Artificial Intelligence **Assignment 1**
comp440/comp557 Fall 2017 **4 September 2017**

This assignment is due September 2 at 8 pm on Canvas. Download `assignment1.zip` *from Canvas. There are four problems worth a total of 125 points and 15 points of extra credit for comp440/required for comp557. Problems 1, 2 and 4 require written work only, Problem 3 requires Python code and a writeup. All written work should be placed in a file called* `writeup.pdf` *with problem numbers clearly identified. All code should be included in* `submission.py` *at the labeled points. For Problem 3, please run the auto grader using the command line* `python grader.py` *and report the results in your writeup. Zip up the entire* `assignment1` *directory which should include your* `writeup.pdf` *and all of the Python code, and submit it as an attachment on Canvas by the due date/time.*

1 Modeling sentence segmentation as search (20 points)

In some languages such as Chinese, and in some English web domains, sentences are written without spaces between the words. An important first step in language processing is segmenting sequences of characters into words. Formally, we have a set of characters (in English, these would be letters) and an unsegmented sequence of characters that we call a sentence. We also have a dictionary D, which is the set of all words in a language, where a word is a sequence of characters. The goal of sentence segmentation is to split the sentence into words from the dictionary. For example in English, if D = {*i,cat,dog,see,sleep,the*}, then given the sentence *iseethecat*, *[i, see, the, cat]* is a possible segmentation.

- (5 points) Suppose our goal is to minimize the number of words in the output segmentation. Construct a deterministic state space model for this task.
- (5 points) Which search algorithms among BFS, DFS, UCS, A*, Bellman-Ford would produce a minimum cost path for your model and why?
- (5 points) If our goal is to maximize the number of words in the segmentation, revise the state space model from above. Which search algorithms work now?
- (5 points) Instead of minimizing the number of words in the segmentation, suppose we had at our disposal a function *fluency(w_1,w_2)* which returns a number (either positive or negative) representing the compatibility of w_1 and w_2 being next to each other. As an example *fluency(an,cat)* would be low and *fluency(a,cat)* would be high. Suppose our utility function is the sum of the fluencies of adjacent words; formally, if the segmentation produces words $w_1, \ldots, w_n$, then the utility is $\sum_{i=2}^{n} fluency(w_{i-1}, w_i)$. Modify the state space model from above to find the most fluent segmentation.

2 Searchable Maps (25 points)

In the US road network there are over 24 million nodes and over 58 million edges. Even on a state of the art computer with an optimized implementation of bi-directional Dijkstra, calculating the fastest route between a random start node s and a random end node t can take a few seconds. Google Maps (for example) performs these queries in a fraction of that time. In this problem we will unravel how we can use A* search towards handling a real sized road network.

- (5 points) The Haversine distance between points a and b written $G(a,b)$ is the distance between two points along the surface of the earth. Let S_H and S_L be the maximum speed and the minimum speed that a car is allowed to travel on any road respectively. Define an admissible heuristic for a search for the fastest path from s to t based on their Haversine distance.
- (5 points) It is infeasible to precompute and save the fastest path between every two nodes (there are over 500 trillion pairs in the US network alone), but it is possible to compute and store the fastest path between every node and a single landmark. Let $T(a,L)$ be the amount of time it takes to get from node a to landmark L. Use either the triangle inequality rule or the reverse triangle inequality rule in conjunction with landmark travel time to formulate a consistent heuristic $h_L(n)$ for a node n expanded by an A* search between s and t. For simplicity you may assume that all roads are traversable in both directions, at exactly the same speed. In other words $T(a,b) = T(b,a)$.
- (5 points) Let h_1 and h_2 be consistent heuristics. Define a new heuristic $h(s) = max(h_1(s), h_2(s))$. Prove that h is consistent.
- (5 points) Let's say you have K landmarks $(L_1, L_2, \dots, L_K)$. Use the intuition from the previous part, and your heuristics from the first two parts to create a single fastest path heuristic function.
- (5 points) Sometimes new roads are constructed, and sometimes roads are closed for repair. Suppose a heuristic h is consistent for a particular state space model. If new edges are added to the search space graph, does h remain consistent for the new model? What if edges are removed? In both cases, give a proof of consistency or a counterexample.

3 Package delivery as search (60 points)

Package delivery companies such as UPS and FedEx route millions of packages every day using thousands of vehicles. How do they decide which package to load, where and when? This is a complex problem with all sorts of constraints such as time, cost and capacity. In this assignment, imagine yourself as the owner of a newly started delivery business. To improve efficiency, you need to build a simple delivery planner.

3.1 Laying the groundwork (15 points)

Before you start delivering packages, you want to get familiar with the basic search algorithms so that you don't have to worry about those details later on.

