CM3038 Artificial Intelligence for Problem Solving

Solution to Laboratory #5: Path Finding in a 2D World

1. Aims

- To model the path finding problem in a 2D world as a search problem.
- To evaluate the performance of depth-first search on the path finding problem.
- To solving the path finding problem using A* search.
- To develop and evaluate heuristics for the path finding problem using A*.

2. Path Finding in the 2D World

In this problem, we find the path of an ant which moves in a 2D world. The ant can move in the directions of north, south, west and east if the target position is not occupied.

2.1. Resources

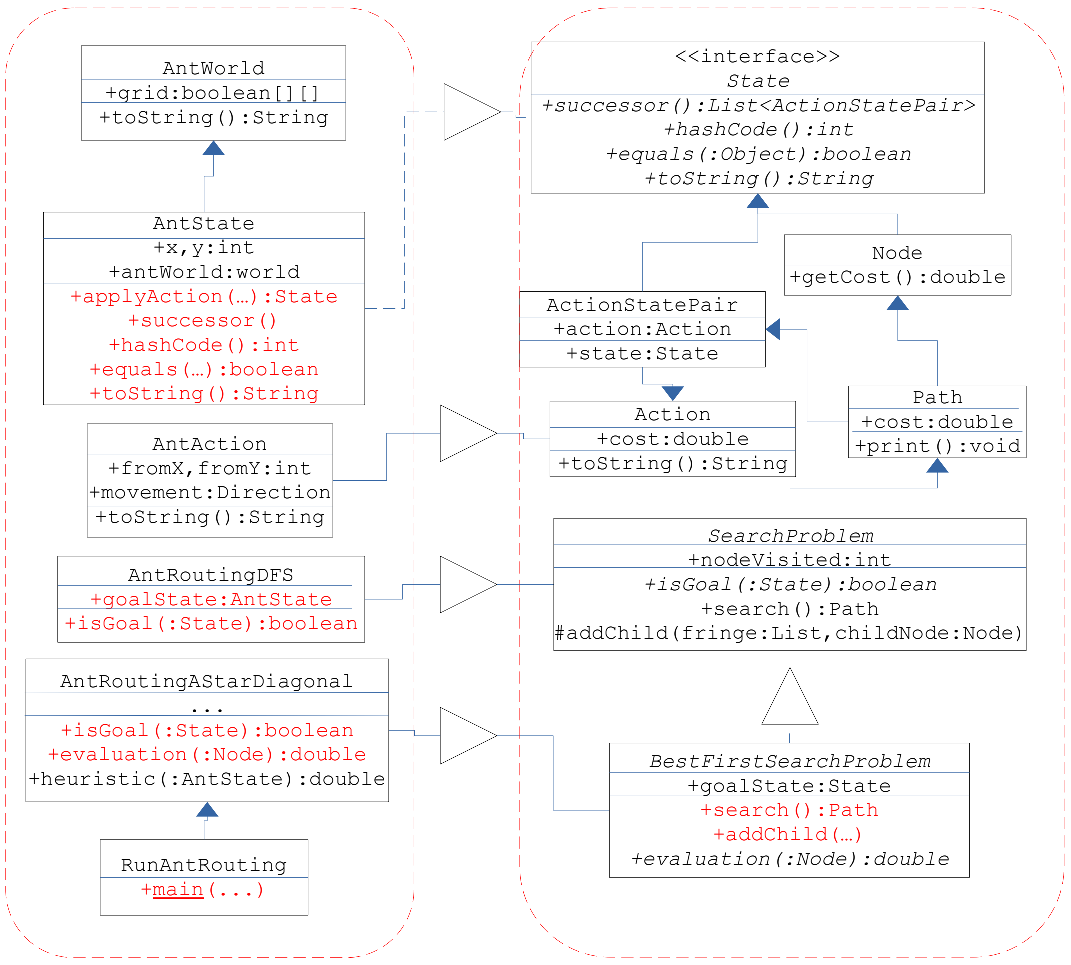
For Java, download the `cm3038-java-lab05.zip` file. This ZIP file contains the followings:

`aips.search`:	This package contains the uninformed search library.
`aips.search.informed`:	This package contains the informed search library.
`aips.lab05.ant`:	This package contains classes for the Ant World problem in Lab#5.

