AI programming: Module 1

The A* algorithm on a navigation task

Bjørnar Walle Alvestad

2015

Norwegian University of Science and Technology

NTNU

Introduction

This report will describe the central aspects of the A* algorithm implemented in project module 1 of the AI programming course (IT-3105). The assignment consisted of solving the navigation problem using a self-implemented version of the A* algorithm, and displaying generated search paths in a graphical interface. Also documented in this report, is problem specific details such as heuristic function design and navigation states used in the algorithm.

This implementation of A* is written in Java, and is a general version of A*, meaning it can be modified to solve a variety of search problems.

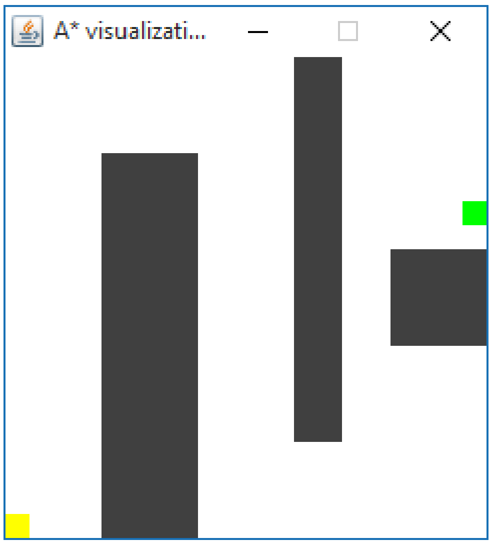

Figure 1: An unsolved maze. Start position is illustrated as a yellow square. The goal position is shown as a green square.

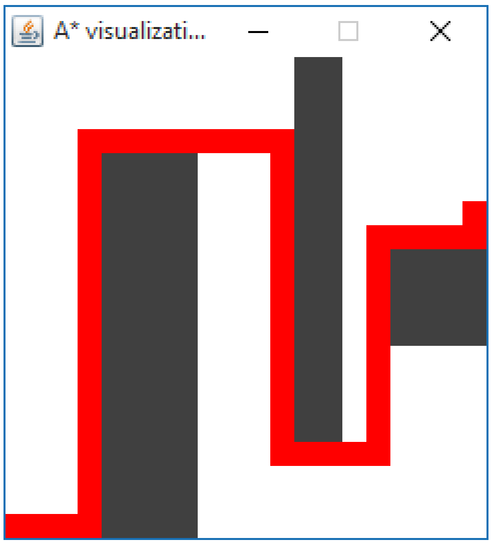

Figure 2: Solved maze with shortest path shown in red.

The general A* algorithm

The A* algorithm is a best-first search algorithm, invented in the 1960's at the Stanford Research Institute. It was initially designed to find the shortest path possible in a navigation task, but it was later realized that the algorithm could be used to solve a variety of search related problems.

A* states

When executed, the algorithm will generate a search tree, where each node denotes a specific state in the search process. The state configuration and content is highly dependent on the type of problem being solved. In this implementation, the state contains the current search position as a 2D coordinate. The abstract class *AstarState* defines the problem specific operations available to the algorithm. When implementing a new search problem, this abstract class must be implemented, and the listed methods overridden. The *AstarState* class enforces the following methods:

- `float heuristic()`
  Executes the heuristic function with respect to this state. It calculates an estimate cost of getting from the current state, to the goal state.
- `boolean isSolution()`
  Checks if this state is a solution state. (This state is the goal state).

```
AstarState:
float heuristic();
boolean isSolution();
AstarState[] generateChildren();
int id();
float arc_cost(AstarState other);
```

- `AstarState[] generateChildren()`
  Generates valid successor states. This is accomplished by applying problem specific operators to the current state.
- `int id()`
  Generates an id unique for this state.
- `float arc_cost(AstarState otherState)`
  Calculates the cost of getting from this state to the *otherState*.

