CS 486/686 Fall 2022
Assignment 1

Blake VanBerlo

85 marks

Due Date: 11:59pm EST on Tuesday October 4, 2022

Changes

- v1.1: Updated the description of h_2 in Question 1 to enhance clarity.
- v1.2 Updated description of the successor function in Question 2 to remove ambiguity in the set of remaining locations. Clarified sorting in successor function.
- v1.3 Updated instructions for getting successors in Q2. Fixed state examples in Q1.
- v1.4 Specified that $\emptyset$ cannot be a successor of $\emptyset$ in Q1.

Instructions

- Submit the signed academic integrity statement any written solutions in a file named `academic_integrity_statement.pdf` to the A1 project on Crowdmark. **(5 marks)**.
- Submit your written answers in a file named `A1_written.pdf` to the A1 project on Crowdmark. I strongly encourage you to complete your write-up in LaTeX, using this source file. If you do, in your submission, please replace the author with your name and student number. Please also remove the due date, the Instructions section, and the Learning goals section. Thanks!
- Submit any code to `Marmoset` at `https://marmoset.student.cs.uwaterloo.ca/`. Be sure to submit your code to the project named `Final`.
- No late assignment will be accepted. This assignment is to be done individually.
- Lead TAs:
  - Connor Raymond Stewart (`crstewart@uwaterloo.ca`)
  - Dake Zhang (`dake.zhang@uwaterloo.ca`)

  The TAs' office hours will be scheduled and posted on LEARN and Piazza.

Learning goals

Uninformed and Heuristic Search

- Understand the components of a search problem.
- Trace the execution of Breadth-First Search and Depth-First Search.
- Define an admissible heuristic. Explain why a heuristic is admissible.
- Trace the execution of the A* search algorithm with an admissible heuristic.
- Implement the breadth-first search and A* search algorithms.

1 Breakfast Time (26 marks)

Every morning, you eat the same breakfast: fried eggs (sunny-side-up), toast with peanut butter, a banana, and a fresh cup of coffee. Since you like to sleep in, you try to make your breakfast as quickly as possible. You will apply a search algorithm to determine the sequence of tasks that will help you make your breakfast as quickly as possible.

Below is a list of tasks that are needed to prepare each part of your breakfast, along with some notes. Each minute, you can either wait or initiate 1 task, and multiple items can be prepared simultaneously. Any secondary task Y_2 can only be initialized after task Y_1 is complete. Each task can only be completed once.

Task	Time [min]	Symbol
Crack the eggs	1	$\mathtt{E}_1$
Fry the eggs	5	$\mathtt{E}_2$
Toast the bread	2	$\mathtt{T}_1$
Apply peanut butter	1	$\mathtt{T}_2$
Peel the banana	1	$\mathtt{B}_1$
Brew the coffee	3	$\mathtt{C}_1$
Pour the coffee	1	$\mathtt{C}_2$

Table 1: Tasks involved in preparing breakfast, along with the time they take to complete.

You define the search problem as follows:

- **States:** The set of breakfast items that have been started and/or completed, written in alphabetical order as a string. The time since the task was initialized is written as a superscript next to the task symbol. Symbols that are not present in the state represent tasks that have not been started yet. For example:
  - $E_2^1 T_1^2$ means that the eggs have been frying for 1 minute and the toast has been toasting for 2 minutes (so it is ready to eat).
  - $B_1^2 C_1^1 E_2^6$ means that the banana was peeled 2 minutes ago, the coffee has been brewing for 1 minute, and the eggs have been frying for 6 minutes. The banana and the eggs are ready to eat.
- **Initial state:** $\emptyset$
- **Goal state:** Any state $B_1^b C_2^c E_2^e T_2^t$ where $b, c, t \geq 1$, and $e \geq 5$
- **Successor function:** State B is a successor of state A if all of the superscripts for unmodified items in A are incremented by 1 and optionally one of the following:
  - An item Y_1^t in A is converted to item Y_2^1 in B (if $\mathtt{Y}_2$ is a valid symbol and t is at least the time required for $\mathtt{Y}_1$ in Table 1).

- Item Y_1^1 is added to B (if `Y₁` was not already in A)

Since you want to start breakfast as soon as possible, you cannot wait to begin breakfast. So as an exception to the above, $\emptyset$ is not a successor of $\emptyset$.

For example, suppose the current state is $E_1^1 T_1^2$. You could either wait, start brewing the coffee, peel the banana, or apply peanut butter to the toast. If the action is to wait, the next state will be $E_1^2 T_1^3$. If the action is to start brewing the coffee, the next state will be $C_1^1 E_1^2 T_1^3$. If the action is to apply peanut butter, the next state will be $E_1^2 T_2^1$.

- **Cost function:** The time elapsed (in minutes)

Please complete the following tasks.

