# Assignment 1: Search

CSC 384H—Fall 2015

Out: Monday Sept. 28th
Due: Electronic Submission Monday Oct 12th, 7:00pm
Late assignments will not be accepted without documented medical excuse
Worth 10% of your final mark

## Handing in this Assignment

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

***What to hand in electronically:*** You must submit your assignment electronically. In particular, you must submit the following:

1. A file of python code called **`warehouse.py`**. This file will contain your implementation of the warehouse delivery planning search problem.

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.

We will test your code electronically. You will be supplied with a testing script that will run a **subset** of the tests. If your code fails all of the tests performed by the script (using python version 3.4.3) you will receive zero marks. If your code passes all of the tests performed by the script you will at least receive a passing grade on the assignment.

When your code is submitted we will run a more extensive set of tests which will include the tests run in the provided testing script and a number of other tests. You have to pass all of these more elaborate tests to obtain full marks on the assignment.

Your code will not be evaluated for partial correctness, it either works or it doesn't. It is your responsibility to hand in something that passes at least some of the tests in the provided testing script.

**Note the testing script will be made available a few days after this assignment is posted.**

- *make certain that your code runs on CDF using python3 (version 3.4.3) using only standard imports.* This version is installed as "python3" on CDF. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- Do not add any non-standard pythons imports from within the python file you submit (the imports that are already in the template files must remain). Once again non-standard imports will cause your code to fail the testing and you will receive zero marks.
- Do not change the supplied starter code. Your code will be tested using the orginal starter code and if it relies on changes you made to the starter code you will receive zero marks.

# 1 Introduction

In this assignment you will use some supplied search routines to solve a simplified version of the Amazon warehouse delivery problem.

**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 a specific problem 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 waterjugs 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 waterjugs 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 (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 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 state space to write a `hashable_state` function. This function returns a data item such that (a) this data 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 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 be called directly by 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`.

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

Become familiar with this code. You will need to understand its general operation when debugging your implementation. Note that the `SearchEngine` method `trace_on` can be useful in debugging your implementation. In addition the search engine object can be configured in various ways so that it performs various types of search.

# 2 Warehouse Domain

In this question you are to solve a simplified warehouse delivery problem. The problem is modeled after the much more complex system used by Amazon in their warehouses (see `https://www.youtube.com/watch?v=6KRjuuEVEZs`).

The warehouse has various products, a set of packing stations, and a set of robots. The products and packing stations are located at various fixed locations around the warehouse. The robots can move around the warehouse.

The typical problem for this domain is to operate the robots so as to complete a set of orders. Each order is for a particular product to be moved to a particular packing station. A robot must be assigned to this job and it will move to the product's location, pick up the product, and then move to the packing station to deliver the product. After the delivery the robot will remain at the packing station until it is assigned another job.

The problem is solved when all orders have been assigned to robots and the robots have completed all jobs they have been assigned. The quality of a solution is measured as the time at which the final delivery is completed. That is, a better solution delivers the orders to the packing stations more quickly.

The problem is simplified in a number of ways from the real application, some of these simplifications include: the robot can only carry one product at a time; each order consists of only one product to be delivered; we do not worry about the robots occupying the same location or otherwise crashing into each other; we don't worry about the actions involved in picking up the product and dropping it off at the packing station—we just ask the robot to "pick-up at location $(x_1, y_1)$ and deliver at location $(x_2, y_2)$" without worrying about the intermediate steps.

### 2.0.1 Locations and Robot Moves

Locations are specified using a simple $x$-$y$ coordinate system in which $x$ and $y$ are both integers greater than or equal to zero.

Robots can move between any two location. If the robot is currently at $(x_1, y_1)$ it can move to location $(x_2, y_2)$. It can only move horizontally or vertically and it takes 1 unit of time to move +1 or -1 in either direction. For example, to move from $(1, 2)$ to $(4, 5)$ the robot will need $\mathrm{abs}(1-4) + \mathrm{abs}(2-5) = 6$ units of time. In particular it needs to move +3 in the $x$ direction and then +3 in the $y$ direction.

### 2.0.2 Concurrency

When we have multiple robots we have to manage them executing delivery jobs concurrently. To achieve this the states of the state space will include (a) a current time and (b) some pending events that are going to occur in the future (i.e., in the future of this particular state—each state can have its own pending future events). This will allow us have a state space in which multiple concurrent activities can be taking place in any state. These activities all have some end time in the future.

In this domain the future events will all be completions of deliveries by different robots. So we can have a state where a bunch of different robots are all concurrently working on different deliveries. Each delivery will complete as some different time in the future.

To manage such states the state space will employ two different types of actions.

1. Actions that start up activities and update the state to include information about the future time when these activities will be completed. These actions generate a new state but that new state will have the same time as the current state. That is, these actions **do not move time forward** and cost zero (since we are trying to find a solution that completes at the earliest time).

2. A single action that moves the time forward. This action operates on a state by moving time forward to the completion of the earliest future event. In this domain, these are that events involving robots finishing deliveries. This action can only move time forward to the time of the earliest delivery among the state's concurrent active deliveries. These actions cost an amount equal to the amount the move time forward.

## 2.1 Example

Say we have the following products with $x$-$y$ locations (locations are represented as pairs $(x, y)$).

```
[['prod1', (0,5)], ['prod2', (1,5)], ['prod3', (2,5)]]
```

The following packing stations with $x$-$y$ locations

```
[['pack1', (0,0)], ['pack2', (1,0)]]]
```

And the following list of delivery orders that have to be completed

```
[['order2', 'prod3', 'pack1'], ['order3', 'prod2', 'pack2'],
 ['order4', 'prod3', 'pack1']]
```

That is we want to deliver two items of `prod3` to packing station `pack1`. In our simplified domain this can only be accomplished with two separate orders. We also want to deliver `prod2` to packing station `pack2`.