For Python, download the `cm3038-python-lab05.zip` file. This ZIP file contains the followings:

`aips/search.py`:	This package contains the uninformed search library.
`aips/informed/search.py`:	This package contains the informed search library.
`aips/lab05/ant.py`:	This file contains classes modelling the Ant World problem in lab#5.

2.2. Class Modeling



Domain dependent classes
for the ant routing problem

Domain independent classes
in the aips library

2.3. The `AntWorld` Class

The `AntWorld` class (or in Python, `AntWorld` in `ant.py`) models the 2D world as a two-dimensional `boolean` array. This class has been implemented for you.

- Examine the `AntWorld` class.
    - See how the 2D world is represented.
    - We use a simple 2D array to represent the world map. Not that we will only create 1 `AntWorld` object which will be referenced in all states, as the map won't change.
- Question:
    - If you need to represent a world with terrain apart from free space and barriers, what do you have to modify?
        - In Java, I use a 2D boolean array as there are 2 possible values in each space: barrier/free. If there are other terrains (e.g. water, mud), then we need to use a 2D array of int or an enumeration type (e.g. values of rock, mud, water, grass-land).

        - In Python, I actually use a 2D int array. You could use a boolean array if you prefer.

2.4. The `AntState` Class

The `AntState` class (or in Python, `AntState` in `ant.py`) models the state of an ant. In Java, it implements the `State` interface. In Python, it is a subclass of `State`.

- Examine the `AntState` class.
- Questions:
    - What information are kept about the ant?
        - The X,Y coordinates of the ant. We also keep an object reference to the world map which is not much about the ant itself. See the next question.
    - The state also keeps a reference to the ant world. Why is this needed?
        - When an ant thinks about a move, it needs to check if that move is valid. This information is only available from the world map (i.e. `AntWorld` object). Keeping the object reference of the world in the state is an easy way to make this information available.
        - On the other hand, when we need to print out the state of the problem, we need to show the location of the ant on the world map. So reference to the world map is needed.
        - There is no need to create a new `AntWorld` object for each state as the map does not change.
- The `toString()` method (or `__str__()` function in Python) returns the `AntState` as a `String`. It is currently incomplete.
    - ***Complete the implementation of the `toString()` method.***
    - Run `TestToString()` to test your method.
        - You don't really need to implement this method as we do not use it in our example. But in case you want to see how the world looks like, it is good to have this method as you can print out a state to the screen.
        - Putting in the "`X`" and "`.`" for occupied and free space is straightforward. The common mistake is not knowing where to put the "O" for the ant. It turns out that this information is not in the ant world object but the `x`,`y` attributes of the `AntState`. You can compare the current column and row of the array you are looking at with the `x`,`y` of the ant (in the `AntState`). If they match, then you are at the position of the ant, and you should append an "`O`" to the result `String`.

2.5. The `Direction` Enumeration Type

Before we can model the actions, we need to represent the 4 directions that the ant can move. In Java, this is done by an enumeration type called `Direction`.

- Examine the `Direction` enumeration type.

    - What directions are modelled?
        - We model the 4 directions of north, south, west and east. Usually people will put an extra "unknown" value there. You can choose not to have it.
        - I would say using enum type in Python is also a good idea. `str` is just the worst data type for this purpose.

In Python, I use simple str of "`N`", "`S`", etc. instead of an enumerated type. You may want to change this to a proper enum in Python but this is not required for this lab.

2.6. The `AntAction` Class

The `AntAction` class (in Python, the `AntAction` class in `ant.py`) models the 4 possible actions of an ant.

