Artificial intelligence in games

Pathfinding

Piotr Fulmański

piotr@fulmanski.pl

3 listopada 2016

Table of contents

Description of the problem

Pathfinding is the finding a shortest route between two given points S (starting) and F (final). An initial data set consist of

state space – a data structure which „describes” the area where we will search for the solution;

starting point – initial position (point) in our state space;

final point – destination position (point) in our state space; the location we want to reach.

As a result we want to get path (sequence of intermediate points) from a starting to a final point.
Please notice that in this presentation terms: position, point and location will be used interchangeably.

A^* or not A^*?

The most common name of the algorithm I'm going to describe in this presentation is A^*. Despite this you should know that it can be classified to certain more general classes of pathfinding algorithms and in consequence can have different names.
For example the A^* algorithm is an example of best-first search algorithm which explores a graph by selecting and expanding *the most promising node* chosen according to a specified rules. Some rules and conditions imposed on selecting the node we call *the most promising node* turns best-first search into A^*.
On the other hand a well known Dijkstra's algorithm can be treated as an uninformed, less powerful, special case of the A^* search algorithm.

The search space

Generally speaking, the search state space is a graph where different nodes are different locations and edges are possible path (transitions) from one location to the other.

The search space

For tile games, where locations are squares, rhombus or hexagons, we can think about search state space as an array.

Calculating path cost

In this algorithm the location cost (evaluation function) $f(n)$ related to location n is calculated based on two factors.

- $g(n)$ cost-to-come which is the movement cost to move from the starting point S to a point n. This value is precise and well known because in this algorithm we know the path from starting point to a location n being evaluated (we evaluate points only from a special set – more about this in the next part of this material).
- $h(n)$ cost-to-go which is the estimated movement cost to move from a point n to the final point F. This is often referred to as the *heuristic* (ancient greek: *find* or *discover*). The reason it is so called that is because it is a guess. We really don't know the actual distance. Yes, after all we are just searching the path, co we don't know the final or any partial cost.

More about h function

Value of the h function can be estimated in a variety of ways. One of the simplest we can use is the Manhattan method where we calculate the sum of horizontal and vertical difference in coordinates between two points. In a square world (tail games with square locations) this is equal the total number of squares moved horizontally and vertically to reach the final square from the starting square, ignoring all diagonal movement, and obstacles that may be in the way.

More about h function

Other option is Euclidean distance calculated simply as we do when we want to find a distance bewten points on a plain with a ruler.

More about *h* function

Inadmissible heuristic

We can think about the heuristic as a rough estimate of the distance between two points. Of course we are trying to estimate the distance along the path as precise as we can. The closer our estimate is to the actual remaining distance, the faster the algorithm will be. If we overestimate this distance, however, it is not guaranteed to give us the shortest path. In such cases, we have what is called an *inadmissible heuristic*.

The Manhattan method is inadmissible because it slightly overestimates the remaining distance. Despite this it is used because of simplicity (it is easy to calculate and explain it), and because it is only a slight overestimation. On the rare occasion when the resulting path is not the shortest possible, it will be nearly as short. Please note also that pathfinding is very local in a sense that we search a path for a certain situation in a game. In most cases games are not static and the game world changes constantly so the path we have found 1-2 seconds ago could be now outdated – it doesn't matter if it was optimal or not.

Initialization

To initialize the algorithm we have to

1. Initialize `OPEN` data structure as a priority queue; priority is a number, the lower it is, the more importane queue's element is;
2. Initialize `CLOSED` data structure as a set data structure (or any other data structure such that we can verify if a given element belongs to it or not);
3. Evaluate point S (calculate $f(S)$ for it) and add it to the `OPEN`.

Main loop

4. As long as `OPEN` is not empty repeat
    1. Choose the best point (the one with the highest priority which means the lowest evaluation value f) from all those that are in the `OPEN` queue, remove it from `OPEN`, call it n and add it to `CLOSED`.
    2. If n is the final point, backtrace path to n (through recorded parents) and return path.
    3. For each `child` from n's children
        1. If `child` is not in `CLOSED`, and it is not in `OPEN` then evaluate it, add it to `OPEN`, and save information about its parent.
        2. Else if `child` is in `OPEN` then evaluate it, adjust its priority in `OPEN` using this new evaluation and change its recorded parent.
        3. Else `child` is in `CLOSED` then do nothing with it.
5. There is no path from S to F; return an empty path.

