Assignment 1: Search

CSC 384H—Winter 2015

Out: Jan 20th
Due: Electronic Submission Monday 9th Feb, 7:00pm
Late assignments will not be accepted without medical excuse
Worth 15% of your final mark.

Be sure to include your name and student number as a comment in all submitted documents.

Handing in this Assignment

What to hand in on paper: No paper submission required.

What to hand in electronically: You must submit all your answers and code electronically. You must submit the following (more details about the contents of these files is given below).

1. A file of python code called **`eightPuzzle.py`**. This file will contain your implementation of the 8-puzzle search space.
2. A file of python code called **`bicycle.py`**. This file will contain your implementation of the bicycle courier delivery planning search space.
3. A PDF file called **`a1_answers.pdf`**. Use any word processing software to produce your written answers and then convert to PDF.

To submit these files electronically, use the CDF secure Web site:

`https://www.cdf.toronto.edu/students/`

or use the CDF **submit** command. Type **man submit** for more information.

Since we will test your code electronically, you must

- *make certain that your code runs on CDF using python3 (version 3.4.1* (installed as "python3" on CDF).
- not add any non-standard pythons imports from within the python files you submit (the imports that are already in the template files must remain).
- include all your written answers in the PDF file you create and submit (using the file name `a1_answers.pdf`)

Warning: marks will be deducted for errors in following the stated guidelines.

1 Introduction

In this assignment you will use some supplied search routines to solve the 8-puzzle and a simplified scheduling problem for bicycle courier deliveries.

What is supplied. You will be supplied with python code implementing various search routines. The file `search.py` contains a set of classes for doing search:

1. `StateSpace` is an abstract base class for implementing search spaces. This base class defines a fixed interface to a state space that is used by the `SearchEngine` class to perform search in that state space.

   For each specific problem that is to be solved one defines a concrete sub-class that inherits from `StateSpace`. This concrete sub-class inherits some of the "utility" routines already implemented in the base class, and overrides some other core routines to provide functionality specific to the problem being represented.

The file `WaterJugs.py` contains an example of a concrete `StateSpace` for solving the water-jugs problem. This `WaterJugs` class inherits some functions (methods)[1] from the abstract class `StateSpace` and overrides the following methods:

(a) The class initializer method "`__init__`". This method initializes the specific state data structures for a `WaterJugs` object, and calls the base class init method to initialize the general data shared by all state space objects.

(b) The `successors` method. Note that each `WaterJugs` object represents a state in the water-jugs state space. So `successors` is implemented as a "self" successor function. That is, the state uses this method to generate its own successors. The `successors` method returns a list of `WaterJugs` objects—each of which is reachable from "self" by a single action. Note that every `WaterJugs` object contains a reference to its parent, the action used to obtain it (in this case a string is used to represent the action), the cost of getting to this state (the g-value), along with the `WaterJugs` specific data structures.

(c) The `hashable_state` method. The search engine uses a python dictionary to implement full cycle checking. This means that it must use a state to index into that dictionary. Often is it necessary to use lists and other **mutable** python objects in the representation of a state, and these mutable data structures cannot be used to index into a dictionary. So to allow the search engine to operate correctly it is the responsibility of the implementor of a concrete state space to write a `hashable_state` function. This function returns a data item that (a) can be used to index into a dictionary (e.g., it might convert a list representation into an immutable tuple) and (b) two states generate the same `hashable_state` if and only if they represent the same state. For example, in water jugs we might have two different `WaterJugs` objects with different parents and different g-values, but they might represent the same state (i.e., the same configuration of the jugs). In this case, both objects should return the same data item when their `hashable_state` method is called.

Note that a concrete sub-class of `StateSpace` can use additional methods to make implementing the successor state method or other methods more modular. These extra methods will not affect the correctness of the search routines.

