Assignment 1: Navigation using A* algorithm

Olav Kåre Vatne

Implementation

By using knowledge about the problem a search algorithm can move through the search space more efficiently. Because the heuristic function can make estimates of the distance between the current state and the goal state, it can be used to guide a search algorithm.

The A* star algorithm takes advantage of the heuristic function by expanding the most promising node first, the one with the smallest $F(x) = G(x) + H(x)$. If this function was not available the best first search would become a breadth first search, where all nodes are methodically expanded out from the root node. A best first search can also be called a hybrid search method between breadth and depth first search, and will employ the two strategies depending on the heuristics function.

My implementation is based on the pseudocode given in the document titled, “Essentials of the A* Algorithm”. By exploiting object inheritance, a general A* implementation can be made which can be used on a lot of different problems. Provided that a problem specific subclass is made that implement some abstract methods.

Search states are encapsulated in a node object. The node object has properties that will enable tree traversal. Each general node has a best parent and children variable. When a node is expanded, and successor search states generated, these are added as children nodes , of the expanded node.

Each node has to support a few methods, to be of any use to the A* implementation.
My node objects therefore has abstract methods for retrieving the heuristic value, the cost of getting to the node. The search state also has to have a way to identify itself, to compare different search states that are generated, and to see if these are identical. It must provide the arc-cost of getting from itself to one of its children nodes, and a way to generate the successor search states encapsulated in node objects.

The A* implementation has no clue about the actual problem, no domain specific knowledge. It relies on the methods mention above, to perform a intelligent search through the search space.

It is the navigation module that is responisble for providing the implementation of these methods. Both a navigationNode and a NavigationState (just a static map) are used to provide the necessary knowledge for the A* implementation.
The text input containing tuples for dimension, goal, start and obstacles are used by navigationState to construct a internal representation of the map. It’s from here all successors are generated. The method generateSuccessors takes the current state, the parent state as

arguments and use their cell position on the board to generate the children for the current state, taking into account board edges and board obstacles.

Since the implementation should support both depth, breadth and best first search, some customization of the general search algorithm had to be made. The search function takes an additional argument “mode” that tells what search technique to use.

The open/frontier list is a list of nodes that are available for expansion. The ways the nodes are added and popped is very important. For best first search the node with the lowest F value has to be popped. This is achieved by using a heap sorted on the F values. For depth first the open/frontier list has to act like a stack, appending and popping from the top of the list. A queue is used for breadth first search. Nodes are popped in first in first out order.

Since the frontier list has to support these different modes, it is encapsulated inside an object. The frontier object has a init method that initialize the list to either a queue, a heap or a stack/list depending on the mode argument. The object also has a pop and a add method that will retrieve and add node depending on the mode. This is how the agenda is managed.

Reusability

The actual search does not have details specific for the navigation task. By exploiting inheritance and abstract objects the search can be oblivious to the problem domain. In the implementation the search only relies on node objects. The node object has necessary methods like generateSuccessor, calcH, getG, isSolution and so on. Since the node has to be general all these methods are abstract, and its the task of the subclass to provide the implementation. These subclasses (like NavigationNode) can have details specifics about the problem, like the board dimensions, goal coordinates and obstacle coodinates. These subclasses can be very different to each other, but will work with the search implementation as long as the abstract methods are implemented.

Heuristic function

The heuristic function is often very dependent on the problem domain, so specific details are necessary. For navigation on a 2D board, where only horizontal or vertical movements are allowed are different to a navigation using a graph with weighted arcs between the nodes. Therefore the heuristic function has to be implemented in the subclass where all the domain details are.

For navigation on a 2D board, manhattan distance was used. This distance gives the minimum number of vertical and horizontal moves from a cell to the goal cell. Even though there probably is an obstacle that will prevent the minimum number of moves to be equal to the manhattan distance its still valuable, and can be used to assess the potential for the nodes in the agenda. Also when a heuristic is inherently optimistic, the A* algorithm is guaranteed to recognize the optimal solution. The manhattan distance will therefore never overestimate the distance to the goal state for this particular problem.

Generating successor states

Generating successor states is also an operation that is very dependent on the problem specifics, and is hard to generalize. Therefore its also the subclass' task to implement it using problem domain specific knowledge.

For the navigation problem, each navigationNode has access to a navigationState that contains the dimension, goal, start and obstacles for the 2D board. This object also has a successor generator that utilize these coordinates to generate successors when a node and its parent is provided.

Any state with a x,y coordinate on the 2D board. can have the maximum of four children states, because its only possible to move left, right, up and down on the board. However, there might be illegal moves since obstacles and the board edge is blocking the path. The generateSuccesors method therefore has four if sentences, checking if the successor is legal. Is the new position outside the board? Is there a obstacle in the way? In addition successors which has the same position as the parent of the current state will not be included. Most nodes will therefore have 3 or less successors.