A* ALGORITHM

Print this file

THEORY

The A* algorithm is a path-finding algorithm whose purpose is to find the shortest path from start to goal (e.g. in the pac-man game start = ghost position whereas goal = pac-man position).
The A* algorithm repeatedly examines the “most promising” (lowest cost) unexplored location it has seen so far. When a location is explored, the algorithm ends when the location that is currently exploring represents the goal; otherwise, the algorithm makes note of all that location’s neighbors for further exploration.
More precisely, A* keeps track of 2 lists of states, called OPEN and CLOSE, for unexamined and examined states, respectively. At the start, CLOSE is

empty, and OPEN has only the starting state. In each iteration the algorithm removes the most promising state from OPEN for examination. If the state is not the goal, its neighbor locations are examined: if they're new, they're placed in OPEN whereas if they're already in OPEN, information about those locations are updated, if this is a cheaper path to them. Locations in CLOSE are ignored if their cost is higher than their previous one; otherwise they're re-opened (removed from CLOSE and pushed into OPEN). If the OPEN list becomes empty before the goal is found, it means there is no path to the goal from the start location.
Each state X includes the following information to determine the shortest path: the cost of the cheapest path that has led to this state from the start (which we'll call costFromStart(X), in literature called g(X)); a heuristic estimate costToGoal(X) (in literature called h(X)) that is the cost of the remaining distance from X to the goal; and finally the totalCost (also called f(X)), defined as
CostFromStart + CostToGoal = totalCost
In addition each state keeps a pointer to its parent state; when a goal state is found, these links can be traced back to the start in order to build the path from start to goal.

HEURISTIC

The idea behind the heuristic cost is to estimate the true cost from a particular node to the goal. It's

important to choose a good heuristic function. If you always knew the real cost to the goal, A* will only follow the best path and never expands anything else, without wasting any search time going down the wrong path, making it very fast. But if the heuristic estimate happens to overestimate the real cost, the heuristic becomes “inadmissible” and the algorithm might not find the optimal path (and might find a terrible path), but it may run faster. If the heuristic part of the total cost is bigger than it should be, it distorts the reasoning by which nodes on the OPEN list are picked off. Since A* always picks the node with the least total cost, this distortion promotes nodes closer to the goal to be picked (since we don’t have weights on edges).

The way to guarantee that the heuristic cost is never overestimated is by calculating the Euclidean distance between the node and the goal. When coding A* for the first time, this is the best thing to do until it’s time to optimize. Since the cost will never be more than this distance, the optimal path will always be found.

The lower h(X), the more node A* expands making it slower. If h(X) is too little, then we’ll continue to get shortest path, but slow down the things. If h(X) is too high, then we give up shortest path, but A* will run faster.

Note: if you set the heuristic to return zero, you will never overestimated the distance to goal, but what you will get is a simple search of every node generated at each step (Dijkstra’s alg).

A* algorithm will not only find a path, if there is one, but it will find the shortest path (because it has the heuristic presence).
An optimization respect Euclidean distance is Manhattan distance following explained.
Heuristic for grid maps --> MANHATTAN DISTANCE

h(n) = D * (abs(n.x – goal.x) + abs(n.y – goal.y))
D = minimum cost for moving from one space to an adjacent space

IMPLEMENTATION

As already said, A* makes use of 2 sets, OPEN and CLOSE, in order to take care about nodes examination, and so I do. The OPEN set contains those nodes that are candidates for examining while the CLOSE set those that have already been examined. Initially, the OPEN set contains just one element: the starting position and the CLOSE set is empty. Each node also keeps a pointer to its parent node so that we can determine how it was found. There is a main loop that repeatedly pulls out the best node n in OPEN (the node with the lowest f value) and examines it. If n is the goal, then we're done. Otherwise, node n is removed from OPEN and added to CLOSE. Then its neighbors n' are examined. If n' don't belong neither to OPEN nor CLOSE then put n' in OPEN; if n' belong to OPEN and it has a lower cost, then the value in the OPEN set must be adjusted. The adjustment operation involves removing the node,

updating it and re-inserting the node into OPEN set. If n' belong to CLOSE and it has a lower cost, then pop it from CLOSE e push it into OPEN (re-open it) with its associated info.
I used two hashtable to implement the 2 sets OPEN and CLOSE in order to prevent having to do a linear search. The hash table has an hash function equals to totalCost's node.
The source code was implemented in J2ME.

Bibliography:

Bibliography references from the following sources:

- Amit's thoughts on path-finding and A-star
  http://theory.stanford.edu/~amitp/GameProgramming
- A* algorithm tutorial (Justin Heyes-Jones)
  http://www.geocities.com/jheyesJones/astar.html
- Game Programming Gems – Mark A. DeLoura – Charles River Media
- Mobile Phone Game Programming – Michael Morrison - SAMS

PSEUDO-CODE

```
Set startNode
Set goalNode
Push startNode into OPEN
While OPEN !empty
{
```

ExtractNode = pop the lowest cost node from OPEN;
if(ExtractNode == goalNode)
{
I've found a path;
construct a path backward from ExtractNode to StartNode;
}
else
{
examine ExtractNode neighborhood (up, down, left, right position)
for each neighbor
{

- if it's an obstacle then don't consider this neighbor --> continue;
- check if the node belongs to OPEN;
- check if the node belongs to CLOSE;
- if the node already belongs to OPEN or CLOSE and its cost is higher than that inserted then continue (don't consider it);
- if the node already belongs to CLOSE and its cost is lower than that inserted then re-open it (that is remove it from CLOSE and insert it into OPEN);
- if the node already belongs to OPEN and its cost is lower than that inserted one then update it;

- if the node doesn't belong neither to OPEN nor to CLOSE then insert it into OPEN;

}

}

push ExtractNode in CLOSE

}

SOURCE CODE - Java programming language (J2ME environment for mobile system)

You can test this program by downloading the Sun Java Wireless Toolkit (http://java.sun.com/j2me/download.html)

the game is still a work in progress so, unfortunately, it presents some issues as well

HSCanvas.java	
Node.java	
Point.java	
ScrambledMIDlet.java	

Resources