EFFICIENT PATH FINDING FOR TILE-BASED 2D GAMES

Masilo Mapaila
University of Cape Town
Rondebosch
Cape Town, South Africa
mplmas002@myuct.ac.za

KEYWORDS

Path finding, A*, A Star, Path searching, Heuristics

ABSTRACT

In this paper we investigate different heuristics used in an A* (A Star) algorithm. This algorithm can be used to achieve efficient path finding within tile-based games. Path finding is a computationally expensive problem that is solved by searching. We investigate different optimisation techniques and further develop techniques which can be incorporated within the existing algorithm. These techniques make path finding for 2D static and dynamic environments faster with less use of memory.

INTRODUCTION

Often tile-based games have characters that are controlled by players. Whenever a player issues a command, it is intended that they behave intelligently in a manner consistent with their roles. This may include carrying a box, painting a wall, etc. But to do these tasks, they have to move from one place to another. This requires a realistic looking path between the two locations. As the number of characters increases, multiple simultaneously paths may be required. In general, human movement is an artificial intelligence (AI) or robotics problem for which there exists a general solution, Exhaustive Search, which is inefficient. Therefore, the aim of this study is to investigate the A* algorithm from the artificial intelligence which can be used for efficient path finding. Moreover, we will develop a tool to find an optimal solution to the path finding problem.

In modern tile-based games, most of the resources are used in the enhancement of graphics and physics and very few are available for AI. Therefore, it is assumed that very limited resources are available (with respect to memory and processing) for finding paths in real time. Further, we assume a 2D environment so that much of the work can be focused on the development of efficient algorithm rather than dealing with the complexities associated with the 3D environments.

The efficiency of path finding within an environment mainly depends upon the complexity of the environment. By complexity, we mean how big the environment is, whether it is static or dynamic and how many and how large the obstacles within the environment are. Further, these obstacles can also be static or dynamic. Therefore our study will focus on both static and dynamic environments that have a range of obstacle sizes. Moreover, the obstacles will sometimes be static and sometimes move around the environment.

BACKGROUND

Path finding is an AI robotics problem that cannot be solved without searching. The main problem in path finding is obstacle avoidance. One of the ways to approach this problem is by ignoring the obstacles until one encounters them (Stout, 1996). This is a simple step-taking algorithm that requires the unit’s current position and its destination position to evaluate a direction vector and information as to whether the units neighbouring region is clear or blocked. This algorithm finds the path along with the movement but the paths generated by this are unrealistic, computationally expensive and require lots of memory. Therefore it becomes necessary to have entire knowledge of path before the movement is applied. This is also necessary in the case where there are weighted regions and finding the cheapest path is important.

Various algorithms exist from conventional AI that can be used for path searching before its execution. These algorithms are presented in terms of changes in the state or traversal of nodes in a graph or a tree. Russell et al (1995) suggested these algorithms and broadly classified them into two classes. One class is uninformed search algorithms such as: Breadth First Search (BFS), Bidirectional BFS, Depth First Search (DFS), Iterative Deepening DFS, etc. These algorithms have no additional information beyond the problem definition and they keep on generating neighbouring states or nodes blindly unless they find the goal. These algorithms do not consider weighted regions, are computationally expensive, requires more memory and may not yield paths in real time. However, they are simple to implement.

The other class of algorithms uses problem specific knowledge or heuristics to find an efficient solution. These include algorithms such as Dijkstra's algorithm, Best First Search (BeFS) and

A-Star (A*). Both the Dijkstra's and the BeFS finds an efficient and optimal path when there are no obstacles within the environment. However, in an environment with obstacles, the former yields a shortest path but is computationally expensive, whereas the later works less but generates non-optimal paths. The A* on the other hand, combines the best of both algorithms and is guaranteed to yield the most efficient and shortest path. It is probably the best choice for path finding since it can be significantly faster, is flexible and can be used in wide range of contexts.

