Report A*

Anders Sildnes, Andrej Leitner *students*

***Abstract*—This text answers assignment 1: Implementing and Testing the A* algorithm. The A*-method is intended to be written general enough to be used in later projects. This text elabortes on the choises made and the working of A*.**

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, rather than considering the time of which the node was spotted. The index value is supposed to estimate how far away from a given goal the current state is, and also how far away from start the state is. Thus, higher index values are not preferred. Each iteration in an A*-search is referred to as a state.

I. Method

Since A* always chooses either the lowest or highest cost for a node, we implemented the A*-agenda with a minimum-heap. Heaps have the property that the root node has a consistenly higher or a lower value than all of its children. Thus, the next state on the agenda is the one with the lowest objective value, and hopefully nearest the goal. Also, insertions are $\log n$, so heaps for agendas are quite fast (at least compared to regular lists that cost at least $n \log n$ for sorting).

A problem that is solvable in A* must have the following properties:

- Iterative, such that a problem during solving can be stored in a state
- An ability to find an objective value for a state
- An ability to generate (multiple) successors from a state
- Assessing whether or not a state is at the goal

Notably, all of these properties are related mostly to the problem itself, not the A*-method. Therefore, we wrote a single A*-method without the notion of class. Its reqirement is that its input is an agenda with states, and a method to assess these states. Since a lot of states are generated, we did not want to store all the information in the states. Therefore we also implemented a class "Problem" that has the ability to assess states and generate successors.

While python does not have the strict notion of abstract classes, we created "Problem" and "State" as abstract superclasses. Then, to create a problem, you need to create one subclasses for each, and implement the abstract methods. This can be done without consulting the code for A*, just consulting the docstring for the abstract methods. Hence we would like to claim that our A*-method is general.

Figure 1 shows the "Problem class":

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 canvas (animation-logic). We wanted to include this method in this class because we felt the animation-logic varies from problem to problem.

In Figure 3, there is the "State" class:

func,funcargs: is the objective function and the arguments to be used in the function. Its currently the same in every state, but now the program can be modified such that in the future, a state and its predecessors can change function when propagating and updating old states.

betterThanOther is the function equivalent of the < operator, used to compare states. It is used to compare two states with similar values. If one state has a shorter or longer ancestor tree than the other, "betterthanother" returns true or false, respectively.

II. A*

Our A*-implementation is intended to be the same as in the assignment text. Below is our attempt at an explanation. By *index* we mean that two states are similar except that their ancestor tree is different.

1) pop the currently most promising state.
2) generate all possbile successor states
3) assign an objective value to each successor
4) if a successor state has a similar index as a previously generate state:
   a) if the successor state is better:
      i) set the remembered state to this index such that future discoveries point to this successor
      ii) propagate back along the ancestor tree of the old state, and ensure that if using the new states' path to get to an ancestor is better, this will be preffered in the future.

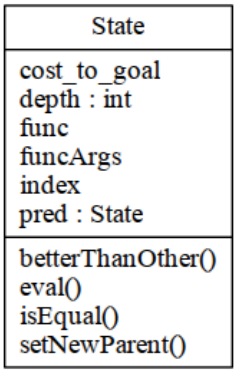

Fig. 1: UML Class diagram for a problem instance

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

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

TABLE I: Comparison of nodes generated

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

TABLE II: Lengths of solution paths

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

Other common heuristics, also used

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

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

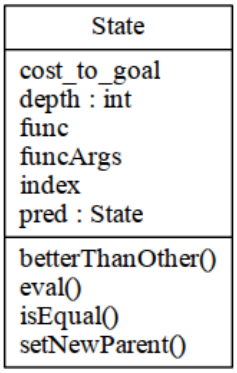

Fig. 2: Case where the manhattan algorithm does not fare well.

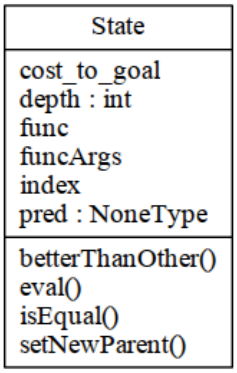

Fig. 3: UML Class diagram for a state instance