



UNIVERSITETET I AGDER

Pathfinding

By

Christian Auby and Tung Doan

Supervisor: Ole-Christoffer Granmo

Project Report for Webmining and data-analysis in autumn 2007

University of Agder

Faculty of Engineering and Science

Grimstad, 30th of November 2007

Status: Final

Keywords: Pathfinding, RTS, C++, A*, AI

Abstract:

The goal is to implement path finding in C++. Path finding is the task of calculating each point that leads from position A to position B, by the cheapest route defined by the implementation and the terrain.

Our main priority is the algorithm itself, and what factors affect performance. We will also discuss what methods can be used to optimize the function using methods not directly related to the actual search implementation.

Table of Contents

1	Table of figures	4
2	Introduction	5
3	Background	6
4	Problem Description	7
5	Requirements.....	8
6	Design Specifications	9
7	Test Cases.....	12
7.1	Legend.....	12
7.2	Original Problem	13
7.2.1	Solution	13
7.3	Backtrack Problem	14
7.3.1	Solution	14
7.4	Complex map	15
8	Computation speed.....	16
8.1	Map size	16
8.2	Complexity	16
8.3	External factors	16
9	Memory usage	17
10	Implementation	18
10.1	Language	18
10.2	Libraries.....	18
10.2.1	STL vector	18
10.2.2	SDL.....	18
10.3	Usage.....	18
11	Discussion.....	20
11.1	ORTS.....	20
11.2	Fog of war	20
11.3	Cost zones	20
11.4	Optimization	20
11.4.1	Zones.....	20

11.4.2	Stored paths.....	21
11.4.3	Data structure	21
12	Conclusion.....	22
13	References	23

1 Table of figures

Figure 1: Original problem from the tutorial	13
Figure 2: The shortest path from point A to point B is highlighted in blue	13
Figure 4: Backtrack solution	14
Figure 3: Backtrack problem	14
Figure 5: Complex map	15

2 Introduction

We chose this project as it interesting for both of us. We have been gamers since the 8-bit era, and are very passionate about games. When we saw that some of the projects were about game logic we didn't hesitate to pick one of them. We chose pathfinding specifically because of the visual demos that are possible for such a project.

The main purpose of our project is to develop an implementation of the A*(A-star) algorithm. [1] The interface should be easy to customize and use in other applications. One example of such an application is the ORTS-engine (Open Real-Time Strategy). [2]

3 Background

Pathfinding is a term used to plot the best route from point A to point B. Pathfinding is used in a variety of games, especially Real Time Strategy games. It is referred to calculate a path around an obstacle or terrain.

In 1995 Westwood Studios released Command & Conquer to universal acclaim. [4] The game's success helped to popularize and define the RTS genre as a whole, and the franchise has been a commercial success ever since. [5]

During those days the AI was fairly simple. The AI always chose the shortest route from point A to point B ignoring any enemy defenses, tiberium (infantry took damage if they walked on tiberium), and they wouldn't destroy sandbags as long as there's a way around it (you could guide the enemy straight to your base defenses).

The latest game in the Command & Conquer franchise Command & Conquer: Tiberium Wars had a very versatile AI. They would try to flank you, avoid your defenses, and attack your weak spot.

4 Problem Description

The original project description is as follows: “Develop a pathfinder for ORTS. The idea for this algorithm is to find the shortest path to an enemy unit and attack this unit. In this assignment obstacles, such as stationary units (mountains etc.) must be considered.”

After talking to our supervisor we decided to concentrate on making a modular implementation that is not tied to ORTS. For this reason our solution is not integrated in ORTS. This also helps differentiate the two groups who have chosen the same task.

5 Requirements

First of all the pathfinder should be able to find the shortest way first on a simple map.

Additionally some units move faster on some terrain than other, and the pathfinder will be able to find the fastest route for the specific unit. For example a jeep would be fast over flat landscapes, but would be considerably slower in woods, while infantry wouldn't lose any speed in woods. The pathfinder should be able to distinguish between these, and jeeps would be given a route that goes around the woods, while infantry would be given a route straight through.

If time allows it we would like to give the AI several options to choose from; such as "shortest way possible", and "avoid enemies".

6 Design Specifications

The algorithm we have chosen is called A*. A* uses a logic approach to find the goal, and by design reaches the best solution first. This fits in very well with the requirements for a game, where speed and good results are important.