- Examine the `AntAction` class.
- Questions:
    - What information are kept about the action?
        - We keep the original coordinates (`fromX`, `fromY`) of the ant and the direction the ant is moving (`movement`). Strictly speaking we don't need the old coordinates in the action as they are kept in the original state (before the action is applied). But for easier implementation, I added the coordinates into action as I don't want to refer back to the original state to get the coordinates.
    - Do we need to keep information of the target position? Why?
        - If you know the original position and the direction of movement, you can always find out the target position. Alternatively you can store it too if you don't mind the duplicate information.
- The `toString()` method returns the `AntAction` as a `String` (or in `__str__()` function in Python). It is currently incomplete.
    - ***Complete the `toString()` method (or Python `__str__()` function).***
    - Run `TestAntAction` to test your method (or `testAntAction.py` in Python).
        - You may not need this method unless you are printing out the actions. In our current setup, we only show the path in the map and not printing out the actions. But this is a good method to have for debugging purpose.

2.7. The `applyAction(…)` Method

The `applyAction(…)` method in the `AntState` class applies an action to change the state. It returns a new `AntState` object. The method is currently incomplete.

- Examine the `applyAction(…)` method in the `AntState` class.
    - ***Complete the `applyAction(...)` method.***

- Run `TestApplyAction` to test your method (or `testApplyAction.py` in Python).
  - The implementation is quite straightforward: Based on the action's movement, you update the value of either x or y in the new state. Remember to create and return a new `AntState` object with the new coordinates. Do not modify the original `AntState` object directly. See the program code.

2.8. The `successor()` Method

The `successor()` method in the `AntState` class returns all valid actions of the current state as a list of `ActionStatePair` objects. The method is currently incomplete.

- Examine the `successor()` method in the `AntState` class.
  - ***Complete the `successor()` method.***
  - Run `TestSuccessor` to test your method (or `testSuccessor.py` in Python).
    - Hints:
      - The world, represented as an `AntWorld` object, can be found in the `world` attribute of the `AntState`.
      - Some actions are not possible if:
        - the ant is at a border
        - OR the target space is occupied
  - The `successor()` method will give a maximum of 4 successor states in the list (because the ant can only move in a maximum of 4 possible directions).
  - Basically we go through each of the 4 possible moves. Check if it is possible (look for occupied space, and check if the ant is on a border), then create the action, find the new state, and add them into the result set as an action-state pair.
  - The common problem is getting the wrong border check. i.e. moving the ant beyond a border. This is more a Java/Python programming skill problem rather than AI.

2.9. Doing Depth-First Search

The `AntRoutingDFS` class (in Python, the `AntRoutingDFS` class in `ant.py`) solves the ant routing problem as an uninformed depth-first search.

- Examine the `AntRoutingDFS` class.
  - Questions:
    - What is the superclass of `AntRoutingDFS`?
      - Its superclass is `SearchProblem`.
    - Is it an uninformed or informed search algorithm?
      - It is an uninformed search algorithm.
    - What method(s) is/are overridden in the `AntRoutingDFS` subclass? Why do we need to do this?

        - We override the `addChild(…)` method to change how a child is added into the fringe. This overrides the default BFS behaviour to DFS.
- Run the `RunAntRouting` class.
    - Questions:
        - How many nodes are visited before the goal is found?
            - This depends on the initial and destination positions. But depth-first search may not explore as many nodes as in Breath-first search.
        - How long is the path found? Do you think this is the optimal/shortest path?
            - In most cases it won't be an optimal path as DFS goes as deep as possible. It will be a very long path.
            - However, if you are lucky, DFS may find a path with the same cost as the optimal path.

