Using A* to solve Navigation Problems

Kristian Våge and Bjørn Bråthen

October 4, 2015

1 Central aspects of the A* program

1.1 The agenda and how it is managed

Three search routines had to be supported out of the same base algorithm with a flick of a notch. This is quite easy when you know that substituting the heap with either a queue or a stack, you get respectively a breadth-first- and a depth-first-algorithm that is suitable to solve the same problems, though with different performance measures.

Substituting the heap is done easily in python, and you can then define some common pop and push methods that modify the agenda, whether its a heap, stack or a queue (see figure 1). In the code snippet, 'opened' is the datastructure of your choice.

```
def open_push(self, opened, node):
  """ Push node onto opened according to mode, and set status """
  node.status = C.status.OPEN

  if self.mode is C.search_mode.A_STAR:
    q.heappush(opened, (node.f, node))
  elif self.mode is C.search_mode.DFS or self.mode is C.search_mode.BFS:
    opened.append(node)

def open_pop(self, opened):
  """Pop node from opened according to mode """
  if self.mode is C.search_mode.A_STAR:
    return q.heappop(opened)[1]
  elif self.mode is C.search_mode.DFS:
    return opened.pop()
  elif self.mode is C.search_mode.BFS:
    return opened.pop(0)
```

Figure 1: Management of the agenda.

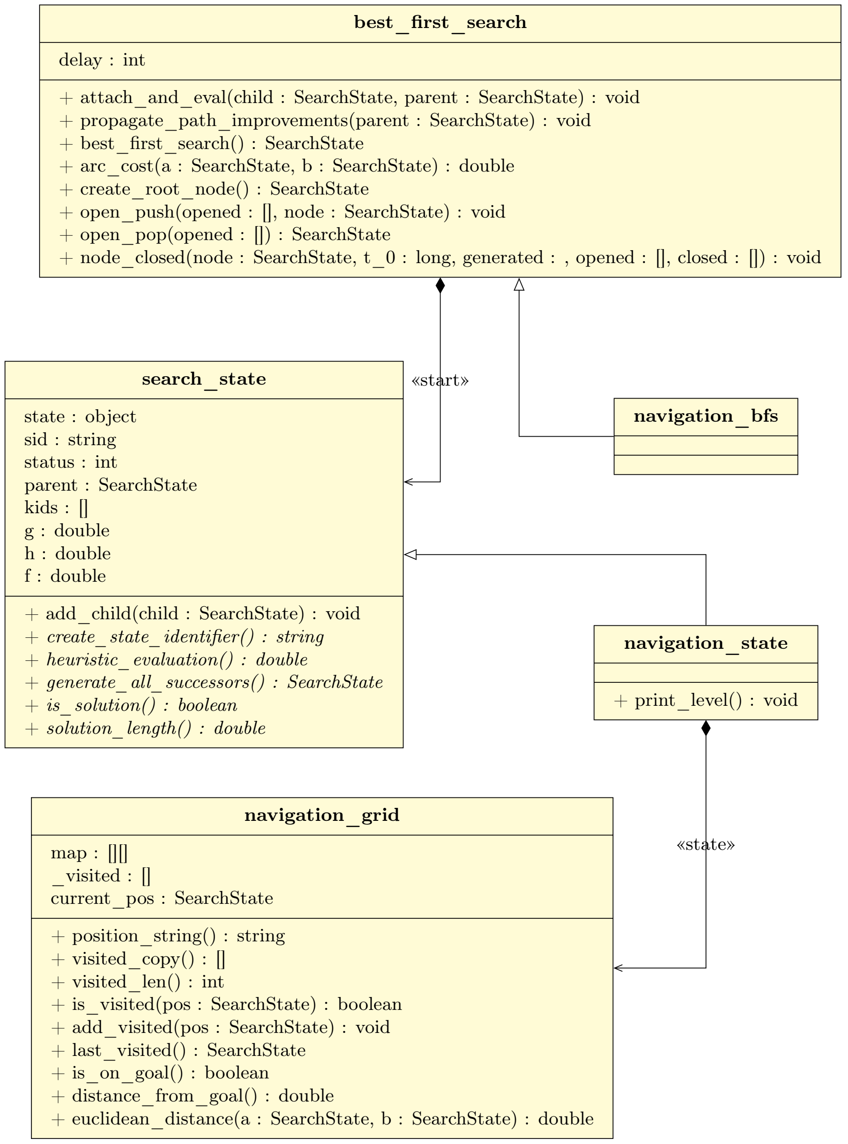
1.2 Generality of the program

Figure **??** shows the class diagram for the navigation problem. As seen, each new problem needs to define two classes that inherit from 'search_state' and 'best_first_search', respectively. Two methods, 'create_root_node' and 'arc_cost', has to be overidden in the subclass of 'best_first_search' for it to work. Likewise, the subclass of 'search_state' has to implement 'heuristic_evaluation', 'create_state_identifier', 'generate_all_successors', 'is_solution' and 'solution_length'. Most of these methods can be reused in similar problems, and this makes almost no effort to have a new type of problem solver up and running in no time.

