# Path Finding Algorithm Implementation

**Presented by: Henry Fernando Granados**

**09/05/2014**

# Table of Contents

# 1. History:

Well as a quick overview of this algortihm we can say that in 1968 Nils Nilsson suggested a heuristic approach for Shakey the Robot to navigate through a room containing obstacles. At that time this path-finding algorithm was called A1, it was a faster version of the then best known formal approach, Dijkstra's algorithm, for finding shortest paths in graphs. Why not mention that Bertram Raphael suggested some significant improvements upon this algorithm, calling the revised version A2. We should also mention that Peter E. Hart introduced an argument that established A2, with only minor changes, to be the best possible algorithm for finding shortest paths. Hart, Nilsson and Raphael then jointly developed a proof that the revised A2 algorithm was optimal for finding shortest paths under certain well-defined conditions.

## 2. Pseudocode:

The Pseudocode of the algorithm is:

```
function A*(start,goal)
    closedset := the empty set    // The set of nodes already evaluated.
    openset := {start}    // The set of tentative nodes to be evaluated, initially containing
the start node
    came_from := the empty map    // The map of navigated nodes.

    g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
    f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

    while openset is not empty
        current := the node in openset having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score < g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor,
```

```
goal)
            if neighbor not in openset
               add neighbor to openset

   return failure

function reconstruct_path(came_from, current_node)
   if current_node in came_from
      p := reconstruct_path(came_from, came_from[current_node])
      return (p + current_node)
   else
      return current_node
```

# 3. Introduction:

There are several situations where is necessary to find a good (or the better) path from the point A to the point B. Games where the simulation of the movement of the non player characters (NPC) is crucial, simulation of traveling (like the salesman system, or fireman system) are good examples of system where the search of a good or the best path could be important. Pathfinding addresses the problem of finding a good path from the starting point to the goal. Not just avoiding obstacles, also minimizing costs.

The present describe my approach to this problem I develop a complete system to calculate the best path from point a to the point B using the A* pathfinding algorithm. Also, I developed an additional mechanism to test different ways to process the adjacent nodes from any given point. Also a complete graphical interfase was developed. Finally several commands were developed to test these mechanism.

# 4. General Design of the Solution

The system was developed using basically the classes described below.

## 4.1.CMap

This class handles the map topology. Include methods to load and display the map. Also include a set of methods to handle the set of adjacent points. These point are stored in a vector and they are sorted by cost. This class includes the following methods:

### 4.1.1. load (string filename)

This method load the specified file name. The file has the following format

Num_Rows Num_Columns

Where Num_Rows is the number of rows of the map and

Num_Columns is the number of columns

These two values must be in the same line. Next each row should include the values of the rows, in the form of 0’s and 1’s where a 0 means an empty space and 1 means a obstacles or walls.

### 4.1.2. obstructed (int x, int y)

This method returns true if the space indicated by x, y is obstructed false otherwise

### 4.1.3. visual (CNodeLocation bestloc, CNodeLocation goal, vector<CNodeLocation>& road)

This method builds a path from the point bestloc to the goal. It return true if there are no points obstructed between both points (and return a vector with the points) and false otherwise.

#### 4.1.4. insertRoad (vector<CNodeLocation> road)

This method inserts the given road in the map. This is for graphics proposes.

#### 4.1.5. setnode (CNodeLocation location)

This method set the current location to be analized. It store the current location and call the method find that will search the adjacent nodes and store it in the vector sort_tbl. This vector is sorted by cost.

#### 4.1.6. find (CNodeLocation location)

This method find and store in the vector sort_tbl the adjacent points.

#### 4.1.7. next (CNodeLocation bestnode_location)

This method return the next best position in the current set of adjacent points sdtored in the vector sort_tbl.

### 4.2. CAStar

This class handle the method associated to the A* search algorithm. This algorithm finds a least-cost path from a given initial node to the goal node. A heuristic cost function is implemented using distance from the starting node to the current node.

The class includes a node bank with pre allocated nodes (to increase the speed and memory handling) and a MasterNodeList a special map to store the pointers to the node bank given the location. For this a hash function is used.

