Cm3038 Artificial Intelligence for Problem Solving

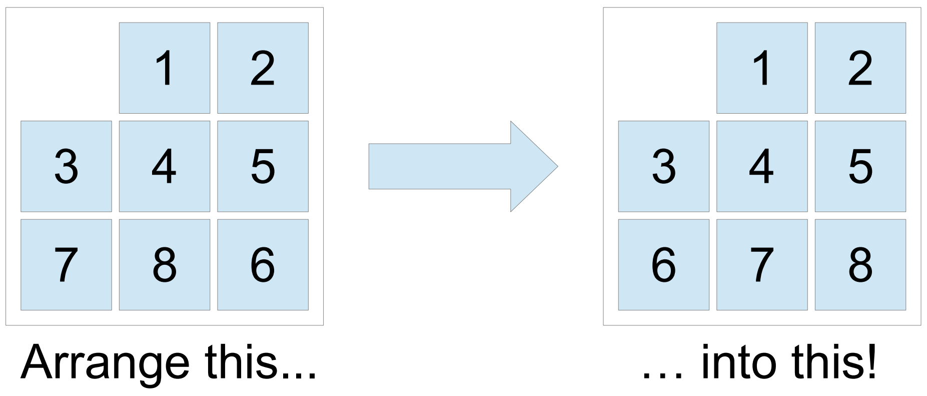
Solution/Discussion to Laboratory #4: Best-First & A* Search

1. Aims

- To solve the 8-Puzzle Problem using Best-First Search.
- To solve the 8-Puzzle Problem using the A* algorithm.
- Explore different heuristics in solving the 8-Puzzle problem.

2. The 8-Puzzle Problem

In the 8-Puzzle, the game board has 8 tiles which can slide into a blank space. The task is to move the tiles into a given pattern. A typical problem is shown below:



2.1. Resources

If you use Java, download the `cm3038-java-lab04.zip` file from Moodle. The 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.lab04.sq8`:	This package contains classes for the 8-Puzzle problem. Most classes are from Lab#3 solution.

Also check out the API document in `cm3038-java-lab04-api.zip`.

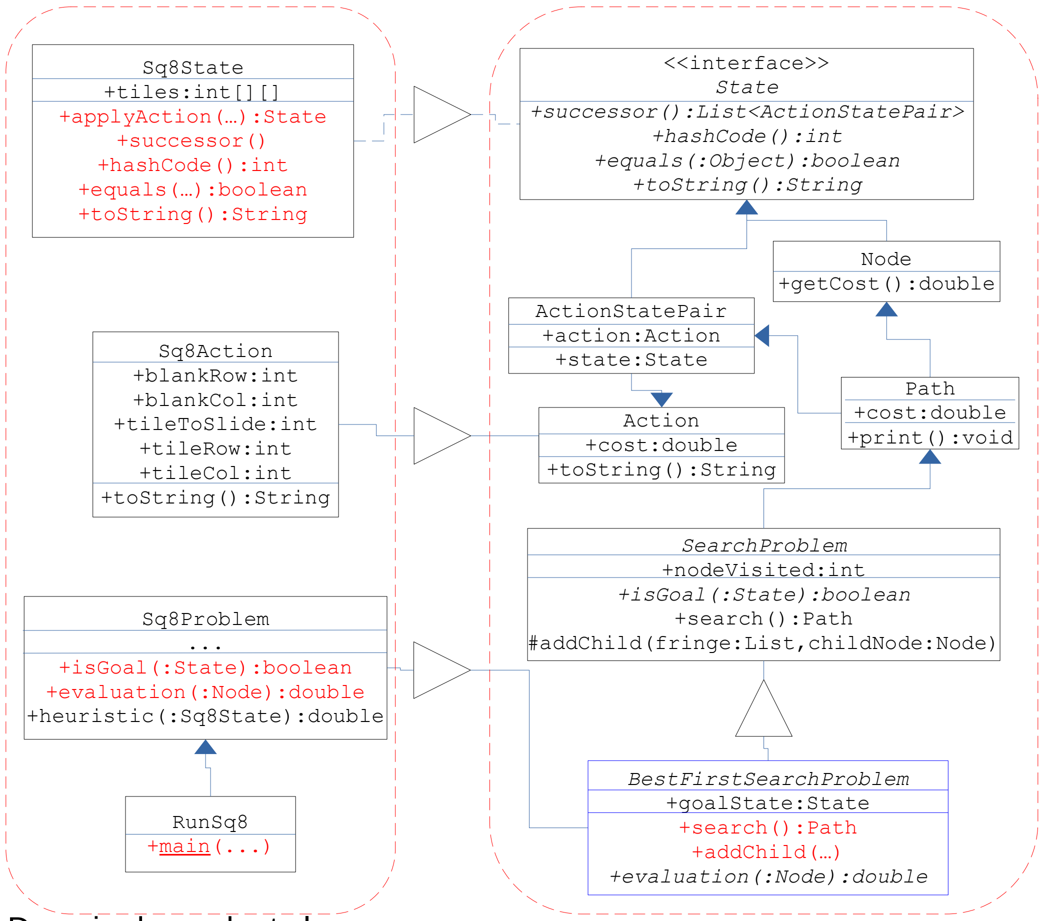
If you use Python, download the `cm3038-python-lab04.zip` file. The ZIP file contains the followings:

`aips/search.py`:	This package contains the uninformed search library.
`aips/informed/search.py`:	This package contains the informed search library.

`aips/lab04/sq8.py`: This package contains classes for the 8-Puzzle problem. Most classes are from Lab#3 solution.

2.2. Class Modeling

The following UML class diagram shows the new `BestFirstSearchProblem` class (in blue) defined in the `aips.search.informed` package. Methods in red are overriding inherited from superclasses.



Domain dependent classes for the 8 Puzzle

Domain independent classes in the aips library

2.3. The `BestFirstSearchProblem` Class for Informed Search

The `BestFirstSearchProblem` class models an informed search problem (for Python, this class is in `aips/informed/search.py`.). It modifies the uninformed `SearchProblem` class by overriding `search()` and `addChild(…)` methods.

- Questions:
  - Why are we overriding the `search()` and `addChild(…)` methods in the `BestFirstSearchProblem` subclass?

        - Hint: Does best-first search do these differently from uninformed searches?
        - If you look at Lecture 04, slide 8, the only difference between best-first search and previous uninformed searches is how nodes in the fringe is sorted.
            - The `addChild(…)` method controls where a child node is inserted into the fringe. So we need to modify this.
            - The `search(…)` method needs to take care of repeating states which can be reached with a lower cost. This also need to be modified.
    - Examine the `addChild(List,Node)` method (`addChild(…)` in Python).
        - How is a child node inserted to keep the fringe list sorted in ascending order of evaluation function value.
            - Every time a node is added to the fringe, we find its correct position based on its *f(n)* value. It is inserted right before the node which has a greater *f(n)* value. This is simple "insertion sort". In this case we can keep our fringe sorted in ascending order of *f(n)* without sorting the whole list.
            - In an older version, I scan through the fringe list from the beginning and stop when I find a node in the fringe having a higher f(n) value than the new node. This gives a time complexity of O(n) where n is the length of the fringe list.
            - In the current (better!) version, I use binary insertion by starting at the middle for the fringe. Depending on the f(n) of the middle node, we can tell if the new node should go to the first or second half of the list. This gives a time complexity of O(log(n)). The actual saving in time is actually huge!
- Examine the `search()` method. Notice the following:
    - Visited nodes are now stored as a map of `<State,Node>`. (In Python, it is a dictionary. But the idea is the same.)
        - We store references to the nodes because we need to modify the cost and parent attributes in a `Node` when needed (in case of a repeating state).
        - We use the state as a key into the map as we are checking if a state has been visited before.
            - There are many state-node pairs that we are storing in the map. This turns out to be a bottleneck in the execution as A* stores many nodes.
    - See how the method handles the situation when a state is re-visited with a cheaper path cost.
        - When the same state is encountered again with a lower cost, we have to replace the last path found (to this state) with the newer one. What we will do is described in Lecture 04, slides 16-25. Simply speaking, we update the old node so that the new path found is reflected. Cost and depth of a node are calculated by methods instead of stored as attributes. So we don't need to update them.