(a) Recall that a heuristic $h(n)$ is consistent if, for every node n and every successor n' of n, we have $h(n) \leq c(n, n') + h(n')$. Prove that a heuristic function that is consistent is also admissible. You may assume that $h(n_G) = 0$ for any goal node n_G.

Hint: try a direct proof. Apply the definition of a consistent heuristic across an optimal path from some arbitrary node to a goal.

Marking Scheme: (6 marks)

- (4 marks) Correct proof
- (2 marks) The proof is clear, succinct, and easy to understand

(b) Define $h_1(n)$ as the number of tasks that have yet to be initiated. Recall that each row in the table is a task. In **4 sentences or less**, explain why $h_1(n)$ is consistent.

Marking Scheme: (4 marks)

- (4 marks) A reasonable explanation

(c) In **4 sentences or less**, explain why $h_1(n)$ is admissible.

Marking Scheme: (4 marks)

- (4 marks) A reasonable explanation

(d) Define the heuristic $h_2(n)$ as the maximum time to complete an unprepared breakfast item. For example, $h_2(E_1^2 C_2^1) = 5$ because the eggs require the maximum time to complete (5 minutes to fry the eggs). As another example, $h_2(B_1^2\, C_1^1\, E_2^5\, T_1^3) = 3$ because the coffee requires the maximum time to complete (2 minutes to finish brewing + 1 minute to pour).

Does h_2 dominate h_1? Explain why or why not in **4 sentences or less**

Marking Scheme: (4 marks)

- (2 marks) Correct answer
- (2 marks) A reasonable explanation

(e) Execute the A* search algorithm on the breakfast problem using $h_2(n)$ as described above. Do not perform any pruning. Add nodes to the frontier in alphabetical order. Remember to stop if you expand the goal state. **Draw the search tree until you have expanded 5 nodes.**

When drawing nodes, remember to write the state in the node. Annotate each node n in the following format: $c + h = f$ where c is the cost of the path to n, h is $h_2(n)$, and f is the sum of the cost and the heuristic values. Clearly indicate which nodes you expanded and in what order. You do not need to write out the frontier, but the tree must show all paths expanded after removing a node from the frontier.

Break any f-value ties using lexicographical order. For example, `B₁E₁` precedes `E₁` and should be expanded first if the f-values for both nodes are the same.

We recommend using https://app.diagrams.net/ to draw the search tree.

Marking Scheme: (8 marks)

- (8 marks) Correct search tree with all nodes correctly expanded in order.

2 The Travelling Trucker (54 marks)

In this programming question, you will solve a slightly modified instance of the Travelling Salesman Problem using the A* search and the breadth-first search algorithms, with multi-path pruning.

Suppose that you are a truck driver and your goal for the day is to deliver freight to all clients. You must leave the garage, make a stop at each client's warehouse to deliver freight, and return back to the garage at the end of the day. We relax the problem slightly so that you can drive through the same city more than once if needed. Since fuel is increasingly expensive, your company would like you to drive the minimum possible distance in total.

Given a list of locations with their respective latitudes and longitudes and the distances between connected cities, the task is to devise the shortest route that will take you from the garage, to each destination on the list, then back to the garage.

You are given two `.csv` files that contain all of the required information about the destinations and the map. Tables 2 and 3 illustrate the format of the `.csv` files for a map with 4 locations. Note that the `.csv` files do not contain column headers or row indices.

ID	Latitude	Longitude	Name
0	43.46	-80.52	Waterloo
1	43.45	-80.49	Kitchener
2	42.99	-81.25	London
3	42.40	-82.19	Chatham

Table 2: `locations.csv` gives the latitude, longitude, and name of the destinations. The row index gives the location ID.

	0	1	2	3
0	-1	3.6	-1	-1
1	3.6	-1	109	1
2	-1	109	-1	114
3	-1	-1	114	-1

Table 3: `distances.csv` gives the distance between locations on the map. The entry at row i and column j gives the distances between locations i and j. An entry of -1 indicates the locations are not directly connected by road.

The search problem is formally defined as follows:

- **States:** The state is the integer ID, latitude, and longitude of the current location, along with a set of the unvisited locations on the delivery list. For example, `(0, 43.46, -80.52, {1, 2})` indicates that the truck is in location 0 and has yet to deliver freight to locations 1 and 2. The state is implemented in the `State` class in the provided code.
- **Initial state:** The location of the garage and the set of locations to drop off freight
- **Goal state:** The location of the garage and the empty set of locations

- **Successor function:** State B is a successor of State A if and only if the following are true:
  - The location in state B is connected to state A by road
  - The location in state B is not equal to the location of state A
  - If State B's location is in the set of remaining locations in State A, then State B's set of remaining locations is the same as State A, but with State B's location removed. Otherwise, the set of remaining locations are equal for both states.
- **Cost function:** The road distance travelled between the locations in State A and State B

Information on the Provided Code

We have provided three Python files in a zipped folder on LEARN. Please read the detailed comments in the provided files carefully. Note that some functions have already been implemented for you. Your task is to implement all of the functions in `search.py`, except for `sample_heuristic()`.