Typically, the A* algorithm traverses within an environment by creating nodes that correspond to various positions it explores. These nodes not only hold a location but also have three attributes associated with them, as suggested by Matthews (2002), which are as follows:

1. Goal Value (g): This represents cost to get from starting node to this node. This is the exact cost that depends on the environment.
2. Heuristic Value (h): This represents estimated cost from this node to the goal node.
3. Fitness Value (f): This is the sum of *g* and *h* values. This represents the best guess for the cost of this path going through this node. The lower the value of *f*, the better is the path.

The *g* value represents the path from start that is supposed to minimise any cost related factor such as distance travelled, time of traversal, fuel consumed, etc. Other factors can also be added such as penalties for passing through undesirable areas, bonuses for passing through desirable areas, aesthetic considerations such as making diagonal moves more expensive than orthogonal moves, etc.

On the other hand, the *h* value gives an estimate of cost to the goal. It is the most important factor in the efficiency of the A*. A bad heuristic can slow down A* and/or produce bad looking paths. Generally, a heuristic is an under-estimate of the actual cost to goal so that A* always generates shortest paths. However, under-estimating the heuristic too much is also not beneficial to A* as it will allow the A* to look for more and more better paths and would take longer time to return the path.

The A* extracts the node which has minimum *f* value and uses two lists, namely an Open and a Closed lists, for unexamined and examined nodes, respectively. These lists form the basis for the A* and their associated data structures determine how efficient the A* works.

The implementation of A* in tile-based games depends on:

- The nature of the game.
- The representation of the world.
- Information about the neighbours of each node.
- The cost functions (including heuristics).
- Speed and memory issues associated with path finding.

No matter how the world looks like, its background has to be quantized so that A* has available required search space. Stout (2000) suggested various ways to quantize the world such as Rectangular Grids, Quad Trees, Convex Polygons, Navigation Meshes, etc. Most of these representations require a great deal of interaction with artists and modellers of the world. For 2D environments, rectangular grids offer an easy way of representing the search space by partitioning into a regular grid of squares. This also allows efficient access of neighbouring nodes to speed up searching. For a typical node at location (x, y), a neighbour can simply be generated at location (x+1, y), (x, y+1), (x+1, y+1), (x-1, y), etc. For other techniques, a lookup table is created consisting of information about the neighbours for fast access of the neighbour's locations.

So far it has been discussed that heuristics form a major part in the workings of A* but what kind of heuristic to be used is another issue. The types of heuristics used mostly depend on the search space representations, on speed and accuracy issues associated with the path finding. Patel (2001) suggests some heuristics such as Manhattan Distance, Diagonal Distance (Delta-Max) and Straight Line (Euclidean) Distance as possible heuristic choices that can be used and tweaked on rectangular grids to the needs of one's game.

Although A* is the best search algorithm, it should be used wisely within a game as it may lead to wasting of resources. This is typically the case when there are large environments within a game that leads to the generation of hundreds and thousands of nodes in the Open and Closed lists. This not only requires excessive amounts of memory but also requires too much processing time, which a game cannot afford. Apart from that, there could be a situation when no possible path exists, causing the A* to be inefficient as it examines every possible location from the start before determining that it is impossible to get to the goal. Moreover, paths generated by A*, although shortest, may not be aesthetically acceptable and would possibly need to be straightened up, even making them smoother and direct. Thus to overcome the weaknesses of A* and to have the optimal use of resources, it requires optimisations on the A* and the path finding. These are discussed in detail as they are dealt with in this study.

THE PATHFINDER TOOL

The Initial Framework

We base our initial design on the Model-View-Controller (MVC) design pattern. The MVC has been proven to be most powerful architecture for GUI. It separates the modelling of the domain, the presentation, and the actions based on user input into three classes (Burbeck, 1992). The figure 1 below represents the relationship between these three classes namely the model, the view and the controller.