Finally, in our example we have two robots. To specify the robots and to capture the fact that a robot might be currently working on a delivery we need to provide information about the status of each robot.

```
[['r1', 'idle', (0,0)], ['r2', 'on_delivery', (1,0), 8]]
```

Here robot `r1` is currently `idle`. That is, it is not working on any delivery. It is located at coordinates $(0, 0)$. Robot `r2` on the other hand is currently `on_delivery`, we do not know its current location (as it is moving) but we know that after its delivery is over it will be located at coordinates $(1, 0)$. Furthermore, we know that it will complete its delivery at time 8.

Finally, the current time in the state is 0.

### 2.1.1 Actions

In this example state there are 4 possible actions that could be executed. We can pick any idle robot and get them to work on any order. In this case we have one idle robot and 3 possible orders, so these generate 3 actions. Finally we can move time forward to the completion of the earliest pending delivery, this is the fourth action.

1. Assign `r1` to `order2`. To deliver this order `r1` will have to move from its current location $(0, 0)$ to pick up product `prod3` which is located at $(2, 5)$ then it will have to move from $(2, 5)$ to packing station `pack1` located at $(0, 0)$. This means that the delivery will be finished in 14 time units from the current state time. The current time is 0 so the delivery will be finished at time 14, and will leave `r1` at location $(0, 0)$.

   The new state generated by this action will have an updated list of delivery orders where `order2` has been removed.

   ```
   [['order3', 'prod2', 'pack2'], ['order4', 'prod3', 'pack1']]
   ```

   The new state will also have an updated status for robot `r1`, so the robot status information will become:

   ```
   [['r1', 'on_delivery, (0,0), 14], ['r2', 'on_delivery', (1,0), 8]]
   ```

   Note that this action does not affect `r2`.

   The action does not change anything else in the state. In particular, the current time remains as 0.

2. We can assign `r1` to `order3`. This action updates the list of orders and the status of `r1`.

3. We can assign `r1` to `order4`.

4. We can move time forward to the time of completion of the earliest delivery. In this example, there is only one delivery in progress and it will complete at time 8. So this action does not change the list or orders, but it will change the robot status information to:

   ```
   [['r1', 'idle', (0,0)], ['r2', 'idle', (1,0)]]
   ```

   That is, `r2` now becomes idle and is located at the final location of the delivery it was on. Furthermore, the time of this new state has been moved forward to 8. Since we started at time 0, the cost of this action is 8 (the other three actions which are all delivery actions cost 0).

Some important notes:

1. It is important to be able to wait for a delivery to complete even when there is an idle robot and an unfulfilled order. In some cases the shortest completion time might be achieved by waiting for an active robot to finish and then using that robot, rather than using a far away idle robot.

2. In the new state after doing the first action of assigning `r1` to `order1`, we have the robot status

   ```
   [['r1', 'on_delivery', (0,0), 14], ['r2', 'on_delivery', (1,0), 8]]
   ```

   In this new state there are no idle robots, so there will be only one possible action that could be executed: the action of updating time. Furthermore, we can only update time to move forward to the smallest of 8 and 14 as 8 is the time of the earliest delivery. If we wanted to wait until time 14, we have to that by executing two move time forward actions: move time to 8, then move time to 14. Note the action that moves the time from 0 to 8 will cost 8, while the action that moves the time from 8 to 14 will cost 14-8=6.

3. A "move time forward" action is performed to move the time forward to the smallest of the completion times of the future deliveries. However, it could be the case that multiple deliveries are scheduled to complete at that time. In this case the robot status of all completed deliveries must be updated. For example, suppose that the state has the following robot status

```
[['r1', 'on_delivery', (0,0), 14], ['r2', 'on_delivery', (1,0), 8]]
 ['r3', 'on_delivery', (0,5), 8]]
```

A move time forward action will move the time to 8 (the smallest of the future delivery times), and will update the status of both robots `r2` and `r3`. The robot status in the new state after this action will be

```
[['r1', 'on_delivery', (0,0), 14], ['r2', 'idle', (1,0)]]
 ['r3', 'idle', (0,5)]]
```

## 2.2 To Do

1. Implement a state representation for this problem and a successor state function. You will need to keep track of at least the following items in your state objects:
   (a) The time in that state.
   (b) The list of unfulfilled orders in that state.
   (c) The status of each robot in that state. When the robot is idle its location needs to be known. When the robot is on an delivery the state needs to track the delivery's completion time and the location of the robot after that the delivery is finished.
2. Implement the successor state function for class `warehouse`. In particular you must implement two types of actions:
   (a) Actions named
   `deliver(<robot-name>, <product-name>,<packing_station-name).`
   In each state their will be one of these actions applicable for each combination of idle robot and unfulfilled order. This action removes the order and updates the status of the robot.
   The cost of these actions are zero.
   (b) An action named `move_forward(newTime)` which creates a successor state where the time is updated to `newTime` and all robots with deliveries scheduled to complete at `newTime` become idle. `newTime` must be the minimum of the completion times among the robots on delivery in the current state.
   The cost of these actions is the increment of time moved. That is, it is `newTime` minus the current time.
3. Implement a `make_init_state` function
4. Implement a set of data access methods for your `warehouse` state objects.
5. Implement a heuristic 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 `warehouse.py`.

**To Submit**

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