A^* summary

Now we are going to formulate A^* algorithm as a pseudocode. We will use the following calls

- `open.add(name, priority, parent)`
  to add point `name` whose parent is `parent` to the priority queue with priority `priority`.
- `open.update(name, priority, parent)`
  to update point `name`: set its parent as `parent` and priority as `priority`.
- `n.getChildren()`
  to get all points connected directly with point `n`.

A^* summary

Algorithm 1 Pseudocode for the A^* algorithm (part 1/2)

```
1: function findPathWithAStar
2:     $open$ := new PriorityQueue
3:     $closed$ := new Set
4:     $open.add(S, eval(S), null)$
```

A^* summary

Algorithm 2 Pseudocode for the A^* algorithm (part 2/2)

```
 5:     while (!$open.isEmpty()$) do
 6:         $n := open.get()$                                        ▷ Step 4.1
 7:         $closed.add(n)$
 8:         if $n = F$ then
 9:             return backtracked path
10:         end if
11:         for all $child$ in $n.getChildren()$ do
12:             if !($closed.has(child) || open.has(child)$) then
13:                 $open.add(child, eval(child), n)$
14:             else if $open.has(child)$ then
15:                 $open.update(child, eval(child), n)$
16:             end if
17:         end for
18:     end while
19:     return null                             ▷ There is no path from $S$ to $F$
20: end function
```

A^* example

Definition of g and h

A^* example

Definition of state space

A^* example
Initialization

We do steps 1-3 from the algorithm (2-4 from pseudocode). Point A is our starting point S, point L – final F. Euclidean distance from S to F is 5.

```
open = [(A,(5=0+5), null)]
closed = []
```

A^* example
Loop iteration: 1

Remove the best (most promising) point – A point – from open queue and add it to `closed`. Find all children of point A and add them ($B1$, $B2$, $B3$, $B4$) to open queue.

```
open = [(B4,(5=1+4), A),
        (B1,(6.09=1+5.09), A),
        (B3,(6.09=1+5.09), A),
        (B2,(7=1+6), A)
       ]
closed = [(A,(5=0+5), null)]
```

A^* example

Loop iteration: 2

Remove the best point ($B4$) from open queue and add it to `closed`.
Find all children of point $B4$ and add them (C) to open queue.

```
open = [(C,(5=2+3), B4),
         (B1,(6.09=1+5.09), A),
         (B3,(6.09=1+5.09), A),
         (B2,(7=1+6), A)
        ]
closed = [A, B4]
```

A^* example

Loop iteration: 3

Remove the best point (C) from open queue and add it to `closed`. Find all children of point C and add them (D) to open queue.

```
open = [(D,(5=3+2), C),
        (B1,(6.09=1+5.09), A),
        (B3,(6.09=1+5.09), A),
        (B2,(7=1+6), A)
       ]
closed = [A, B4, C]
```

A^* example

Loop iteration: 4

Remove the best point (D) from open queue and add it to `closed`. Find all children of point D and add them ($E1$, $E2$) to open queue.

```
open = [(B1,(6.09=1+5.09), A),
        (B3,(6.09=1+5.09), A),
        (E1,(6.23=4+2.23), D),
        (E2,(6.23=4+2.23), D),
        (B2,(7=1+6), A)
       ]
closed = [A, B4, C, D]
```

A^* example

Loop iteration: 5

Remove the best point ($B1$) from open queue and add it to closed. Find all children of point $B1$ and add them (F) to open queue[a].

```
open = [(B3,(6.09=1+5.09), A),
        (E1,(6.23=4+2.23), D),
        (E2,(6.23=4+2.23), D),
        (B2,(7=1+6), A),
        (F,(7.38=2+5.38), B1),
       ]
closed = [A, B4, C, D, B1]
```

[a]Please do not confuse F wich is a child of $B1$ with our goal F.