We have chosen to implement the algorithm as a class. The class structure and its most important members are as follows:

Class cAstar public member functions:

- void Init(cPoint* start, cPoint* end)
- bool goNext()
- void checkBorders()
- void cleanup()

Public member variables:

- bool canCutCorners

The class also has several protected members it uses internally. Two of these are of great importance:

- virtual int getCost(int unit, int terrain)
- virtual bool isWalkable(cPoint* pos)

GetCost must be written specifically for the application the algorithm is used, and performs the task of weighing different types of terrain for different types of units differently. In other words it performs most of the functionality set by our requirements. As this function is virtual, it can be overloaded by a derived class. This is the recommended way of implementing custom movement cost.

isWalkable depends on the level layout, and must also take friendly units, buildings in addition to enemy units and buildings. This function is also game specific, and our implementation can be overloaded in a derived class just like GetCost. isWalkable can also be used to find a path that avoids enemies, by returning false if an enemy unit or building is visible in the vicinity of the point that is being checked.

In addition to those from our original specification we also added a helper function to our class, cPoint** cAstar::getPath(). This is a function that returns the full path as an array of cPoint object pointers, zero terminated. We added this function because it would greatly cut down the time it would take to write our test cases. If the algorithm does not find any path it returns null.

Although very practical this function gives little control to when the search is being done, requiring all to be calculated at once. If it is to be used in a live game starting it in a separate thread would be preferable, as once it's started it works until it is finished.

cPoint is a simple class that implements a point:

class cPoint

- cPoint* parent
- int x, y
- int cost
- int distance

Distance is the distance to the goal, calculated any way we find fit.

Cost is the amount of time needed to walk, or rather, an arbitrary number who's sum should be as small as possible. In our task the cost is related to movement; moving from one square to another, diagonally and so on. Different terrain will intice different cost, thus taking into account sand, grass, hills and other terrain types.

Parent is an important member, as it is what makes the cPoint a linked class. Every point points to it's parent point, except for the head point which points to null. By following these pointers the full path can be traversed when the algorithm has found the end point.

7 Test Cases

This chapter will present various test cases and other points of interest that we have researched during our development and test phase. For the most part we got the expected results, but there were some surprises.

The initial problem presented by the tutorial we used is as follows: find the most optimal path from point A to point B on a matrix. The matrix contained obstacles that a unit would have to work around. These experiments are for showing search paths, and the path the unit chooses on different maps.

7.1 Legend

Green = Sweep path

Purple = Point A and Point B

Red = Obstacles

Black = Walkable

Blue = Chosen Path

7.3 Backtrack Problem

In this scenario point A is surrounded by obstacles, the path first has to move further away from point B and move around the obstacles.

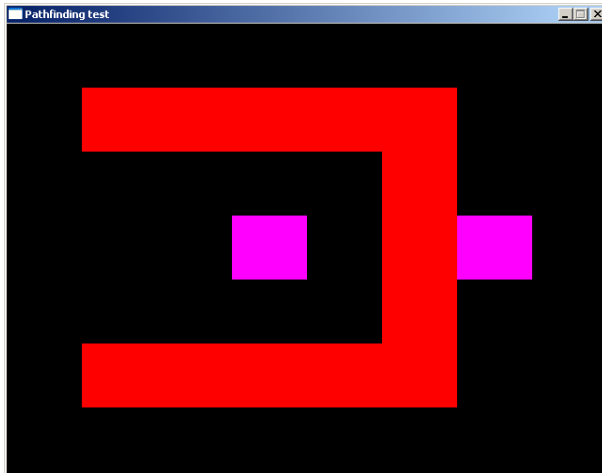


Figure 3: Backtrack problem

7.3.1 Solution

Here A* algorithm would first try the shortest way, when it hits the obstacle it will search the area behind until it finds an opening. Then it will search until it finds the point B. As the algorithm tries the shortest way first the search area is bigger than the initial problem.

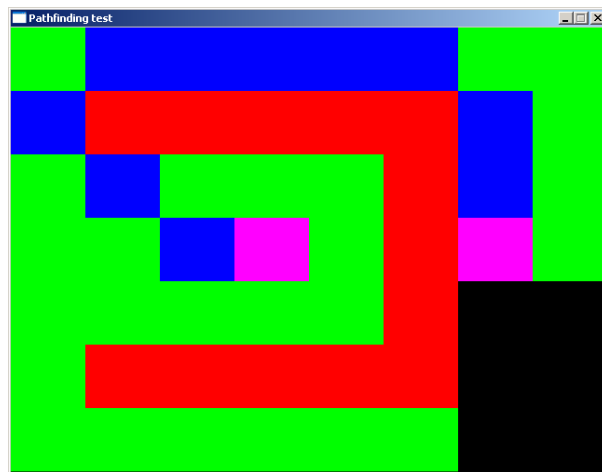


Figure 4: Backtrack solution

There are two paths to point B which is exactly the same distance. In this case A* would just pick the path it finds first.

