Using A* to solve Navigation Problem

Johannes Omberg Lier   Per-Christian Berg

October 4, 2015

1 Implemention of A*-algorithm

1.1 Agenda loop

The agenda in our code is represented by our own implementation of a heap queue, and uses the python module *heapq*. By using a heap queue the task of sorting each search node by the f-value is done quickly and without much effort. We chose the heap queue as our agenda because the heap queue will always have the element with the lowest value as its first element. We implemented our agenda in such a way that if the search mode is either breadth-first-search (bfs) the *pop*-function will return the first element in the queue and the *add*-function will add elements in a normal fashion (first in first out). If the search mode is depth-first-search (dfs), the *pop*-function will return the last element and *add*-function will word in the same way as with bfs (last in first out).

```
def pop(self):
        if self.astar: return heappop(self.storage)
        elif self.bfs: return self.storage.pop(0)
        elif self.dfs: return self.storage.pop()

def add(self,element):
        if self.astar: heappush(self.storage,element)
        elif self.bfs: self.storage.append(element)
        elif self.dfs: self.storage.append(element)
```

Listing 1: Adding and popping from the agenda

Our A*-loop takes the first search node generated and generates its children-nodes. For each child that is not yet created (child.status is None), the child's f-value is calculated. All the newly generated children is placed in the open queue.

1.2 Heuristic function

To be able to determine which cell is the better choice to expand during the navigation search you need to able to rank the different cells. In short, this ranking is defined as the amount of steps taken to get to a cell pluss an estimated amount of steps needed to get to the goal. This sum is the previously mentioned f-value. The estimation is the h-value and is crucial for a properly functioning A*-algorithm. The A* guarantees to find the shortest possible path, if one exists, if the heuristic is *admissible*, which means that the estimated distance from one cell to the goal is never overestimated. A simple way of making sure that the heuristic is never overestimated is to set the h-value to the sum of the distances from the cell to goal in x-axis and in the y-axis. This is the Manhattan distance and is recommended to use when the path can turn in four different directions. In the situation where the path can turn in eight different directions, the euclidean distance is the best choice.

```
def arc_cost(self,currentNode,neighbour):
                a = currentNode.state[0]-neighbour.state[0]
                b = currentNode.state[1]-neighbour.state[1]
                return sqrt((a**2) + (b**2)) * self.cost

def attachAndEval(self,node,currentNode):
        node.parent = currentNode
        node.g = currentNode.g + self.arc_cost(currentNode,node)
        node.f = node.g + node.h
```

Listing 2: Estimation and updating the f-value in a node

2 Generality

2.1 Abstraction

Our A* implementation is in our opinion general and can easily be used in other problems that the navigation problem solved in this module. The search node is not defined in the A*-algorithm and can therefore take any implemented node from the agenda as long as it has the required values like the values needed for the heuristics, parent nodes, child nodes and status field. When reviewing the code for this problem you will find code that is specific for this problem but it is limited to the gui implementation.

2.2 Reuse

To be able to use the A* implementation for another problem you need, as briefly mentioned above, search nodes that has the required values. Also, a method of generating the neighbors needs to be implemented outside the A*-algorithm.

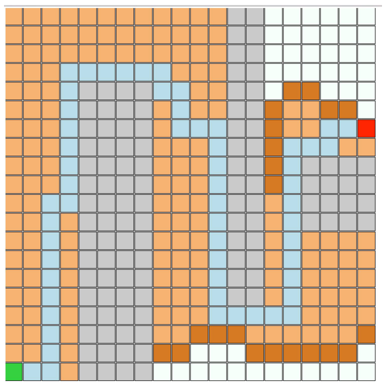

Figure 1: The result of running the A*-algorithm with the fifth navigation scenario. **Lightbrown: Closed nodes, Dark brown: Open nodes, Gray: Obstacle, Blue: Path**

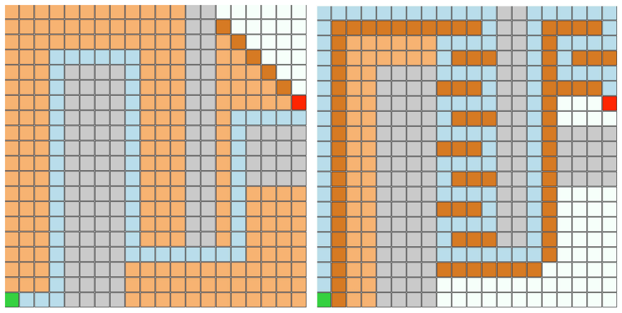

Figure 2: The same scenario as in figure 1, but with bfs (left) and dfs (right)

3 Result

Mode	**Length of path**	**No. Generated nodes**	**No. Open**
A*	59	243	21
BFS	59	267	5
DFS	107	232	81

Figure 1 shows the result after running our implementation on the fifth navigation scenario described in the assignment. When running with A*, the algorithm generates 243 nodes where 21 of them where left open. The length of the path was 59 (including start and goal node). When running bfs 267 nodes where generated where 5 was left opened. Dfs generated 232 nodes where generated where 81 where never closed. As expected, the paths from bfs and A* had the same length. For dfs the total number of nodes in the path was 107.