A^* example

Loop iteration: 6

Remove the best point ($B3$) from open queue and add it to `closed`. Find all children of point $B3$ and add them (G) to open queue.

```
open = [(E1,(6.23=4+2.23), D),
        (E2,(6.23=4+2.23), D),
        (B2,(7=1+6), A),
        (F,(7.38=2+5.38), B1),
        (G,(7.38=2+5.38), B3),
       ]
closed = [A, B4, C, D, B1, B3]
```

A^* example

Loop iteration: 7

Remove the best point ($E1$) from open queue and add it to `closed`. Find all children of point $E1$ and add them (H) to open queue.

```
open = [(E2,(6.23=4+2.23), D),
        (B2,(7=1+6), A),
        (F,(7.38=2+5.38), B1),
        (G,(7.38=2+5.38), B3),
        (H,(7.82=5+2.82), E1),
       ]
closed = [A, B4, C, D, B1, B3,
          E1]
```

Remove the best point ($E2$) from open queue and add it to `closed`. Find all children of point $E2$ and add them (I) to open queue.

```
open = [(I,(6.41=5+1.41), E2),
        (B2,(7=1+6), A),
        (F,(7.38=2+5.38), B1),
        (G,(7.38=2+5.38), B3),
        (H,(7.82=5+2.82), E1),
       ]
closed = [A, B4, C, D, B1, B3,
          E1, E2]
```

A^* example

Loop iteration: 9

Remove the best point (I) from open queue and add it to `closed`. Find all children of point I and add them (J) to open queue.

```
open = [(B2,(7=1+6), A),
         (J,(7=6+1), I),
         (F,(7.38=2+5.38), B1),
         (G,(7.38=2+5.38), B3),
         (H,(7.82=5+2.82), E1),
       ]
closed = [A, B4, C, D, B1, B3,
          E1, E2, I]
```

A^* example

Loop iteration: 10

Remove the best point ($B2$) from open queue and add it to `closed`. Find all children of point $B2$ and add them (K) to open queue.

```
open = [(J,(7=6+1), I),
         (F,(7.38=2+5.38), B1),
         (G,(7.38=2+5.38), B3),
         (H,(7.82=5+2.82), E1),
         (K,(9=2+7, B2)
        ]
closed = [A, B4, C, D, B1, B3,
           E1, E2, I, B2]
```

A^* example
Loop iteration: 11

Remove the best point (J) from open queue and add it to `closed`. Find all children of point J and add them ($L1$, $L2$) to open queue.

```
open = [(L1,(7=7+0), J),
         (F,(7.38=2+5.38), B1),
         (G,(7.38=2+5.38), B3),
         (H,(7.82=5+2.82), E1),
         (L2,(8.41=7+1.41, J),
         (K,(9=2+7, B2)
       ]
closed = [A, B4, C, D, B1, B3,
          E1, E2, I, B2, J]
```

A^* example

Loop iteration: 11

Remove the best point ($L1$) from open queue and add it to `closed`. Because point $L1$ is our final point F we can return backtracked path (A, $B4$, C, D, $E2$, I, J, $L1$)

```
open = [(L1,(7=7+0), J),
        (F,(7.38=2+5.38), B1),
        (G,(7.38=2+5.38), B3),
        (H,(7.82=5+2.82), E1),
        (L2,(8.41=7+1.41, J),
        (K,(9=2+7, B2)
       ]
closed = [A, B4, C, D, B1, B3,
          E1, E2, I, B2, J, L1]
```

Why do we have a $*$ in the name of the algorithm?

The star $(*)$ symbol cames from optimization theory where is used to denote optimal solution of something. Strictly speaking, the algorithm described so far is an A algorithm. We can turn it into optimal A^* if we choose correct heuristic function h. From theoretical point of view, h should be *admisible* heuristic, i.e., such that $h(n) \leq h^*(n)$. The A search with an admissible heuristic is called A^*, which is guaranteed to be optimal.
How to find an admissible heuristic? A hint is that such heuristic function shoud never overestimates the actual cost-to-go. A good examples of admissible heuristics are

- $h(n) = 0$ this always works, but it's not hard to guess that it is not very useful because reduced A^* algorithm to so called *Uniform Cost Search* algorithm which is complete and optimal but not to fast.
- $h(v) = \text{distance}(n, F)$, when the vertices of the graphs are physical locations.
- $h(v) = ||n - F||_p$, when the vertices of the graph are points in a normed vector space.

Precalculated pathfinding

It turns out that much of information needed for pathfinding is static, and can be precalculated ahead of time. In most cases a game world is static or semi-static in a sense that changes are prepare for game designers so they know how it would evolve. In these circumstances some pathfinding procedures, even time consuming, can be done before a game will go on sale.

Precalculated pathfinding

Idea: optimal substructure

One thing worth to note is that pathfinding problem has *optimal substructure*: we can take any subsegment B-C on a shortest path A-D and that subsegment is guaranteed to be the shortest path between two points B-C located on path A-D.

