IT3105 Module 1

Iver Egge
Johan Slettevold

October 2015

1 The A* agenda loop

The A* algorithm was divided into an incremental solver and a complete solver to ease the communication with the gui. The incremental solver allows to solve the problem step by step.

The A* agenda loop works based on the search-type, which can be best-first, depth-first, or breadth-first.

In the three first steps, the loop checks whether the there are items in the open list; then pops a node from a suitable place in the open list and adds it to the closed list, then finnaly checks if the popped node is a solution.

```
# Check whether a path can be found
if not self.open_set:
    return ["FAIL: no path found"]

# Remove the next promising node, X, from open_heap and open_set, then add it to clo
if self.search_type == "best-first":
    X = heapq.heappop(self.open_heap)
    self.open_set.remove(X)
elif self.search_type == "breadth-first":
    X = self.open_heap.pop(0)
    self.open_set.remove(X)
elif self.search_type == "depth-first":
    X = self.open_heap.pop()
    self.open_set.remove(X)
else:
    return ["ERROR: search_type"]
self.closed_set.add(X)

# Look for end properties
if self.graph.goal_found(node=X):
    path = self.retrace_path(X, [X])
    return ["SUCCESS: path found", path]
```

2 Generality of the A*

The generality of the A* can be found in the use of the classes Graph and Node. These two classes provide all help functions that A* needs, and can be implemented to describe any problem. The search type and distance type could be judged as non-general, but were placed in A* because of their natural connection to the algorithm.

```
class AStar:

    # Initialization in two steps for easier variable control
    def __init__(self, graph):
        self.graph = graph
        self.n0 = graph.n0
        self.search_type = "best-first"
        self.max_nodes = 100000

    def initialize(self, distance_type):
        # Initialize the algorithm
        self.distance_type = distance_type
        self.n0.set_f(g=0, h=self.calculate_h(self.n0))
        self.open_set = set()
        self.open_heap = []
        self.closed_set = set()
        self.open_set.add(self.n0)
        heapq.heappush(self.open_heap, self.n0)
```

3 Heuristics function

The heuristics function calculates the h calue for a node based on the distance type. A simple manhattan distance is calculated with the distance from the current x and y value to the goal x and y value.

```
    # This method calculates the h value depending on distance_type
    def calculate_h(self, node):
        if self.distance_type == "manhattan distance":
            x_distance = abs(self.graph.b_pos_x-node.pos_x)
            y_distance = abs(self.graph.b_pos_y-node.pos_y)
            manhattan_distance = x_distance + y_distance
            return manhattan_distance
        elif self.distance_type == "euclidian distance":
            raise NotImplementedError
        elif self.distance_type == "csp":
            h = 0
            for vertex, domain in node.domains.iteritems():
                h += len(domain) - 1
            return h
        else:
            raise NotImplementedError
```

4 Generate successors

The generate successors uses the Node class. There, the successors are generated based on the problem. In module 1 the method looks like this.

```
# Generate all nodes connected to self
    def generate_successors(self):
        successors = set()
        if self.pos_x < self.graph.columns-1:
            right = self.graph.graph[self.pos_x+1][self.pos_y]
            if right.tag is not "X":
                successors.add(right)
        if self.pos_y < self.graph.rows-1:
            below = self.graph.graph[self.pos_x][self.pos_y+1]
            if below.tag is not "X":
                successors.add(below)
        if self.pos_x > 0:
            left = self.graph.graph[self.pos_x-1][self.pos_y]
            if left.tag is not "X":
                successors.add(left)
        if self.pos_y > 0:
            above = self.graph.graph[self.pos_x][self.pos_y-1]
            if above.tag is not "X":
                successors.add(above)
        return successors
```

After the successors are calculated, A* checks if they have been visited earlier, and continues by rating their g, h, and f values This fucntionality uses the rather straight forward methods ”progapage path improvement” and ”attach and eval”.

```
            # Check whether nodes have been visited before.
            # Update the ones that has. Add the rest to open_set and open_heap
            for S in successors:
                X.kids.append(S)
                if S not in self.closed_set:
                    if S not in self.open_set:
                        self.attach_and_eval(S, X)
                        self.open_set.add(S)
                        if self.search_type == "best-first":
                            heapq.heappush(self.open_heap, S)
                        elif self.search_type == "breadth-first":
```

```
            self.open_heap.append(S)
        elif self.search_type == "depth-first":
            self.open_heap.append(S)
        else:
            return ["ERROR: search_type"]
    elif X.g + self.graph.calculate_arc_cost(X, S) < S.g:
        self.attach_and_eval(S, X)
        self.propagate_path_improvements(S)
elif X.g + self.graph.calculate_arc_cost(X, S) < S.g:
    self.attach_and_eval(S, X)
```