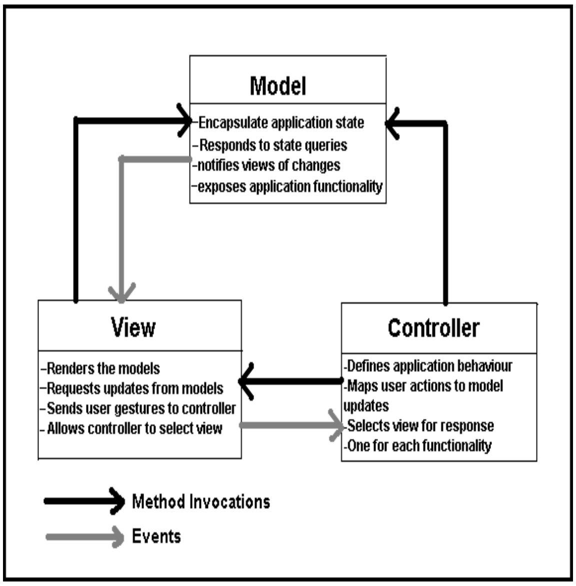

Figure 1: Model-View-Controller pattern

We decide to use MVC pattern as it will allow adding new functionality in the future without making any drastic changes. These additions may include creation of multiple views and controllers and maintaining synchronisation of the views whenever a model changes, addition of models of different types with separate views and/ or controllers for these models, porting of existing work to another platform, etc.

The A* Algorithm

The A* forms the core of the study. It needs information regarding memory (storage), environment (search space) and start and end locations. As discussed, we partitioned the space into a rectangular grid with the same height and width as the size of the environment. This partitioning is carried out in two levels of inheritance as suggested by Higgins (2002). This has two significant advantages. Firstly, a generic path finding engine can be built to support different environments with the same basic functionality. Secondly, this technique emphasises the use of templates instead of base classes and virtual functions, which significantly reduces the assembly overhead associated with the virtual functions. Further, A* requires some information from the grid that determines whether a particular grid square is passable or obstructed. In addition, it needs information as to whether a particular node is in the Open or Closed list. This information needs to be passed to A* as quickly as possible and at the same time it should be stored efficiently. We approached this by using an unsigned char data structure that stores these different states as status flags. Figure 2 shows C/ C++ representation of the status flags.

```
typedef unsigned char ASFlags;
enum
  {
    asfClear        = 0x00,
    asfPassable     = 0x01,
    asfBlocked      = 0x02,
    asfInOpen       = 0x04,
    asfInClosed     = 0x08,
    asfObstructed   = 0x16,
  };
```

Figure 2: A* states as status flags

By using a single variable, it requires 1 byte per A* node to store its state information which can be retrieved by simple array as a lookup. The size of the array is made to the maximum size of the search space and storage and retrieval of information is made efficient by use of bitwise operators. With this, a node can be in more than one state at one time. This not only reduces memory requirements but also allows path-finding data to be made independent of the search space. This allows path finding for multiple characters to be done simultaneously.

The A* uses this node information in order to keep track of nodes presence in either an Open or Closed list. For this, efficient data structures are needed for both the lists. With the above status flags, no additional data structure is used for the Closed list as its functionality is achieved by simply updating the status flags. However, the main task of A* lies in the working of the Open list. Typically, Open list operations include extraction from a sorted list, insertion into a sorted list, updating the cost of a node in the list and resorting the list, and determining whether it is empty or not. Patel

(2001) suggested different data structures that can be used for the Open list and recommended the use of priority queues as the most efficient data structure. Although, priority queues can be implemented by standard template library (STL) as suggested by Nelson (1996), its STL implementation is limited and does not perform all Open list operations. Instead, we implement priority queues as binary heaps and used STL heap operations on STL vector containers. A binary heap is a sorted tree in which a parent always has a value lower than its children. However, there is no ordering among the siblings and so it is not a completely ordered tree but is sufficient for A* to perform the insertions and extractions in only O(log n) (Lester, 2003). Figure 3 and 4 shows a typical case of a binary heap in a tree and array (STL vector) representation, respectively.