2. `SearchEngine` is a class that implements a number of search routines. By creating an object of this class initialized with various parameters we can set that object's search strategy and invoke that object's search algorithm on a problem. The `SearchEngine` class uses two other auxiliary classes:

   (a) `sNode` is a class used by the search routines to implement nodes in the search space. Remember that nodes in the search space represent paths in the State Space. In addition, the nodes also store other information useful for the search routines and implement comparison functions used to sort OPEN in the search routines.

   (b) `Open` is a class used by the search routines to implement the OPEN set. Dependent on the search strategy different types of data structures are used in the class to store the OPEN set. `sNode` and `Open` are internal classes used by `SearchEngine`.

Become familiar with this code. You will need to understand its general operator when debugging your implementation. Note that the `SearchEngine` method `trace_on` can be useful in debugging your implementation.

[1] Note that a function contained in a class is typically called a method.

2 Question 1 (Warmup): Implement the 8-puzzle: worth 20/100 Marks

Solve the 8-puzzle as a search problem by implementing it as a search space problem and using the routines in `search.py` to solve it.

1. Submit your python implementation in the file `eightPuzzle.py`. You will be supplied with a template file `eightPuzzle.py` that will specify the code you have to implement to interface with the search routines. As specified in the template file you will have to implement two heuristics: the misplaced tiles heuristic and the Manhattan distance heuristic.

2. The website also contains a link to the file `eightpuzzle_tests.py` which contains a number of test cases that you can run. The file `eightpuzzle_tests_output` is the output generated by a master solution.

3. In the file `a1_answers.pdf` answer the following "Q1" questions. **NOTE the space limits on your answer. If you exceed the space limits marks will be deducted.**

   (a) Which heuristic performs better, misplaced tiles or Manhattan distance. (One sentence).

   (b) Create a graph (up to 1/2 page) of the number of nodes explored by the Manhattan distance heuristic on the x-axis, and the number of nodes explored by the misplaced tiles heuristic on the y-axis. Put a point on this graph for each eightpuzzle problem solved from file `eightpuzzle_tests.py`. Can you say anything about the trend, i.e., as Manhattan distance has to explore more nodes how is the number of nodes explored by misplaced tiles growing? (1-2 sentences).

3 Question 2: Courier Domain (Worth 80/100 Marks)

In this question you are to solve a simplified scheduling problem for bicycle courier deliveries.

You have decided to work as a bike courier in order to make more money for tuition. Your innovation is a flexible pricing system that you hope will appeal to your customers. The key to making money with this pricing system is for you to figure out how to schedule each day's deliveries so as to maximize your income for that day.

Every day, you receive a list of delivery jobs. Each job is specified by a job name, a pickup location, a pickup time, a drop-off location, a package weight, and a sliding scale of payments such that you get paid more for faster deliveries. In particular, this sliding scale of payments is specified as a list of payment-time pairs giving the payment you will obtain if you can make the delivery by the specified time.

Your task is to deliver all jobs while maximizing the amount of money earned. So, e.g., you might choose to deliver package A at a later time because delivering package B earlier pays you more money.

You also have a map specifying all of the pickup and dropoff locations as well as the time (in minutes) required to move between locations (in this domain you can travel directly between any two locations).

A package cannot be picked up before the specified pickup time, and we assume that the earliest pickup time is 420 (we use a 24 hour clock and units of minutes, so 420 is 7:00am). We also assume that the deliveries must be completed by 1140 (7:00pm). So after 1140 no more travel is allowed, and the problem is unsolvable if you can't find a schedule whose final delivery is at or before 1140. Also, assume that on your bicycle you have carry at most 10000 grams of weight (10kg). (We won't worry about limits on the dimensions of the package). Hence, you can carry more than one package at a time, but never more than 10000 in total package weight.

