# 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 $h(s,t) \leq Cost(s,n) + h(n,t)$. Consider all sign combinations of $T(s,L) - T(L,t)$ and $T(n,L) - T(L,t)$ with the fact that $T(s,L) \leq Cost(s,n) + T(n,L)$ (triangular inequality), we see that $h(s,t) \leq Cost(s,n) + h(n,t)$ 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 $h_1(n) \geq h_2(n)$, then $h(n) = h_1(n)$ and there are two possibilities: first, $h_1(m) \geq h_2(m)$, 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 $h(n) = h_1(n) \leq Cost(n,m) + h_1(m) < Cost(n,m) + h_2(m)$, $h$ 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. Image 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 $h(m) \leq Cost(m,n) + h(n)$ will change. So $h$ will remain consistent.

## 3 Designing Search Algorithms: Protein Folding

- $S$ contains $s_{path}$ which is an ordered list recording all coordinates of residues that have been placed up to this point. $S_0 = s_{[(0,0)]}$
  $S_{Goal}$ is achieved when $len(path) = len(input)$
  $A$ contains $a_{s_0}$ and $a_{s_{path},direction}$, where $direction$ can be $(1,0), (-1,0), (0,i), (0,-i)$ and $path$ does not contain $path.lastElement + direction$
  $Successor(s_{path}, a_{path,direction}) = s_{append(path,\ (path.lastElement+direction))}$
  $Cost(s_{path}, a_{path,direction})$ = sum of distances from the new residual generated by $a_{path,direction}$ to all existing H-type residual in $path$.

- A*, with heuristic $h(i) = k * i + \frac{k!}{2*(k-2)!}$ where $i$ represents the $i$-th H-type residual has already been placed and $k$ represents the number of remaining H-type residual in the sequence. Since the distance between a pair is at least 1, this heuristic is consistent.
  Since the state space is too large, other types of searching algorithm will make a lot of ”unguided” exploration and waste a lot of resources. A* will provide an idea of the general direction that should be followed, and thus decrease the actual searching time.

- *The heuristic used is described in the above section.
  Function foldMinSearch
  input sequence (String)
  frontier = PriorityQueue of residual nodes that orders them by $Cost() + h()$
  add ($s_0$, 0 + heuristicHelper(number of H's in sequence, 0) to frontier
  visited = empty list
  while frontier is not empty:
  current, cost = frontier.pop
  add (current, cost) to visited
  if (len(current.path) == len(sequence)):
  return current.path
  foreach neighbour n of current:
  if (n not in frontier) AND (n not in visited):
  add $(n, cost + cost_n + h(n) - h(current))$ to frontier
  else if $cost_n > cost + Cost(current, n) + h(n) - h(current)$:
  add $(cost_n > cost + Cost(current, n) + h(n) - h(current))$ to frontier
  return not found

  Time Complexity: $O(numberofstatespace) = O(l * log(l) * 3^l)$, where we assume the branching factor is about 3 and there are $l$ acids in the input sequence; the presence of $l$ and $log(l)$ is to account for the innermost loop and the use of a priority queue, respectively.
  Space Complexity is similar to time complexity: $O(l * log(l) * 3^l)$.
  This algorithm is guaranteed to find an optimal solution because it's A* with a consistent heuristic.