Example

Given that the route A-B-C-D is the shortest route from A to D, the shortest route from B to D is B-D even if there are other alternative paths. In some sense it has recursiv nature: any given shortest route is made up of smaller shortest routes. If there exist a shortest route from B to D, then the shortest route from A to D would have to be

```
route(A, D) := findRoute(A, B) + route(B, D)
```

Precalculated pathfinding

Idea: one step ahead

Second key obervation very important to note is that while we're at A, it doesn't actually matter to us what the shortest route from B to D is. We don't need to calculate `route(B, D)`. We don't need any part of it until we are actually at B, and even then on the same way, we only need to know the first step of the route. So we don't need the whole path but only information about next step. Calculating the whole path could be a waste of time if, for example, we would change our target in a next step.

Precalculated pathfinding

Lookup table (example: state space)

Consider the sample state space depicted as a graph

A	B, E
B	A, C, E
C	B, D
D	C, E
E	A, B, D

```
A-B-C
 \| |
  E |
   \|
    D
```

Precalculated pathfinding

Lookup table (example: generation)

Having a state space we can use any pathfinding algorithm to generate a lookup table. This table is built by finding the path from each given node to each given node, and storing the first (non-starting) node in the path.

	A	B	C	D	E
A		B	B	E	E
B	A		C	E	E
C	B	B		D	B
D	E	E	C		E
E	A	B	B	D	

From this table we know that E is a first non-starting node on the best path from D to B.

Precalculated pathfinding

Lookup table (example: usage)

To use the lookup table, all we need to know is our current location (starting point), and our destination (finall point). Current location determines a row in lookup table, while destination a column. At the intersection of row and column we can find a node we should go to.

When the state space is dynamic, i.e. new nodes or edges can (dis)appear, lookup table also should be dynamic. This means that we should recalculate it. It might be a problem. The question is: *How to update lookup table to make as little changes as it is possible?*
In the next part we will explain a simple algorithm we can use to solve this problem.

Precalculated pathfinding

Dynamic state space (example: initialization)

As for A^* algorith, we need two lists: an `open` and `closed`. The `open` list represents nodes that we need to check for updates, and the `closed` list represents nodes we have already checked.
Now let's say that connection from A to E was removed. In consequence both nodes should be added to `open` list, while `closed` should be empty.

```
open = [A, E]
closed = []
```

Precalculated pathfinding

Dynamic state space (example: iteration 1)

Select and remove first node from `open` (A), add it to `closed` and rebuild its row in the lookup table. Because the row has changed, we have to add all node's neighbors which are not yet opened or closed to the `open` list.

```
open = [E, B]
closed = [A]
```

The oryginal row

	A	B	C	D	E
A		B	B	E	E

The rebuilded row

	A	B	C	D	E
A		B	B	B	B

Precalculated pathfinding
Dynamic state space (example: iteration 2)

Select and remove first node from `open` (E), add it to `closed` and rebuild its row in the lookup table. Because the row has changed, we have to add all node's neighbors which are not yet opened or closed to the `open` list.

```
open = [B, D]
closed = [A, E]
```

The oryginal row

	A	B	C	D	E
E	A	B	B	D	

The rebuilded row

	A	B	C	D	E
E	B	B	B	D	

Precalculated pathfinding

Dynamic state space (example: iteration 3)

Select and remove first node from `open` (B), add it to `closed` and rebuild its row in the lookup table. Because the row has changed, we have to add all node's neighbors which are not yet opened or closed to the `open` list.

```
open = [D]
closed = [A, E, B]
```

The oryginal row

	A	B	C	D	E
B	A		C	E	E

The rebuilded row hasn't changed

	A	B	C	D	E
B	A		C	E	E

Precalculated pathfinding

Dynamic state space (example: iteration 4)

Select and remove first node from `open` (D), add it to `closed` and rebuild its row in the lookup table. Because the row has changed, we have to add all node's neighbors which are not yet opened or closed to the `open` list.

```
open = []
closed = [A, E, B, D]
```

The oryginal row

	A	B	C	D	E
D	E	E	C		E

The rebuilded row hasn't changed

	A	B	C	D	E
D	E	E	C		E

Precalculated pathfinding

Dynamic state space (example: summary)

Notice that we didn't have to check node C. To summarize: two rows was recalculated (red ones in the table below), two other was examined but hasn't changed (yellow) and one left untouched (white).

	A	B	C	D	E
A		B	B	B	B
B	A		C	E	E
C	B	B		D	B
D	E	E	C		E
E	B	B	B	D	