7.4 Complex map

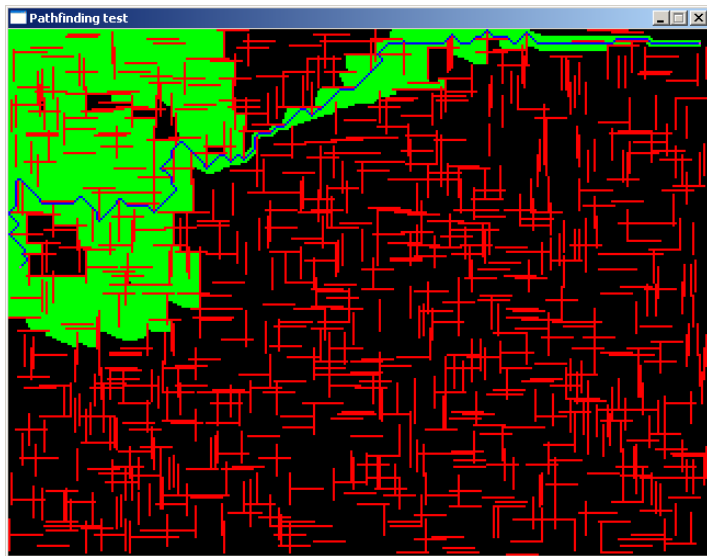


Figure 5: Complex map

In this case a lot of the area had to be swept to find the most optimal path. This map combines several of the factors that impacts calculation speed: large map, many obstacles and a path that often changes direction. This path takes approximately 5 seconds to calculate. Note that there is possible to construct a problem that is much more complex than this, but these will most likely not be seen in a real scenario.

8 Computation speed

Various factors play a vital role in how fast the A* algorithm can calculate the path.

8.1 Map size

The size of the matrix greatly impacts the time it takes to calculate the path. The initial problems presented are very small in size and is calculated almost instantaneously. On a 7x5 matrix where the path from point A to point B is a straight path, it takes 1 millisecond to calculate. On a 160x120 matrix it takes 50 milliseconds to calculate a straight path. Thus the time for a straight path scales almost linearly to the distance from Point A to B.

8.2 Complexity

Obstacle complexity impacts the calculation speed more dramatically and is hard to predict. Optimizing code for this is outside our assignment as it might require specialized cases.

8.3 External factors

Location of the two points is highly vital for calculation speed. If the two points are close together size the map size hardly matters.

The A* algorithm takes heavy use of list operations. In our implementation we decided to use the standard C++ list library. This library is very complete and contains everything needed for a list class, but not might be optimal for our use. Accustomed link list implementation specifically written for our task would most likely perform much better.

Implementation of this class is outside our assignment, but should be considered in real uses of the algorithm.

9 Memory usage

Total memory usage is dependent on the same factors as speed, but in all cases the memory requirements are minimal compared to computational requirements. Memory usage range from about 1 MByte up to 6 MByte for a 50 * 50 map, and this includes the full application as well as the path data. Any other considerations regarding memory are not necessary at this point.

In a live application with many units memory usage will grow linearly with the amount of units. Situations like this will also quickly run out of CPU resources, and could be solved by grouping units that are close together, and only calculate their common path once.

10 Implementation

10.1 Language

ORTS is written in C++, and as the task originally tied to ORTS choosing C++ was easy. We chose to write the algorithm independent of ORTS, and although that gave us larger language freedom, we chose C++ anyway. The algorithm is CPU intense, and would perform much worse in a scripted environment. C++ also supplies containers fit for our task in the STL library.

10.2 Libraries

10.2.1 STL vector

STL vector is a dynamic array that can automatically resize itself when inserting or erasing an object. Inserting and erasing in the back of the vector takes a constant amount of time. Inserting or deleting in the middle of the vector is linear in time. Vector is one of the most used containers and easy to use. The interface is similar to a C array.

Vector is used internally in our cAstar class for keeping the open and closed lists.

10.2.2 SDL

Simple DirectMedia Layer is a software library in C that delivers platform independent functionalities for graphics, sound and APIs. SDL enables a programmer to write multimedia applications and run it on most operating systems.

SDL is only used in the demo application, for visually displaying the algorithm results. Up and down can be used to step through each point, page up and page down can be used to show and hide all steps.