10-30-20-34-38-30-24

Figure 3: Binary heaps tree representation

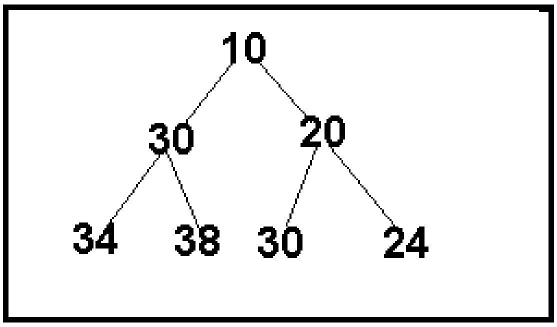

Figure 4: Array representation of binary heap of figure 3

Memory Management

A* requires memory for extraction of nodes and so it is important to have a memory manager which provides an efficient way of dynamic memory allocation for A* nodes. We implement this by using the buffering technique (Figure 5). In buffering, a piece of memory is kept aside by the system to be used for dedicated task. Here we reserve this for the storage of A* nodes.

For A*, it is a good way to manage nodes because A* requires lot of nodes to progress its search. Initially, when a request is made, a piece of memory is dedicated before A* starts execution. During the course of execution, if all the memory gets exhausted, a new buffer is created to progress. The size of this buffer is allowed to change so that less memory is wasted. This size mainly depends on the complexity of the environment and therefore requires tuning before it is used in an application.

This design has significant advantages even though sometimes extra memory is allocated, which increases the memory requirement. Firstly, this results in better use of memory with respect to fragmentation. If smaller nodes are created and deleted on the fly, it leads to fragments in the memory that would make this piece of memory unsuitable for other purposes. Secondly, creation and deletion of new nodes at run time requires the same time as creating one large chunk of memory. If smaller nodes were created at run time then this would affect the performance.

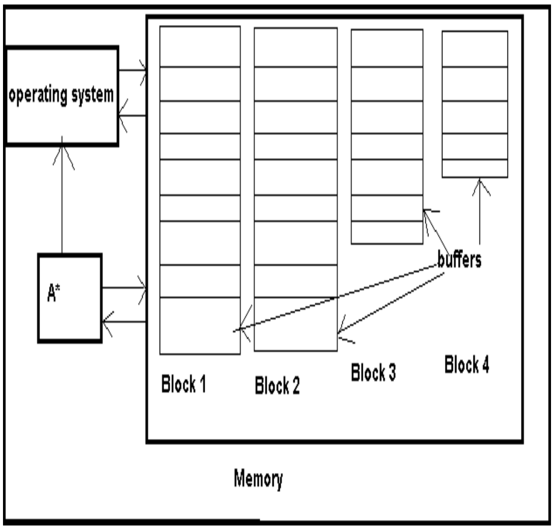

Figure 5: Buffering for memory management

Costs and Heuristics

This forms the main part of research within this study. A* requires two cost functions to proceed its search. These are the actual cost (g) and the heuristic cost (h), which depends on the environment and search space representation.

For rectangular grids, we assume movement in all possible directions and therefore each A* node has a maximum of eight neighbours (four diagonal and four orthogonal) (figure 6). For A* to generate straight paths, a penalty is added for a movement towards the diagonal neighbour as shown in figure 6. However, this cost is scaled by a factor of 10 in order to avoid any floating point calculations to speed up searching within the A*.

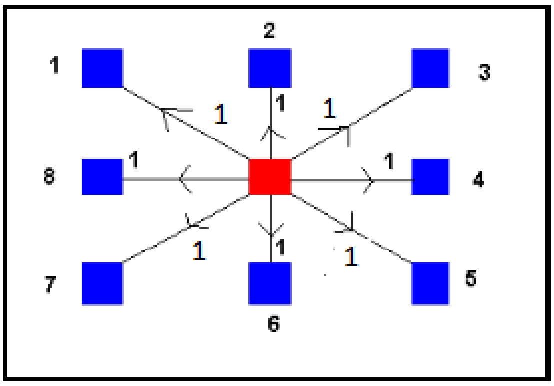

Figure 6: A* node with its neighbours and their respective costs of movement.