1a. (5 points) We have already implemented uniform cost search (UCS) for you. Take a close look at the code (see `util.py`) to familiarize yourself with the algorithm, and try running it on the trivial test case (`util.trivialProblem` in `util.py`). Start `ipython` and type in the following after navigating to the assignment1 folder.

```
import util
ucs = util.UniformCostSearch(verbose=3)
ucs.solve(util.trivialProblem)
```

This should print out a trace of what the algorithm is doing.

Experiment with different graphs and look at the output to get an intuition for the algorithm. Note that UCS might explore many more states than the number of states along the optimal path. Let's try to quantify this: fill out `createUCSTestCase(n)` in `submission.py`. The function takes an integer n as input, and returns a search problem on which UCS explores at least n states whereas the optimal path only takes at most two actions. Run `python grader.py` to test your implementation.

1b. (10 points) Recall from class that you can implement A* by an elegant trick: take a problem P and reduce it to another problem Q, such that running A* on P is the same as running UCS on Q. Fill out `astarReduction(problem, heuristic)` in `submission.py`. Note that UCS, which is doing all the hard work, has no idea that it's actually running A*! You can test your A* implementation by running `python grader.py`.

3.2 Package delivery (45 points)

Now you can build a package delivery planner by applying the algorithms you developed. Let's treat those algorithms as black boxes and focus on modeling.

You will deliver packages in the following scenario: the city you live in can be viewed as a $m \times n$ grid, where each cell is either free (.) or occupied by a building (#). As you just started, you have only one truck (T). You have k orders, where the i-th order has a pickup location (Pi) and a drop off location (Di). Here's an example of a scenario:

```
D0 .  .  .  .  T
.  #  #  P1 .  .
.  .  #  #  .  .
P0 .  .  .  .  D1
```

There are 6 actions in total: you can drive the truck around (moving north, south, east, or west), pick up or drop off packages. Finally, you must return to your starting location(make sure you don't crash into any building during the whole process!). When you enter Pi, you can choose to pick up the i-th package if it hasn't been picked up. When you enter Di, you can choose to drop off the i-th package if you have picked it up. Moving from one cell to an adjacent cell costs 1 plus the number of packages that are being carried. You want to get all packages delivered and return to starting location, with minimal total cost.

2a. (5 points) Formalize the problem as a search problem, that is, what are the states, actions, costs, initial state, and goal test? Try to find a minimal representation of the states. In addition, how many states are there in your search problem? (express your answer as a function of m, n, and k, also include a brief explanation).

2b. (10 points) Implement the search problem you described by filling out `DeliveryProblem` in `submission.py`. You can run uniform cost search on it to solve the sample delivery problems in `util.py`. If your code is correct, you should be able to run the following:

```
import util, submission
ucs = util.UniformCostSearch(verbose=1)
scenario = util.deliveryScenario1
ucs.solve(submission.DeliveryProblem(scenario))
scenario.simulate(ucs.actions, True)  # Visualize the solution
print ucs.numStatesExplored, 'number of states explored.'
```

2c. (10 points) Now you have delivered some packages, it's time to make the search faster (at least in theory). Let's consider consistent heuristics which correspond to solving a relaxed problem. The first relaxation is assuming that you can drive through buildings and not deliver any packages, but you still have to go back to starting position. Implement `createHeuristic1(scenario)` in `submission.py` by using such relaxation. (Remember a heuristic function takes a state, and returns an estimate of the minimum cost from this state to the goal state) In addition, run A* with your heuristic in `util.deliveryScenario1`, and in your writeup write down how many states were explored. Run `python grader.py` to test your implementation.

2d. (10 points) You were perhaps a bit too relaxed, so in this relaxed problem, let's suppose you have to deliver some given package, and after that you have to go back to starting position - but you can still drive through buildings. Implement `createHeuristic2(scenario, package)` in `submission.py` by using such relaxation. In addition, run A* with your heuristic in `util.deliveryScenario2` and in your writeup write down how many states were explored. Run `python grader.py` to test your implementation.

2e. (10 points) In this final relaxation, each time you will deliver the most costly package (recall that the maximum over consistent heuristics is still a consistent heuristic), you can still drive through buildings. Implement `createHeuristic3(scenario)` in `submission.py` by using such relaxation. In addition, run A* with your heuristic in `util.deliveryScenario3`, and in your writeup write down how many states were explored. Run `python grader.py` to test your implementation.

4 Designing Search Algorithms: Protein Folding (20 points)

Proteins, which are initially synthesized in the cell as long chains of amino acid building blocks, automatically fold down into more compact working shapes. Exactly how they do this, and how they do it quickly and consistently, is poorly understood. The problem is so notoriously difficult that even dramatic simplifications are difficult to solve.