This class includes the following methods:

### 4.2.1. FindPath (CNodeLocation start, CNodeLocation goal)

This method executes the main find path algorithm. Basically this method execute the following algorithm (pseudocode) (based on [Wikipedia] slightly modified using ideas from [Rabin])

if path not initialized

if there are visual between start and goal

build road and return sucess

initialize variables

Push in Priority Queue the startnode

While there are nodes in the Priority Queue

Bestnode = pop node from Priority Queue

If Bestnode == goal

Construct Path To Goal and return success

For each node adjacent to Bestnode

If node is not in open set and node cost is better

Add node to open set

if we don´t find the goal return failed

### 4.2.2. ConstructPathToGoal (CNodeLocation start, CAStarNode* bestnode)

Store in m_road the nodes from bestnode to the goal

### 4.2.3. CostFromNodeToNode (CAStarNode* newnode, CAStarNode* bestnode)

Calculate the cost of this node. It use euclidean distance = sqrt (x * x + y * y)

#### 4.2.4. GetNodeFromMasterNodeList (CNodeLocation node_location)

Return the pre allocated node of this location (node_location).

#### 4.2.5. StoreNodeInMasterNodeList (CAStarNode* newNode)

Store the node in the master node list. Use a hash function to calculate the index based on the position of the node (x, y)

#### 4.2.6. GetFreeNodeFromNodeBank (void)

Get a free node from the node bank

### 4.3. PriorityQueue

This class handles the priority queue, the implementation for the open and closed set. There are three main operations involved with the open set: search the best node, remove a node and insert a node. Insertion and remove are operations typical of a priority queue. All this operations must be executed keeping the queue sorted by priority, or, in our case, by cost of the node. Actually as the queue is sorted all the time, the first operation (search the best node) it's just take the first node of the queue.

This class is implemented basically with the push_heap STL function. This function allow to sort part of the container using a user provided function.

This class includes the following methods:

#### 4.3.1. GetFreeNodeFromNodeBank (void)

Extract the node at the beggining of the queue. This node has the best cost

#### 4.3.2. PushPriorityQueue (CAStarNode* node )

Insert the node in the queue

#### 4.3.3. UpdateNodeOnPriorityQueue (CAStarNode* node)

Search the node in the priority queue and update it.

### 4.4. LLQueue

This class handles the priority queue, as well (same the previous class) but this time is implemented using a linked link. The linked list is very fast doing insertions but very bad doing the updating and the removing. So we added to this implementation a hash table, implemented with the STL std::map, and using a similar approach that the one used for handle the master node list.

This class has the same methods as the previous class so we omit here to describe it again.

### 4.5. IPriorityQueue

Finally this class is an interfase for the priority queue needed to store the open list. This class define the following methods using the C++ virtual declaration:

#### 4.5.1. GetFreeNodeFromNodeBank (void)

#### 4.5.2. PushPriorityQueue (CAStarNode* node )

#### 4.5.3. UpdateNodeOnPriorityQueue (CAStarNode* node)

Each class (PriorityQueue and LLQueue) inherits from IPriorityQueue and implement those methods. So that means that I could do something like the following in the main function:

```
if (priorityqueue_flag) {
as.m_openlist = new PriorityQueue;
} else { // LLQueue is the default
as.m_openlist = new LLQueue;
}
```

This way I have an mechanism to decide which implementation I want to use and test.

# 5. Results:

Using the comando -h we can see the  Main Menu.

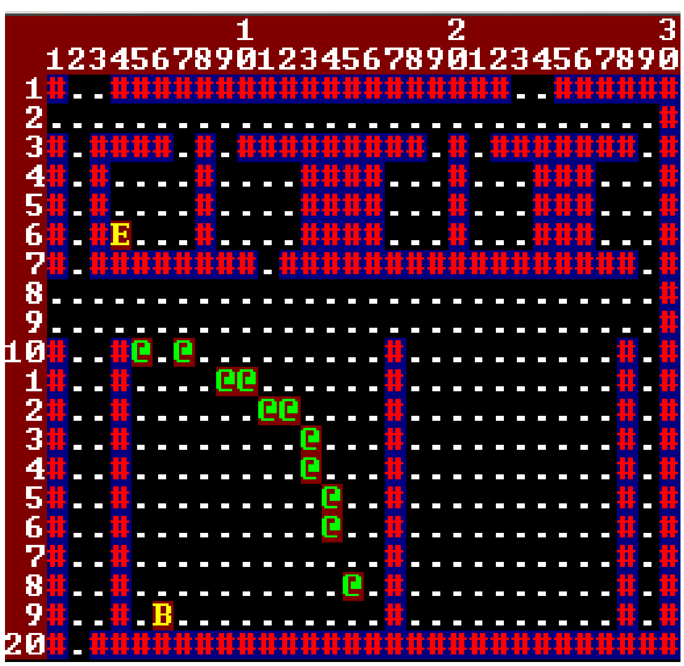
```
USAGE: main -f inputFile -h -s sx sy -e ex ey
-h       Print this help
-f       Select the specified input file. Default map1.txt
-s       Insert the starting point, it waits 2 numbers white spaced.
         first number is the row, second number is the column
         For example: -s 18 5
-e       Insert the ending point, it waits 2 numbers white spaced.
         first number is the row, second number is the column
         For example: -e 18 5
-p       Set priority queue as the system to handle the queue of nodes
-print   Print the map and exit

C:\Users\Sony\Desktop\Release>_
```