3.1 Example

For example, you might have 4 locations, locA, locB, locC, and locD. The map will be specified as a list of two python lists, e.g.,

```
[
  ['locA', 'locB', 'locC', 'locD'],
  [['locA', 'locB', 15], ['locA', 'locC', 25], ['locA', 'locD', 10],
   ['locB', 'locC', 10], ['locB', 'locD', 13], ['locC', 'locD', 20]]
]
```

Notice this list has two elements, the first is a list of locations, and the second is a list specifying the time required to get between two locations (in minutes). This list says that it takes 20 minutes hours to get from locC to locD. Notice that the time is the same in the other direction (i.e., it takes 20 minutes to get from locD to locC), and that the list of times specifies the travel time between two locations in only one direction (i.e., the time from LocB to LocC is specified in the list while the time between LocC and LocB is not in the list).

With this map you might have the following list of deliveries to perform for the day:

```
[['Job1', 'locA', 480, 'locC', 5000
    [[510, 25], [570, 20], [600, 10], [660, 5]]],
 ['Job2', 'locB', 600, 'locC', 5000
    [[630, 25], [730, 5]]],
 ['Job3', 'locC', 540, 'locD', 10000
    [[545, 50], [570, 25], [600, 5]]]]
```

This list specifies that you have to do three delivery jobs, Job1, Job2 and Job3. Job1 is a pickup at locA that is available for pick up at 480 (8am). The package must go to locC, and it weights 5000 grams. You will earn the following payoffs if you deliver the package at time t:

1. If $480 \leq t \leq 510$ then you receive \$25.
2. If $510 < t \leq 570$ then you receive \$20.
3. If $570 < t \leq 600$ then you receive \$10.
4. If $600 < t \leq 660$ then you receive \$5.
5. If $660 < t$ then you receive \$0.

Note, (a) you may assume that the payoffs always decrease as the delivery times are later and (b) you earn nothing if the delivery is past the last payoff time **but you still have to make the delivery before or at 1140**.

A possible schedule might be to arrive at locA at 480 to do Job1 (note Job1's package isn't available before 480), deliver it to locC at 505 (the map specifies that it takes 25 minutes to get from locA to locC). Then wait there until 540 to pickup Job3's package, deliver that to locD at 560. Then proceed to locB arriving at 573, wait until 600 to pickup Job2's package, and then finally deliver that to locC at 610. This delivery schedule will earn you \$25+\$25+\$25 = \$75—after which you can head out for a good breakfast.

Note that this schedule only involved taking one package at a time. But you could have other schedules that involve transporting Job1 and Job2 at the same time (their weights sum to 10000 grams so you can transport both at the same time).

3.2 HOME location

If is also convenient to introduce a `home` location where you start off every day. `home` will not appear in the location map and has to be dealt with as a special case in your `successors` method. In particular, the initial state of search always has `home` as its current location (and it should be the only state that has that location since we don't need to have an action to go back home at the end of the day). The successors of the initial state (i.e., a state with home as its current location) are all states where you have arrived at a job location on time for the pickup at that time. Hence, the initial state has k successors when there are k delivery jobs.

In the example above there would be three successors of the initial `home` state as there are three delivery jobs:

1. A state where you are at locA at time 480 ready to pickup Job1.
2. A state where you are at locB at time 600 ready to pickup Job2.
3. A state where you are at locC at time 540 ready to pickup Job3.

3.3 To do

1. Implement a state representation for this problem and the successor state function. You will need to keep track of at least the following items in your state objects.
   (a) The jobs currently being carried by the courier.
   (b) The current location of the courier.
   (c) The current time.
   (d) The current amount of money earned.
   (e) A list of unstarted jobs.

   You will implement your state representation by completing the implementation of the `StateSpace` class `bicycle`, whose incomplete implementation is contained in the file `bicycle.py`.
2. Implement the successor state function for class `bicycle`. In particular, you are to implement the following actions.
   (a) Actions named `first_pickup(<job_name>)`.[2] This action moves the courier from "home" to the pickup location of `<job_name>`, sets the current time to pickup time of `<job_name>`, and starts `<job_name>`.
   (b) Actions named `pickup(<job_name>)`. This action involves traveling to the pickup location of an unstarted job to pickup `<job_name>`. A job is started once it has been picked. We cannot perform a `pickup(<job_name>)` action if adding `<job_name>` to the jobs already carried by the courier causes the total weight carried to exceed the courier's limit of 10,000 grams.
   (c) Actions named `deliver(<job_name>)`. This action involves traveling to the dropoff location of a job that is currently being carried and there delivering the job and collecting the correct amount of money (depending on when the job is delivered). After delivering the job the courier is no longer carrying it.