10.3 Usage

Using the public functions from our design specification finding a path works like this:

```
// Init positions
star.init(start, end);

// Search until found.
while(true)
{
    // Find the next optimal position and go there
    if(!star.goNext())
    {
        cout << "No path found!" << endl << endl;
        system("pause");
        exit(0);
    }

    // Are we at the goal? if so, stop searching
    if(star.found)
    {
        end->parent = star.pos->parent;
        break;
    }

    // Check squares next to this and add them to the open list
```

```
        star.checkBorders();  
    }
```

Alternately, using the new helper function the same can be achieved by replacing the loop with:

```
cPoint** solution = star.getPath();
```

In an actual game the first method would be preferred, as it gives greater control. It is also more adoptable to a threaded approach, by threading the loop and give the thread a lower priority than the user interface and other parts that need to be responsive.

11 Discussion

11.1 ORTS

From the beginning we were asked to implement our project in ORTS. However we felt that we should concentrate on the algorithm and make it as flexible as possible instead of spending time on getting it to work in ORTS. Our experience with ORTS showed that the path usually is quite linear and doesn't seem challenging, but we would probably have to spend a significant amount of time on understanding how ORTS works.

11.2 Fog of war

When we first started with this project we wanted to implement fog of war, but this turned out to be unnecessary. Fog of war wouldn't affect how the algorithm solves the problem. In most RTS-games the AI knows the layout of the map, only hiding enemy units and buildings inside fog of war. Depending on the game's specific implementation of `isWalkable()`, this could be done by returning true for terrain that has enemy buildings on it as long as the area is unexplored. As the unit is moving, it won't know that there is a building in the way until it clears fog of war, at which point it will have to reassess the path it is on.

11.3 Cost zones

The levels we made prioritized a unified test environment, and as such all squares have the same cost. However, the actual zone cost is retrieved by the `GetCost()` function, and by customizing this any number of different cost scenarios can be achieved. Any class that inherits `cAStar` can overload this function to suit specific needs.

11.4 Optimization

During the performance analysis of our implementation it became clear that other methods for optimizations would be preferable in a live game. As the map size increases the need for a way to simplify the path finding becomes apparent.

One option for preventing the game to stall during calculation is to split the calculations into separate threads, which does not actually make the calculation time shorter but rather makes the game run smoothly even though it is doing heavy work in the background.

11.4.1 Zones

Zones are larger areas of the map containing smaller areas (the actual tiles), being treated as tiles. This can reduce the search complexity immensely, effectively decreasing the map size as the virtual map size can be many times smaller than the actual map size in tiles.

These zones would preferably be made manually, as they depend on the layout of the map. One way to do this is building tools for this into the map editor used for the game. This way the level designer will be able to generate custom zones that act as one tile for the path finding algorithm, finding a rough estimate for the proper path. Once two zones have been chosen, a more accurate path between these two can be calculated, saving the algorithm from having to calculate the full path at once.

11.4.2 Stored paths

Another method would be to pre-calculate paths, or partly pre-calculate them. This ties into the zone method, by having the path from one zone to another loaded from disk or calculated during level loading, to reduce the complexity even further. For paths to look more natural it could be wise for the level designer to plot parts of these paths manually.

11.4.3 Data structure

As mentioned earlier the data structure used to store all the points heavily affect performance. If performance were the main priority more thorough research into the available containers would be preferable. For us it was a matter of simplicity; having worked with the STL implementation of vector before, we felt confident we'd be able to implement the algorithm in time.

Some data structures are self sorting, a feature that might have been useful for us. Vector does not have such a function, and for us to find the best current point a linear search is required. It is unclear how big an impact such a structure would have on our implementation though, as other parts of the algorithm require linear search in the underlying container even if it is auto sorting.

We did some basic testing with the priority queue container, and results were promising but inconclusive, giving much faster search for the cheapest coordinate (one lookup), but not affecting the overall performance as much.

12 Conclusion

Pathfinding is a complex problem in real time systems, requiring heavy work that is tasking to the CPU. The A* algorithm is one of the more popular algorithms to solve such problems, and we feel our implementation of the algorithm gives the results we expected during the planning phase.

The interface we give the user is easy to use, and the return values easy to understand. The test cases we have run throughout development have proven that our implementation is able to solve the tasks we have given it.

Future work could be concentrated on implementing the optimizations in our discussion chapter. In addition, even though we chose not to integrate into OTRS this could also be achieved at a later time now that our core implementation is done.

13 References

- [1] <http://en.wikipedia.org/wiki/A%2A> – last visited 2007.11.30
- [2] <http://www.cs.ualberta.ca/~mburo/orts/> – last visited 2007.11.30
- [3] <http://www.policyalmanac.org/games/aStarTutorial.htm> - last visited 2007.11.30
- [4] <http://www.gamestats.com/objects/003/003220/> - last visited 2007.11.30
- [5] <http://www.gamespot.com/pc/strategy/commandconquergenerals/news.html?sid=2911739> – last visited 2007.11.30