Initially the Manhattan Distance heuristic is used as it is supposedly the best underestimating heuristic for rectangular grids (Patel, 2001). The underestimated Manhattan distance simply adds the absolute values of the difference of their respective X and Y coordinates (figure 7). This is further scaled by factor of 10 in order to avoid floating point calculations and to make it consistent with the scale of actual cost.

H = (abs(X current – X goal) + abs(Y current – Y goal)) * Factor

Figure 7: Manhattan distance heuristic

The Manhattan distance heuristic generates optimal paths in real time. However, this is true in the case where there are no or very few static obstacles. As the size and the number of obstacles increases, A* not only spends more time on searching but also requires more memory for the nodes as it needs more nodes to find a path. Thus in order to reduce the time and memory requirements when finding paths with obstacles, Rabin (2000) suggested an overestimation in heuristics. Such overestimation works such that the sub optimal realistic looking paths are generated with a speed consistent with a regular Manhattan distance heuristic function with no obstacles. This requires combining of an underestimated Manhattan distance heuristic along with an overestimated heuristic. However overestimation is a research issue and no general solution exists at present. We approached this problem by using ideas from Patel (2001) diagonal movement cost (Delta-Max) along with lot of experimentation and have come up with an overestimate as shown in figure 8.

Overestimate = max (abs(X current – X goal), abs(Y current – Y goal)) * 15

Figure 8: Overestimate heuristic cost.

The value of 15 as a scale factor is determined by constantly tuning the heuristic on a series of data sets. Initially A* algorithm runs on the Manhattan distance heuristic till it encounters an obstacle and then it runs on the overestimated heuristic. This not only has significant performance improvement both in terms of memory and the speed of path finding but also results in realistic and optimal looking paths as generated with Manhattan distance heuristic only.

Another heuristic is the Euclidean distance which is calculated to be the straight-line distance from the start node to the target node. A sample test of this is shown in the following section.

A Sample Test

We checked the developed heuristic on a predefined set of start and end locations in an environment which has large static obstacles. The following figures (9, 10 and 11) show and compare the type of path generated by using different heuristic functions for same start and end location.

Clearly from figures 9 and 10, the paths generated are the same, but this is not always the case. The Euclidean distance heuristic requires A* to search more nodes in order to generate the shortest path. This is evident from figure 9 which shows the nodes searched in different colour from the original grid colour.

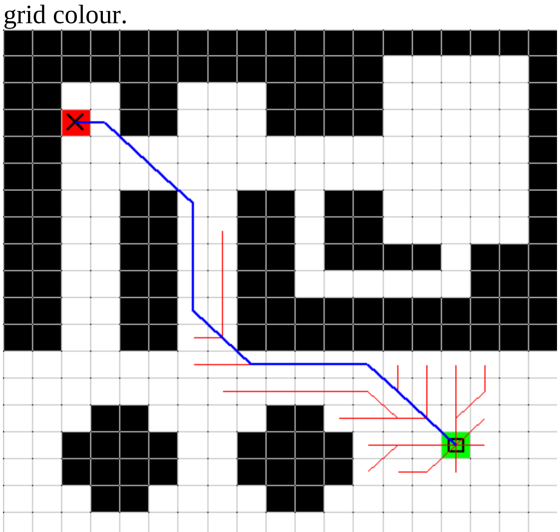
Figure 9: Path finding example using Euclidean distance heuristic. (Optimal path)

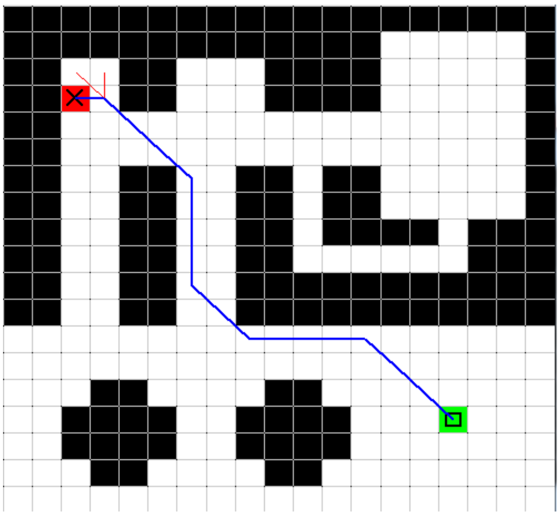

Figure 10: Path finding example using Manhattan distance heuristic. (Optimal path)

On the other hand, the Manhattan heuristic generates the same path as figure 9 while making A* search fewer nodes.