One stripped-down version of protein-folding is the so-called H/P (hydrophobic/polar) lattice model introduced by Lau and Dill in 1989, which we will use in this problem. In the H/P model, there are only two kinds of amino acids: hydrophobic amino acids (red in figures below) and hydrophilic or polar amino acids (blue in figures below). We will represent amino acid sequences in the binary H/P alphabet, using 1 to denote hydrophobic residues and 0 for the polar ones. For example, the amino acid chain in Figure 1 is represented by the sequence $[0, 1, 1, 1, 1, 0]$.

Since proteins exist in a watery environment, the hydrophobic amino acids prefer to cling to each other, compact and apart from the water as much as possible. We call a folded configuration of the protein a conformation. We will consider conformations in two dimensions, treating amino acids as links in a chain. Each link may bend by a multiple of 90 degrees, but the distance between links may not change (and is taken to be 1). No self-intersections are permitted, and the final conformation is constrained to lie on a single plane. A conformation of a protein is thus a self-avoiding walk on a two-dimensional square lattice or a Manhattan grid.

We represent the conformation of an amino acid chain of length n by a sequence of length $n-1$ as follows. We use complex numbers 1 for east, i for north, -1 for west, and -i for south. Each element k of the conformation represents the direction of the link joining the k^{th} and $k+1^{st}$ amino acids in the chain. The straight line conformation of the chain to the left in Figure 1 is represented by the sequence $[1, 1, 1, 1, 1]$, the middle conformation is $[1, 1, i, i, i]$ while the conformation to the right is $[1, 1, i, -1, -1]$.

We can calculate the locations of the individual proteins on the amino acid chain from its H/P sequence and its conformation. The chain $[0, 1, 1, 1, 1, 0]$ with conformation $[1, 1, 1, 1, 1]$ has its residues at location vector $[0, 1, 2, 3, 4, 5]$, while the same chain with conformation $[1, 1, i, i, i]$ has residues at $[0, 1, 2, 2+i, 2+2i, 2+3i]$.

Each conformation of the protein sequence is associated with a free energy. The free energy of a conformation is calculated by adding the distances between all pairs of hydrophobic amino acids. The free energy associated with the conformation on the left of Figure 1 is 10, with the conformation in the center is $5+\sqrt{2}+\sqrt{5}$ and that of the right is $4+2*\sqrt{2}$. Given an amino acid sequence of a protein in the H/P alphabet, the task is to determine an *optimal* conformation.

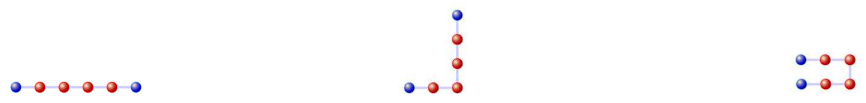

Figure 1: The amino acid chain to the left gets folded in two steps into an optimal conformation.

- (5 points) Formulate the problem of finding the minimal energy configuration of an amino acid sequence as a state space search problem. That is, identify the state space, the initial states and goal states, the successor function, and the path cost function. Design your successor function carefully to allow you to effectively find low energy configurations. You can consult papers that solve this problem to get ideas for state space representation and successor functions; please cite them fully (with URLs) in your writeup.

- (5 points) What family of search algorithms are suitable for solving the problem of finding minimal free energy configurations? Why?
- (10 points) Design an effective search algorithm for finding good solutions to this folding problem. Present your algorithm in pseudocode. What is the time and space complexity of your algorithm? Is your algorithm guaranteed to find an optimal solution? Explain your answers.
- (15 points: Extra credit for 440/Required for 557) Implement your algorithm in Python and print the best folded configuration you are able to achieve on the problem shown in Figure 2, where the amino acid starts out as a linear string of 34 amino acids. Does your implementation achieve the configurations shown in Figure 2? Why or why not? This problem will be graded on quality of foldings generated on the example shown in Figure 2. You are encouraged to look at the literature to find good representations of successor functions to achieve foldings of high quality.

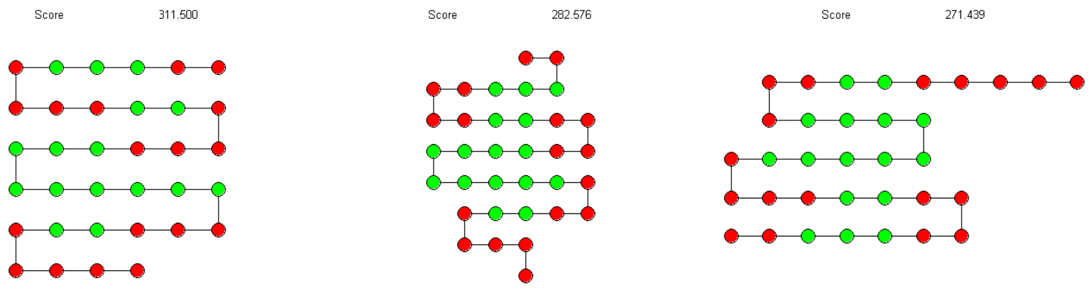

Figure 2: Three examples of folded amino acid chains.