Report A*

Anders Sildnes, Andrej Leitner *students*

***Abstract*—This text answers assignment 1 this is some more text that I can probably fill in somehow. Hello my name is alfred.**

A* is a local search algorithm. It is similar to DFS and BFS, except that when choosing a new node, it always chooses a node based on an index value.

I. Method

Since A* always chooses either the lowest or highest cost for a node, we desided to implement the queue with a heap. All heaps have the property that the root node has a consistenly higher or a lower value than all of its children. In addition, insertions are $\log n$, so maintaining such a queue is relatively fast. A regular list would cost at least $n \log n$ for sorting.

A* star is a method that should work equally on all *problems*. A problem needs to maintain a list of *states*. Since problem and states is the only variying input, we decided to model these using different classed and objects.

While python does not have the strict notion of abstract classes, we created a superclass "Problem" and an abstract superclass "State". These have general methods to be used in A*. The methods are listed in Figure 1. Explanation:

In the *Problem* class, the network

network is a reference to the (currently) cartesian plane of vertices and edges.

Destructor is invoked after the queue of states is exhausted (clean-up and animation). We included this since some animations will e.g. paint the goal node as a part of a path rather than a final state, etc.

genNeighbour() will genereate all possible successor states from the current state

triggerstart generates a starting state to be used in A*

updateStates paints and animates the next state. In the next state is a direct successor, we often dont need to paint every other state.

In Figure 3, you will find the following:

func,funcargs: when propagating and updating old states, I found it useful if all states had some knowledge of their own function and arguments. This is redundant (since often the function is the same in all states), but easier to read.

betterThanOther is the function equivalent of the <operator, used to compare states.

What can be seen from the abstract classes and framework is that none of the code mentions the exact type of problem to be solved. This means the code can be used to model any problem that has an iteratively progressing scheme.

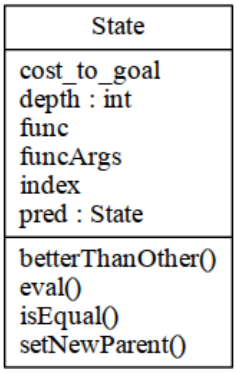

Fig. 1: UML Class diagram for a problem instance

II. A*

The assignment description models an A* search as follows:

1) pop the currently most promising state.
2) generate all possbile successor states
3) if a successor state has a similar index value or position:
   a) if the successor state is better:
      i) set the remembered state to this index such that future discoveries point to this successor
   b)

All states need a method to be indexed. This index should be similar between states at the same point. This means that the heuristic function returns the same value. The objective function, however, usually measured as the depth, can be different. In general we want the shortest path, so whenever two states have the same index, the one with the shortes ancestor tree is remembered as the state for the index, and the other is forgotten. In our case, the index value (assumed to be an integer) is stored in a dictionary. The lookup, and insertions are $O(1)$.

Since each successor will need to have an objective value, we use the objective function *attach-and-eval* whenever generating a successor. Similar to the A*-code given in the assignment text, we do not actually evaluate the state until it is popped off the queue.

III. Generating successor states

In a classical problem like search, we cannot generate completely random states. This would essentially be forking the problem in two: instead of only looking for the goal, we are now also looking for a path to the start. Other situations, does not necessarily require that you find an optimal path, e.g. the 8-queens puzzle. Here, you could make random choices and easilier find a valid solution.

Since the generation of successor states seems to depend on the problem, we opted with saying that the implementation of *genSuccessors* should be done for each problem. We could say that an A* search should just reason with its states and "state-children", but we felt this would clutter more than help.

To keep it general, the generation of successor states is kept

TABLE I: Comparison of nodes generated

DFS	1	100	110
BFS	9123	123	daw
BeFS	9123	123	* dwa

IV. Comparison of heuristic functions

In this assignment, we were supposed to find the shortest path from a point S to a point D on 2D-grid. This topic is widely discussed in other literature, so I will not elaborate.

A common heuristic function is the *manhattan distance*, which measures the distance from a point $P : (x_i, y_i)$ to $D : (x_j, y_j)$, by taking the difference between x_i, x_j summed with the difference between y_i, y_j. Here, x_j, y_j is the goal we are trying to reach, and x_i, y_i is the current point of the A* search. The downside of the manhattan distance is that it does not circumfere obstacles. Consider figure Figure 2

Other common heuristics, also used

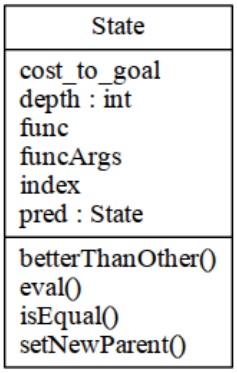

Fig. 2: Case where the manhattan algorithm does not fare well.

TABLE II: Lengths of solution paths

DFS	1	100	110
BFS	9123	123	daw
BeFS	9123	123	* dwa

Anders Sildnes, 4.års student, master informatikk, NTNU.

Andrej Leitner, 4.års student, master informatikk, NTNU.

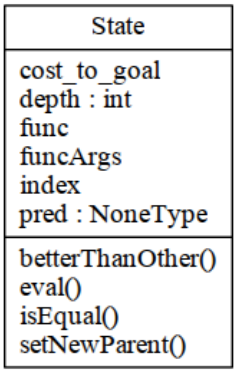

Fig. 3: UML Class diagram for a state instance