2.4. The `RunSq8` Class

The `aips.lab04.sq8.RunSq8` class (`sq8.py` in Python) runs the search.

- Examine the `main(...)` method (`run()` in Python).
    - See how a `Sq8Problem` object is created with the initial and goal state.
        - `Sq8State` has a constructor that takes a 2D `int` array. This is a convenient method to have when you want to test any state.
    - See how the search is invoked on the `Sq8Problem` object.
        - You should notice the following:
            - A `Sq8Problem` object is firstly created with the initial and goal state objects.
            - Then we invoke the `search()` method on the `Sq8Problem` object, which returns a Path object.
            - The return result is testing for `null` (when no solution is found), or we use the `print()` method on the `Path` object to print the path.
- Note:
    - The current implement runs Greedy Best-First, using a heuristic of "counting misplaced tiles".

2.5. The `Sq8Problem` Class

The `aips.lab04.sq8.Sq8Problem` class models the 8-Puzzle as an informed search problem (for Python, the `Sq8Problem` class is in `sq8.py`). You need to supply two important functions.

Class/Component	Description
`evaluation(Node)`	• This evaluation function returns a score for a `Node`. ◦ The score is in fact calculated based on the state stored in the node. ◦ The lower the score, the more promising the node. • The path cost from the initial state to node can be found in the `node.getCost()` method. • The state in the node can be found in `node.state`. ◦ It is a `State` object. But you can safely cast it into a `Sq8State` object. ◦ The `Sq8State` object has a 2D `int` array `tiles` array that stores the tiles on the game board. ◦ A value of `0` in the array means the empty space. • Note that depending on your strategy, you may not need to involve the heuristic function in the evaluation function. If you do not use any heuristic, you are doing uninformed search.
`heuristic(State)`	• This heuristic function is only needed if your

	evaluation function uses a heuristic. • It estimates the distance from the given state/node to a goal. • The goal state can be found in the `this.goalState` attribute. • The goal is a `State` object. You can safely cast it to `Sq8State` as that is the object you gave in creating the `Sq8Problem` problem instance. • The heuristic can be any non-over-estimating function. i.e. the return value must not be greater than the real cost to the goal. • You can use different heuristic functions. In general a heuristic function does not need to be admissible unless you are doing A* (where using an admissible h(n), and f(n)=g(n)+h(n) is a must, by definition).

Your job is to supply the evaluation function and heuristic function.

2.6. Some Heuristic Ideas

The heuristic function must be admissible. i.e. for al possible states in the domain, the value it returns must not be greater than the real cost to the goal state.

Some possible heuristics are:

1. Count the number of misplaced tiles. Because it must take you at least that number of steps to move all misplaced tiles into their correct positions. **This one is already implemented for you. However, it may not perform well.**
    - Question:
        - Is this heuristic admissible? Explain.
            - It is admissible.
            - Because you need at least 1 move to fix each misplaced tile.
            - The actual cost to fix all misplaced tiles can never be cheaper than the sum of fixing all of them.
2. Find the straight-line distance of each tile to its correct position. Then sum this up across all misplaced tiles.
    - Note:
        - Although we calculate the distance here, **our solution cost is the number of steps/moves.**
        - The (sum of) distance can be mapped directly into the number of steps with no danger of over-estimating.
    - Question:
        - Is this heuristic admissible? Explain.
            - It is admissible.
            - In this domain, the cost to fix 1 misplaced tile is never cheaper than the straight-line distance between the current and target position of this tile. In reality, the actual cost is usually higher because a tile cannot move in a straight-line and other tiles will be in its way.

            - The actual cost to fix all misplaced tiles cannot be cheaper than the sum of fixing all of them.