1.3 Heuristics

Both the *Manhattan distance* and *Euclidean distance* has been tested on the navigation problem. Each adjusted the behaviour of the agent. Using the manhattan distance the agent seem preferring to minimize the amount of $90deg$ turns before reaching the final goal. The euclidean distance seems to do the quite opposite, and the agent goes straight towards the target even though the agent cannot move diagonally. It tries to emulate the behaviour of walking diagonally and follows a diagonal line between the current position and the goal. In the example board number two (Figure 3), the manhattan distance wins by generating least amount of nodes, because it can just move right and then up without encounter an obstacle.

The euclidean agent has to avoid an obstacle and therefore is generating a few more nodes. On the other hand, in example 5 (Figure **??**), the euclidean agent slips through the opening between the two last obstacles,



best_first_search
delay : int
+ attach_and_eval(child : SearchState, parent : SearchState) : void
+ propagate_path_improvements(parent : SearchState) : void
+ best_first_search() : SearchState
+ arc_cost(a : SearchState, b : SearchState) : double
+ create_root_node() : SearchState
+ open_push(opened : [], node : SearchState) : void
+ open_pop(opened : []) : SearchState
+ node_closed(node : SearchState, t_0 : long, generated : , opened : [], closed : []) : void
«start»
search_state
state : object
sid : string
status : int
parent : SearchState
kids : []
g : double
h : double
f : double
+ add_child(child : SearchState) : void
+ create_state_identifier() : string
+ heuristic_evaluation() : double
+ generate_all_successors() : SearchState
+ is_solution() : boolean
+ solution_length() : double
navigation_bfs
navigation_state
+ print_level() : void
«state»
navigation_grid
map : [][]
_visited : []
current_pos : SearchState
+ position_string() : string
+ visited_copy() : []
+ visited_len() : int
+ is_visited(pos : SearchState) : boolean
+ add_visited(pos : SearchState) : void
+ last_visited() : SearchState
+ is_on_goal() : boolean
+ distance_from_goal() : double
+ euclidean_distance(a : SearchState, b : SearchState) : double

Figure 2: Class diagram for the navigation task.

while the manhattan agent needs to explore the area before the right-most obstacle. The performance of the agent depends on the heuristics compared with the structure of the board. Switching the euclidean and manhattan when using diagonal moves, seem to result in similar end results, though either of the two may visit more states than the other.

The arc-cost has to be changed whether the agent can move diagonal or not. A arc-cost below 1 is suitable for strict axis-movements and a arc-cost above 1 for diagonal movement. If the arc-cost is below 1 for diagonal movement, the agent would do unnecessary and dramatic turns which result in wierd behavior.

1.4 Generating successor states

From each viable state, a maximum of eight possible successor states can be created, each by moving in each direction from its currently held position; NORTH, SOUTH, EAST, WEST, NORTH-EAST, NORTH-WEST, SOUTH-EAST, SOUTH-WEST. If the diagonal variable is set, eight states are generates, else only four is generated, as seen in figure 4.

Other considered limitations are that the new position should be inside the given dimensions of the board,

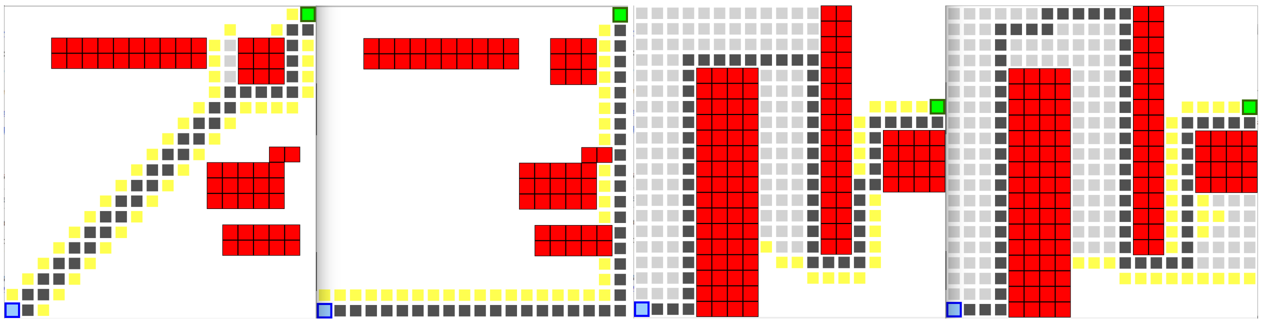


Figure 3: Comparing euclidian distance and manhattan distance in board example 2 and 5, respectively.

and that obstacles are not walkable, together with unnecessity to visit positions that previously have been visited. Every state that is within the scope of those limitations, is created and added to the heap.

```
if self.diagonal:
  viable_movements = [[-1, -1], [1, 1], [1, -1], [-1, 1],
                      [-1, 0], [1, 0], [0, -1], [0, 1]]
else:
  viable_movements = [[-1, 0], [1, 0], [0, -1], [0, 1]]
```

Figure 4: Successor generator vectors.

1.5 Runs on the provided boards

Table 1 summarize the results of a run on each of the provided problems.

A*

	g		e		
board	e	m	e	m	p
0	37		21		19
1					
2					
3					
4					
5					

Depth-first

board	g	e	p
0	73	62	21
1			
2			
3			
4			
5			

Breadth-first

board	g	e	p
0	73	73	19
1			
2			
3			
4			
5			

Table 1: Results of the provided problems in the proplem set. In the tables, ‘g’ is nodes generated, ‘e’ is nodes expanded and ‘p’ is the path length from start to goal.