2.10.Doing A* Search

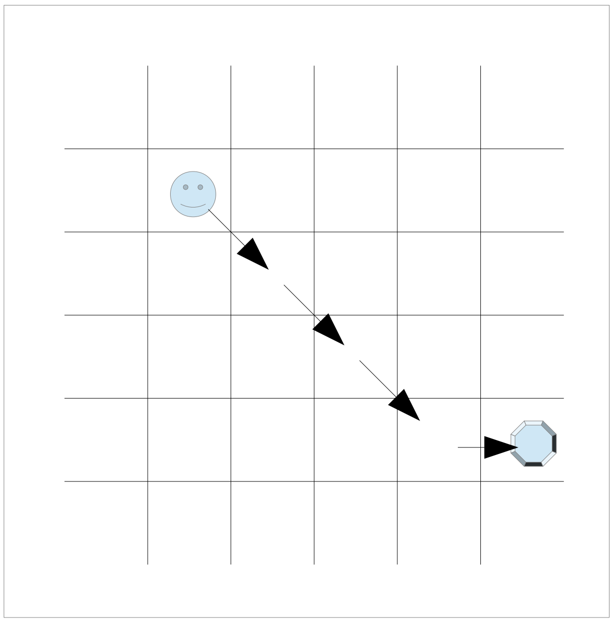
The A* search requires a heuristic to estimate the remaining distance of a state to a goal. The heuristics must be an under-estimate.

One way to invent heuristics for a problem is to relax the original rules. According to the ant world's rule:

- An ant can travel in 4 direction to a non-occupied space.

2.10.1.The Diagonal Move Heuristics

We can relax the restriction by allowing the ant to move diagonally and ignoring occupied squares. The `AntRoutingAStarDiagonal` class is implemented for you.

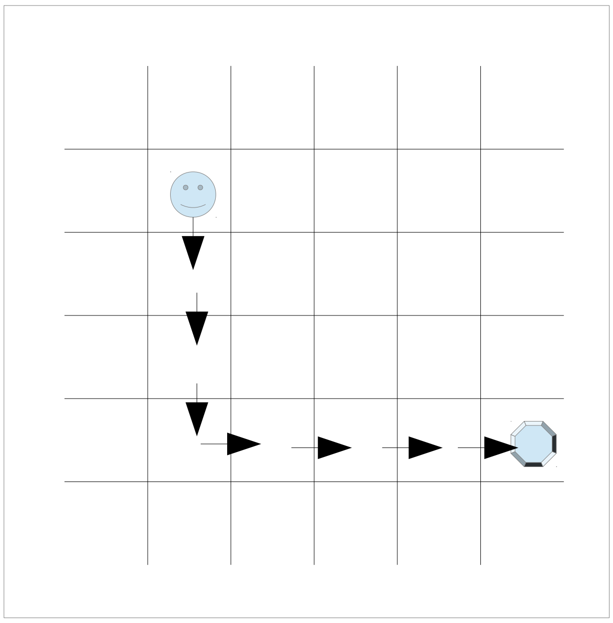


One diagonal movement reduces both horizonal and vertical difference simultaneously. The heuristic value is the maximum of the horizontal and vertical differences.

- Examine the `AntRoutingAStarDiagonal` class.
    - Questions:
        - Is the “Diagonal Move” heuristic admissible? Explain.
            - It is. Because in the original problem, it takes 2 moves to go to a diagonal position. Diagonal move shortens it to 1 move. So the actual number of moves required must be no less than the diagonal move number.
        - What is the superclass of `AntRoutingAStarDiagonal`?
            - It is `BestFirstSearchProblem`.
        - Is it an uninformed or informed search algorithm?
            - It is an informed search algorithm.
        - What is the evaluation function in the A* search?
            - f(n)=g(n)+h(n). i.e. cost from initial state to the current node plus the estimated heuristic value from the current state to a goal.
    - The heuristic function is implemented for you. See how its value is calculated.
- ***Modify the `RunAntRouting` class to do a A* search using the “Diagonal Move” heuristic.***
    - In the lecture, we discussed that if diagonal move is allowed, then the heuristic value is the maximum of the vertical and horizontal differences. Make sure that you take the absolute value of both differences as you don’t know where your ant is from the target destination.
- Compare the result with the uninformed depth-first search.
    - Does it explore fewer nodes?
        - It may not explore fewer nodes than depth-first search. Because an informed search like A* has to explore the best choice at each decision. It may end up exploring more nodes. If depth-first search is lucky enough, it may find a solution faster than informed searches.
    - Does it find a cheaper solution?
        - A* always finds an optimal solution. So unless DFS is very lucky, A* will always find a cheaper solution than DFS.