Using the comand –print we can print the map:

```
            1         2         3
  123456789012345678901234567890
 1B..###################..######
 2.............................#
 3#.####.#.#########.#.#######.#
 4#.#....#....####...#...###...#
 5#.#....#....####...#...###...#
 6#.#....#....####...#...###...#
 7#.########.#################.#
 8.............................#
 9.............................#
10#..#............#..........#.#
 1#..#............#..........#.#
 2#..#............#..........#.#
 3#..#............#..........#.#
 4#..#............#..........#.#
 5#..#............#..........#.#
 6#..#............#..........#.#
 7#..#............#..........#.#
 8#..#............#..........#.#
 9#..#............#..........#.#
20#.############################
```

We can choose by the command prompt the starting and ending point

And with the –a flag we can display how the algorithm evolve throught the map looking for the best part



# 6. Extensions:

I had implemented basically 4 extensions, (2 trivial and 2 not so trivial)

## 6.1. Visual method

This method allow the engine to detect which points are obviously open, for example, both points are not obstructed by obstacles or walls. In this cases the engine just need to construct the path between the two points without the overhead of the engine and return the path.

I am using a routine to generate a line, the best and most efficient function to find the points between two points.

## 6.2. Point of view methods to handle the set of adjacent points.

The literature offer several hints about how to implement the set of candidates closer to the current node. As I am implementing a map based on a grid, the set of candidates obviously are the adjacent points. I store this points in a vector and all is handled inside the map class. The method CMap::find take cares of detect and store which are the adjacent points.

## 6.3. LLQueue and IPriorityQueue Interfase

There are a lot of recommendations of how to represent the set of open nodes. For example:

Unsorted arrays or linked lists

Sorted arrays

Sorted linked lists

Sorted skip lists

Indexed arrays

Hash tables

Binary heaps

Splay trees

HOT queues

Data Structure Comparison

Hybrid representations

This list was taken from [Amit]

I was trying to implement the HOT queues because is suppose to be the best implementation. Unfortunately the literature is not clear how to implement this [Cherkassky and Goldberg]

Finally I choose to implement the linked list with a hash table as described in the section class LLQueue. Also, I implemented a interfase to be able to choose which implementation to use LLQueue or Priority Queue.

I did several test using both implementations and yes, in all our linked list plus hash table executed with better performance that the traditional Priority queue implemented using STL std::push_heap

## 6.4. Graphics interface

I have implemented a graphic interfase compatible with the Windows console using the Win32Api. Check the previous section to see several screenshots.

Also, I implemented several commands to handle each feature of the engine as described below:

| | |
|---|---|
| **-h** | Print the help |
| **-f file** | Select the specified input file. Default map1.txt |
| **-s row col** | Insert the starting point, it waits 2 numbers white spaced<br>For example: -s 18 5 |
| **-e row col** | Insert the ending point, it waits 2 numbers white spaced<br>For example: -e 18 5 |
| **-p** | Set priority queue as the system to handle the queue of nodes. Default is LLQueue |
| **-print** | Print the map and exit |

## 7. References:


- [Cherkassky and Goldberg] Boris V. Cherkassky , Andrew V. Goldberg Heap-on-Top Priority Queues
- (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.2448)
- [Amit] Amit game programming technnics
- http://theory.stanford.edu/~amitp/GameProgramming/
- [Wikipedia]  A* search algorithm
- http://en.wikipedia.org/wiki/A*_search_algorithm
- [Ravin] Steve Ravin A* Aesthetic Optimizations