1. `search.py`: Contains all the functions you will implement in this part of the assignment. You must implement the following functions: `is_goal()`, `get_path()`, `get_successors()`, `BFS()`, `A_star()`, and `custom_heuristic()`.
2. `utils.py`: Contains handy helper functions, along with implementations of a `State` class and a `Node` class for your search tree.
3. `example.py`: Demonstrates how to use the code to load the location and distance CSV files, and how to execute your search algorithms.

Zip and submit only the `search.py` file to Marmoset. We will use our version of `search.py` to test your code. Do not modify any provided function signatures in `search.py`. Doing so will cause you to fail our tests. Feel free to add any new code to `search.py`.

The Heuristic Functions for A* Search:

We have provided the implementation of a simple admissible heuristic function (see `sample_heuristic() in search.py`). This function is equal to the minimum road distance between any pair of locations multiplied by the number of remaining locations plus 1.

You must implement your own heuristic function in `custom_heuristic()`. As a hint, consider that the shortest travel distance between 2 points on Earth is the orthodromic distance. Consider using the `orthodromic_distance()` function available in `utils.py`.

Testing Your Program

Debugging and testing are essential skills for computer scientists. For this question, debugging your program may be especially challenging because of ties. Among "correct" implementations, the number of nodes expanded may vary widely due to how we handle the nodes with the same heuristic value on the frontier. Please test your code using **Python 3.8.5**.

Implement **multi-path pruning for both BFS and A***. When there are multiple paths to a state, multi-path pruning explores the first path to the state and discards any subsequent path to the same state. Use an explored set to keep track of the states that have been expanded by the algorithm. When you **remove** a state from the frontier, check whether the state is in the explored set or not. If the state is in the explored set, then do nothing. Otherwise, add the state to the explored set and continue with the algorithm. Note that we perform pruning after we **remove** a state from the frontier, not before we **add** a state to the frontier.

BFS's behaviour depends on the order of adding a state's successors to the frontier. We will break ties by using the number of remaining locations to visit in each state and their location IDs. Note that this will be done for you already if you produce a Python `set` of states with `get_successors()` and then sort it with `sorted()` (see the custom `==`, `<`, and `>` operators in the `State` class). At each step, BFS will add the successors to the frontier in **increasing** order of the number of remaining locations and then in increasing order of their IDs.

A* search will also break ties using the state comparison desribed above (see the overloaded operators for the `Node` class). Among several states with the same f-value, A* will expand the state with the fewest remaining locations. If two states have the same number of remaining locations, A* will break ties using the smallest location ID.

Please complete the following tasks:

Submit your solutions to part (a) on Marmoset and submit your solution to part (b) on Crowdmark.

(a) Complete the empty functions in `search.py` and submit `search.py` on Marmoset. Marmoset will evaluate your program for its correctness and efficiency.

For correctness, we have written unit tests for these functions: `is_goal()`, `get_path()`, `get_successors()`, `BFS()`, `A_star()`.

For each function, Marmoset provides one public test, which tests the function in a trivial scenario. There are also several secret tests. Before the deadline, you can only view the results of the public tests. After the deadline, Marmoset will run all the tests and calculate your marks.

Each test runs the function up to a predefined time limit. The test passes if and only if the function terminates within the time limit and returns the expected result. Each test is all or nothing — there are no partial marks available.

Marking Scheme: (46 marks)

Unit tests on `get_path`, `is_goal`, `blocking_heuristic`, `get_successors`, `dfs`, and `a_star`.

- `is_goal()`: (1 public test + 2 secret tests) * 1 mark = 3 marks.
- `get_path()`: (1 public test + 2 secret tests) * 1 mark = 3 marks.
- `get_successors()`: (1 public test + 3 secret tests) * 2 marks = 8 marks.
- `BFS()`: (1 public test + 3 secret tests) * 4 marks = 16 marks.
- `A_star()`: (1 public test + 3 secret tests) * 4 marks = 16 marks.

(b) Describe the heuristic function you implemented in `custom_heuristic()`. In your answer, justify why it is consistent. Lastly, comment on how A* with your heuristic compares to A* with the sample heuristic (e.g., cost of solution, number of nodes expanded, domination). **Be as concise as possible**.

Marking Scheme: (8 marks)

- (2 marks) Clear description of custom heuristic
- (4 marks) Correct and concise justification for consistency
- (2 marks) Illustrative comparison with sample heuristic