[2] Note that in this domain actions are represented as strings, and you have to generate the right string by replacing `<job_name>` with a job name from the input list of deliveries (using Python's `format` string method).

3. The courier is not allowed to perform a `pickup<job_name>` action if the pickup location of `<job_name>` is the same as the delivery location of some package currently being carried. (The delivery has to be made first, then the pickup can be executed).

4. Implement a `make_start_state` function

5. Implement a set of data access methods for your `bicycle` state objects.

6. Implement two heuristics for this domain so that you can use A* search.

For the last 3 items the details of the functions to be implemented are contained in the starter file `bicycle.py`.

Note also that your solution should accomodate impossible job lists—job lists that are impossible to complete before 1140, or lists that contain jobs that are too heavy to be carried. In such cases your solver should return that no feasible solution exists. (Hint, your successor state function should not generate successors whose time is greater than 1140).

3.4 Minimizing Costs vs. Maximizing Profit

We want to find a solution that returns the maximum revenue. However, A* returns a minimal "cost" solution. However, you can find a maximum revenue solution by defining the costs of the actions and the heuristic function in the right way. We can treat all of the `first_pickup` and `pickup` actions as having zero cost, while each `deliver(<job_name>)` action has a cost equal to the maximum possible revenue for `<job_name>` minus the actual revenue earned by the delivery acton. Then a minimum cost solution will minimize the "lost" revenue, thus maximizing the "gained" revenue.

In the example above potentially one could earn a maximum of $100, but the solution found yielded only $75, so you can consider this to be a solution that costs $25 (your lost revenue). In fact, this is the minimum lost cost solution (maximum revenue solution) as it isn't possible to deliver Job3 by 545.

In a similar way we can make our heuristics admissible, if they underestimate the future "lost" revenue.

Note that in general with zero cost actions A* need not be complete. But in this case it is possible to see that one can only perform a limited number of zero cost actions before a non-zero cost action must be executed (we can at only pickup up to 10000 grams of jobs before we have no more zero cost `pickup` actions to execute). Therefore, there are no infinite length action sequences of zero cost.

To Submit

1. Submit your python implementation in the file `bicycle.py`. A template file of the same name is provided to. The template specifies the format of input and output to your delivery schedule solver, so that we can run automated tests.

2. In the file `a1_answers.pdf` answer the following "Q2" questions:

   (a) For each of the two specified heuristics admissible say if it is admissible or not (one sentence). What would the answer be if the courier is allowed to only carry one package at a time (one sentence).

   (b) Does information from the location map have to be included in the hashable state? (1-2 sentences).

   (c) If part of the state can be computed from other parts (but is included for convenience) does it have to be part of the hashable state? (1 sentence).

(d) Does stopping the courier from performing a `pickup` actions at a dropoff location for a jobs they are currently carrying affect (a) whether or not a solution exists and (b) the cost of the optimal solution? (1 sentence). Does blocking these `pickup` actions potentially make the search more efficient? (1 sentence).

(e) If the courier is at some location, e.g., locA and there is an unstarted job that can be picked up at locA, is it always best for the courier to pickup up that job? (1-2 sentences).

(f) Including the current time in the state limits the effectiveness of cycle checking—being at the same location carrying the same jobs, having earned the same money, and having the same set of unstarted jobs will not be a cycle if the time is different. What problem specific rule can we use to prune states from the OPEN list that will still allow us to find the maximum revenue solution? (1-2 sentences)