_0, a_{0,w}) = 0$ and $Cost(s_{i,last}, a_{i,w}) = fluency(last, w)$.

2 Searchable Maps

- We define the cost as the time required to travel from s to t. Since we need a heuristic that never overestimates the cost, we define it as the lowest possible cost, which is obtained by a traveling along a straight line from s to t at the highest possible speed:

$$h(s,t) = \frac{G(s,t)}{S_H}$$

- The heuristic is defined as following:

$$h(s,t) = |T(s,L) - T(L,t)|$$

  Proof: Suppose there is a neighbour node n, then we need to prove that $h(s,t) \le Cost(s,n) + h(n,t)$. Consider all sign combinations of $T(s,L) - T(L,t)$ and $T(n,L) - T(L,t)$ with the fact that $T(s,L) \le Cost(s,n) + T(n,L)$ (triangular inequality), we see that $h(s,t) \le Cost(s,n) + h(n,t)$ holds at all time.

- Suppose the goal node is t, it is trivial that $h(t) = h_1(t) = h_2(t) = 0$. Consider any pair of node n and m where there is an action to get m from n. Suppose $h_1(n) \ge h_2(n)$, then $h(n) = h_1(n)$ and there are two possibilities: first, $h_1(m) \ge h_2(m)$, which makes $h(m) = h_1(m)$ and obviously makes h consistent on n and m (h is exactly h_1); second, $h_1(m) < h_2(m)$, then since $h(n) = h_1(n) \le Cost(n,m) + h_1(m) < Cost(n,m) + h_2(m)$, h is also consistent on n and m. So h will always be consistent in this case
  Since h_1 and h_2 are symmetric, the above conclusion will also hold for $h_1(n) < h_2(n)$. So h is always consistent.

- According to part c, the basic idea is to take the max of all heuristics:

$$h(s,t) = max(|T(s,L_1) - T(L_1,t)|, |T(s,L_2) - T(L_2,t)|, ..., |T(s,L_K) - T(L_K,t)|, \frac{G(s,t)}{S_H})$$

- For adding edges, h will NOT remain consistent. Imagine the new edge draws a straight line from s to t, then as long as the original path estimate of h is not a straight line on its own, h will overestimate, which makes it not only inconsistent but also not admissible.
  For removing edges, h will still remain consistent. Since for any edge remains in the graph, none of the three parts in the inequality $h(m) \le Cost(m,n) + h(n)$ will change. So h will remain consistent.

3 Package Delivery as Search

- 2a
  - States: Location (x, y) of the truck, a vector S of length $numPackages$ where $S[i] \in \{-1, 0, 1\}$ where -1 denotes package i has not been picked up, 0 denotes package i has been picked up and not dropped off and 1 denotes package i has been dropped off.
  - Actions: North, East, West, South, Pickup and Dropoff
  - Successor: If action is North, East, West or South, update location respectively. If action is Pickup, in the vector in state, change from -1 to 0 for the corresponding package. If action is Drop-off, in the vector in state, change from 0 to 1 for the corresponding package.
  - Cost: If action is North, East, West or South, cost is 1 plus number of packages carried If action is Pickup or Dropoff, cost is 0.
  - Initial state: Truck's starting location and a vector of -1 of length number of packages to pickup.
  - Goal test: If truck's location is the starting location and if the vector in the state are all 1s. Number of states: $m * n * 3^k$.
- 2c. Number of states explored: 56
- 2d. Number of states explored: 34
- 2e. Number of states explored: 56

4 Designing Search Algorithms: Protein Folding

Literature referenced[1]

- Suppose the input sequence s has L acids. S contains $s_{directions}$ which is a length-L ordered list, of which each element l_i is the direction from the i-th acid to the $i+1$-th acid. Each direction is one of $N(0,1)$, $S(0,-1)$, $W(-1,0)$, or $E(1,0)$. There is a constraint that there must exist no pair (i,j) such that $\sum_{k=i}^{j} l_k = (0,0)$ (self-avoiding).

  s_0 is a list of L randomly generated directions that satisfies the self-avoiding constriant.

  s_{Goal} is achieved when $Cost(s)$ becomes smaller than a fixed threshold.

  A contains $a_{i,dir}$, which changes the direction of the i-th acid to dir if such change will not break the self-avoiding constraint.

  $Succ(s_{directions}, a_{i,dir}) = s_{directions'}$,
  where $directions'[i] = dir$ and $directions'[j] = directions[j], \forall j \neq i$

  $Cost(s_{directions}, a_{i,dir})$ is the difference between sum of all H-H pairs of acid locations generated by $directions$ and that of $directions'$.

- Local search algorithm.
  Because we only care about the final conformation of the protein, not how we change the sequence topologically to achieve that.
  Also because the search space is too large to run algorithms that search for global optimum.

- Function ProteinFolding
  Input: protein sequence in its string form

```
    $Population <- 200$ randomly generated conformation
    Iterate for 200 times:
        sort $Population$ based on their costs in ascending order
        retain the first half of $Population$
        replace the second half of $Population$ with children of randomly chosen parents by a 2-point crossover with rate 0.2-0.8.
        randomly mutate each new-born conformation at chance 0.4, by randomly altering one of its direction.
    return the conformation with the lowest cost
```

  Time Complexity: $O(l * m * n(log(n))$, where l is the length of the input sequence, m is the iteration limits, and n is the population size. ($nlog(n)$ because of the sort)
  Space Complexity is similar to time complexity: $O(l * n)$.
  This algorithm is not guaranteed to find an optimal solution because it's a local search algorithm that can only provide an approximately good solution.

- Run proteinFolding.py that is attached. The best conformation we got by this algorithm has a total cost of 271.3290352447519, with acids placed in order in the following coordinates: (0, 0), (0, -1), (0, -2), (1, -2), (2, -2), (2, -1), (3, -1), (3, -2), (3, -3), (2, -3), (2, -4), (3, -4), (4, -4), (4, -5), (3, -5), (2, -5), (1, -5), (0, -5), (0, -4), (1, -4), (1, -3), (0, -3), (-1, -3), (-1, -4), (-1, -5), (-1, -6), (-2, -6), (-2, -5), (-2, -4), (-2, -3), (-2, -2), (-2, -1), (-2, 0), (-2, 1)

References

[1] *An effective hybrid of hill climbing and genetic algorithm for 2D triangular protein structure prediction*