3. Find the sum of the Manhattan distance of all tiles to their goal positions.
    - For a misplaced tile, the Manhattan distance from its current to goal position is the sum of the vertical and horizontal differences.
    - The Manhattan distance of a tile's current and its goal position is the minimum number of steps to move this tile into its correct position. **This estimate is closer to the real cost than heuristic#2 above.**
    - Question:
        - Is this heuristic admissible? Explain.
            - It is admissible.
            - Because in this domain, a tile can only move horizontally or vertically. It cannot move in a diagonal.
            - So the Manhattan distance between a tile's current and target positions is the minimum number of moves (i.e. cost) to fix this tile if nothing is stopping it.
            - In reality the cost will be higher as there are other tiles in its way.
            - The cost to fix all misplaced tiles cannot be cheaper than the sum to fix all of them.
        - For a misplaced tile, compare its Manhattan distance with the straight-line distance to its target position. Which heuristic is "more informed"? (i.e. closer to the actual value)
            - The closer a heuristic function's value to the actual cost, the "more informed" it is. (i.e. the estimate is more accurate)
            - If you look at Manhattan distance versus straight-line distance, for any state, Manhattan distance gives a more accurate estimate as it follow the game rules closer. In particular, it respect the rule that a tile can only move vertically or horizontally, not diagonally.

2.7. Greedy Best First Search

Once you have a heuristic function, you can try to do Greedy best-first search.

Greedy best-first search uses the following evaluation function:

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

- Compare the performed of Greedy Best-first with different heuristic functions (in term of the cost of the solution found and number of nodes explored).
- Greedy best-first is not optimal. If you get an optimal solution then you are lucky. But it usually explores fewer nodes that A* because Greedy Best-first does not consider so many things. i.e. it is not that smart but goes simple.

2.8. A* Search

A* search needs an admissible heuristic. It uses the following evaluation function:

$$f(n)=g(n)+h(n)$$

g(n) is the path cost from the initial state to node *n*. You can get it by `node.getCost()` in thes `evaluation(Node)` method.

- Try different heuristic functions with A*. Compare the number of nodes explored and the cost of the solution found.
  - Do they all find the optimal solution?
    - As far as the heuristic is admissible, A* should give you an optimal solution. We have proved this in Lecture 04, slides 24-27.
    - Note that different heuristics may give you a different optimal solution because there may be multiple optimal solutions. But the solutions cost must be the same.
    - However, A* should not give a worse result than Greedy best-first because it is guaranteed to give you the optimal solution (lowest cost in this case).
  - Can you explain the difference in performance (number of nodes explored)?
    - In general, A* explores more nodes than Greedy best-first. This is the price to pay for optimality.
    - Sometimes A* may look like it is hung but indeed it is continuing the search. I have modified the BestFirstSearchProblem class to print out a message every 1000 nodes explored. You can see that it is remember many nodes. Thus the slower speed.
- Compare the result of A* with Greedy Best-First.

2.9. General Questions

- Is it possible for Greedy best-first to find a solution with the same cost as in A*?
  - It is possible. Greedy best-first does not guarantee optimality but it can find an optimal solution by chance.
- It is possible for Greedy best-first to find a solution with a lower cost than in A*?
  - This is not possible.
  - If a cheaper solution exists, A* should have found it. Otherwise A* is not optimal.
- It is possible for Greedy best-first to find a solution while exploring fewer nodes than A*?
  - It is possible.
- In general, why is A* exploring more nodes than Greedy best-first?
  - A*'s optimality comes with a price. Usually this is in the form of exploring more nodes.
- Is it possible for A* to find a solution with a lower cost then BFS?
  - Assuming all actions have the same cost, it is not possible for A* to find a cheaper solution than BFS as both are optimal.
  - However, if action cost is not uniform, the result can be different. This is because A* optimises the solution cost while BFS looks for

    the shallowest solution. These two criteria won't be the same if the action cost is not uniform.
  - See Lecture 02, slide 13 for an example.
- It is possible for A* to find a solution while exploring more nodes than BFS?
  - No.
  - BFS is the worst strategy to guarantee optimality. Unless you use a very poorly informed heuristic (e.g. h(n)=0), then you will get the same performance as BFS. Otherwise A* should have a lower node count.