Node

```
Node:
public Node                 parent;
public ArrayList<Node> childs;
public AstarState           state;
public Status               status;
//Status = { OPEN, CLOSED }
public float g;
public float h;
public float f;
```

The search tree is composed of *Node* objects, which is connected by parent-child references. All nodes will have exactly one parent node (except for the root), and one state object (*AstarState*). The *status* field marks the node as *OPEN* or *CLOSED*, depending on whether or not that node have had its children generated. References to potential children nodes are stored in the *childs*-list.

The float data member *g* denotes the cost of traveling from the start state to the state denoted by this node. *h* is an estimated cost for traveling from the current state to the goal state. *f* is computed as following: $f(n) = g(n) + h(n)$, where *n* is a node in the search tree.

The open-node list

At the core, the A* algorithm consists of an *open-node* list and a *closed-node* list. The open-list is specified as an abstract class, and later implemented in the classes *Agenda*, *Queue*, and *Stack*. These are data containers with different rules regarding insertion and removal of data elements. The different search modes of A* requires different types of data structures serving as the open list:

- **Best first mode:**	*openList = Agenda.*
- **Breadth first mode:**	*openList = Queue.*
- **Depth first mode:**	*openList = Stack.*

The abstract class *OpenList* enforces the following methods:

```
OpenList:
void add(Node n);
Node get();
int size();
boolean contains(Node n);
void sort(List l);
```

- `void add(Node n)`
  Adds a new node to the data structure. A stack will add the new node at the top/start. A queue will add the new node at the end. An agenda will add the node at the end, then sort the list in ascending order based on the *f* values.
- `Node get()`
  Retrieves and removes the appropiate node from the data structure. A stack will pop the top/start node. A queue will retrieve the node at the end. An agenda retrieve the top/start node, because that will have the lowest *f* value (due to ascending sorting).
- `int size()`
  Returns the number of elements in the data structure.
- `boolean contains(Node n)`
  Searches the data structure for the specified node element. Nodes are consdiered equal if the underlying state-object of the nodes are equal.

- `void sort(List l)`
  Sorts the specified list in ascending order based on the elements *f* value.

Problem specification

The navigation problem introduces a 2-dimensional board with size X*Y, a series of obstacles, a start position and a goal position. The task is to find the short path from start to goal, if possible. The board is represented using X*Y squares, where each square is either open (not obstacle), or closed (is obstacle). This is represented in the application in a 2-dimensional array of boolean type.

The abstract class *AstarState* is implemented in the class *MazeState*. This class defines all problem specific details and logic. It contains one data member; position, which denotes the position of this state on the board.

The heuristic function

```
public float heuristic() {
    return manhattanDist(position, board.goal);
}

private int manhattanDist(Point p1, Point p2) {
    return Math.abs(p1.x - p2.x) +
           Math.abs(p1.y - p2.y);
}
```

The heuristic function is implemented in the method *`heuristic()`*. In this navigation problem, moving up, down, left and right are considered legal moves. No diagonal movement. Therefore, a good heuristic function for estimating the distance to goal, is the Manhattan distance between the current position and the goal position. This is also guaranteed to be an underestimate (or perfect estimate) of the actual distance, because diagonal movement is illegal and the fact that Manhattan distance ignores obstacles that may obscure the estimated path.

Successor states generation

Successor states are generated using the *`generateChildren()`* method. Legal operators in this navigation problem is moving in either direction along an axis. The *generateChildren* method creates child-states by creating a new *MazeState* object, move the position in that object and then evaluating if the position is valid. A valid position must fulfil the following requirements:

- Non-negative position coordinates
  (`x >= 0` AND `y >= 0`).
- Position is inside board bounds
  (`x < board.X` AND `y < board.Y`).
- Position is not blocked by an obstacle.

```
private boolean isValidPos(Point p) {
    if (p.x >= board.X || p.x < 0 ||
        p.y >= board.Y || p.y < 0)
    return false;
    return board.isFree(p.x, p.y);
}
```

This procedure is repeated for all the movement directions (up, down, left, right). States with valid positions are added to a list, which is in turn returned to the method caller.

References

Downing, Keith L., *AI Programming (IT-3105) Project Module #1: Implementing and Testing the A* Algorithm on a Navigation Task.* [Online]

Downing, Keith L., *Essentials of the A* Algorithm*. [Online].