2.10.2.The Manhattan Distance Heuristics

The Manhattan distance heuristic relaxes the condition of "a non-occupied space". It estimates the remaining distance to the goal as the sum of the vertical and horizontal differences between a state and the goal, ignoring if a square is occupied.



The Manhattan distance is the sum of the horizontal and vertical differences.

Before we proceed: Remember that a heuristic function is domain-dependent. While the Manhattan Distance works in this "2D route finding" domain, it may not apply to other domains. **Some students tend to copy-and-paste and blindly apply the Manhattan Distance to all domains. This is wrong!** In some domains Manhattan Distance does not make sense. You need to study the suitability of a heuristic to a domain before using it. **You have been warned!**

- Question:
  - Is the Manhattan distance heuristic admissible? Explain.
    - It is. Because the Manhattan distance ignores all barrier on the path. So the actual number of moves required must be at least as expensive than the Manhattan distance if you need to get around the barrier. It can never be cheaper.
- Examine the `AntRoutingAStarManhattan` class. The `heuristic(…)` method is incomplete.
  - Questions:
    - What is the superclass of `AntRoutingAStarManhattan`?

            - In my solution I extend it from `AntRoutingAStarDiagonal`. This is a smart way of code reuse as I only need to override the heuristic method.
            - You can extend from `BestFirstSearchProblem` but you'll need to repeat other methods you have in `AntRoutingAStarDiagonal`.
        - What method(s) is/are we overriding in `AntRoutingAStarManhattan`?
            - We are overriding the `heuristic(…)` method alone.
    - Complete the `heuristic(…)` method.
- ***Modify the `RunAntRouting` class to perform an A* search using the Manhattan distance heuristic.***
- Compare the result with the Diagonal Move heuristics.
    - Does it explore fewer nodes?
        - It should, as Manhattan distance dominates Diagonal move heuristic, meaning that the MD distance is closer to the real number of moves required than DM.
    - Does it find a cheaper solution?
        - It does not find a cheaper solution than Diagonal move as they are both A* searches. The solution found is already an optimal one.
        - Note that while the solution cost found using both heuristic is the same, it does not mean the same solution is found. It is because there can be multiple solution paths which have the same cost.
    - Which heuristic is more informed? Manhattan or Diagonal?
        - Manhattan, of course. It is clear than the Diagonal move heuristic shortens the distance too much assuming that the ant can move in both x and y at the same time.

2.11.Doing Greedy Best-First Search

The Greedy Best-First search is similar to A*, except the evaluation function only consider the heuristics but ignoring the cost so far:

$$f(n)=h(n)$$

The `AntRoutingGreedyManhattan` class implements the Greedy Best-first search by extending the `AntRoutingAStarManhattan` class. It overrides the `evaluation(…)` method.

- Examine the `AntRoutingGreedy` class to see how the greedy best-first search is developed from the A* search.
    - What method is modified?
        - It simply changes the evaluation function from f(n)=g(n)+h(n) to f(n)=h(n), ignoring g(n) (i.e. the cost to the current node).
- ***Modify the `RunAntRouting` to do a Greedy Best-first search.***
- Compare the result with the A* search.

    - Questions:
        - Does it explore fewer nodes?
            - It may, in most cases.
            - This is easy to understand as Greedy doesn't guarantee optimality as in A*. To be optimal, A* needs to pay a price which is usually in having more nodes explored.
        - Does it find a cheaper solution?
            - It should not, because A* is finding the optimal solution. If it is lucky, Greedy best-first may find a solution of the same cost as A* but it cannot be cheaper.
        - Is it possible to fool the Greedy Best-first search to find a longer path? How?
            - Yes. You can create a U-shape "honey pot" so that the ant will go to the bottom as it looks closer to the destination. But in fact it has to go over the rim.
        - Try to change the world map to fool Greedy Best-first into find a longer solution path.
- Try using other heuristic functions with Greedy Best-first.
    - Compare the result of Greedy Best-first with the corresponding A* algorithm that uses the same heuristic function.