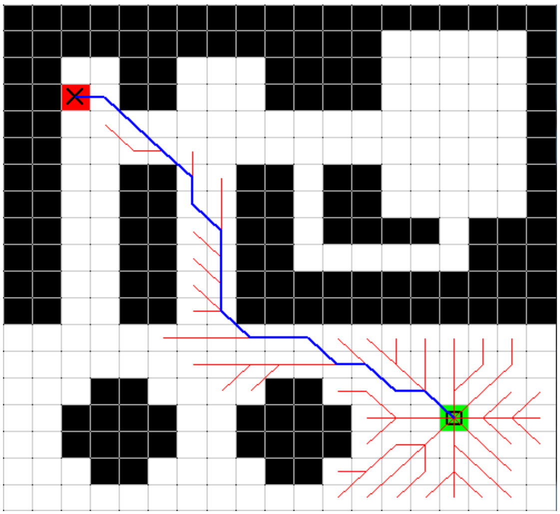

Figure 11: Path finding example using Delta-Max distance heuristic. (Optimal path)

The Delta-Max generates a different path from figure 9 and 10 while making A* search more nodes.

CONCLUSION AND FUTURE WORK

In this paper we presented a build up to an efficient tool for path finding for 2D environments. At present, this tool has limitations and we see the work presented here as a step towards the development of a complete tool. The current work focussed on static and dynamic environments and we have worked with both fixed and dynamic cost environments.

We incorporated a number of optimisations while developing the A* algorithm. In future, we will extend this to post processing techniques. These are mainly application specific and therefore utility libraries would be developed so that they can be used depending upon the application.
In summary, much remains to be done in the field of path finding in games. Most of the research in academic AI has been focused on robotics and very little has been done towards their application in tile-based games. This study bridges that gap and with further research, it would be possible to develop a complete tool that would be useful in academia and would certainly benefit the game industry.

REFERENCES

Beman Dawes, David Abrahams, 1998-2005. Available: http:www.boost.org

Burbeck S, 1992. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC) [online], University of Illinois at Urbana Champaign, Available: http:// st www.cs.illinois.edu/users/smarch/st-docs/mvc.html

Higgins D F, 2002. How to Achieve Lightning-Fast in Rabin S, AI Game Programming Wisdom, Hingham, Massachusetts: Charles River Media, pp. 133-144

Lester P, 2003. Using Binary Heaps in A* Pathfinding [online], Available: http://www.policyalmanac.org/games/binaryHeaps.htm

Matthews J, 2002. Basic A* Pathfinding Made Simple, in Rabin S, AI Game Programming Wisdom, Hingham, Massachusetts: Charles River Media, pp. 105-113

Nelson M, 1996. Priority Queues and the STL [online], Dr. Dobb's Journal. Available: http://marknelson.us/1996/01/01/priority-queues/

Patel A J, 2001. Amit's Game Programming Information [online], Stanford University. Available: http://www-cs-students.stanford.edu/~amitp/gameprog.html

Rabin S, 2000. A* Speed Optimizations, in Deloura M, Game Programming Gems, Rockland, Massachusetts: Charles River Media, pp. 272-287

Russell S and Norvig P, 1995. Artificial Intelligence: A Modern Approach, 2nd edition, Upper Saddle River, N.J.: Prentice Hall

Stout B W, 1996. Smart Moves: Intelligent Path-Finding [online], Game Developer (October), pp. 28-35
Available:http://www.gamasutra.com/view/feature/3317/smart_move_intelligent_.php

Stout B W, 2000. The Basics of A* for Path Planning, in Deloura M, Game Programming Gems, Rockland, Massachusetts: Charles River Media, pp 254-262