An Optimised Implementation of the A/Algorithm using the STL

2 authors:

Bryan Duggan
Technological University Dublin - City Campus
25 PUBLICATIONS 79 CITATIONS
SEE PROFILE

Fredrick Mtenzi
Technological University Dublin - City Campus
87 PUBLICATIONS 493 CITATIONS
SEE PROFILE

An Optimised Implementation of the A* Algorithm using the STL

Bryan Duggan, Fred Mtenzi
School of Computing,
Dublin Institute of Technology,
Dublin 8, Ireland.
{bryan.duggan,fred.mtenzi}@comp.dit.ie

Abstract

In this paper we present our work on developing a fast, memory efficient and student friendly implementation of the A* algorithm for use in teaching using the game Dalek World. We first present our criteria for evaluating an implementation of the A* namely that it should be easy for students to understand, support multiple heuristics and demonstrate some of the optimisation issues which are relevant to implementing path finding. We compare three candidate A* implementations using various data structures to hold the open and closed lists. We conclude that our own implementation based on the STL `map` and `priority_queue` data structures is both the easiest to learn from and also the fastest.

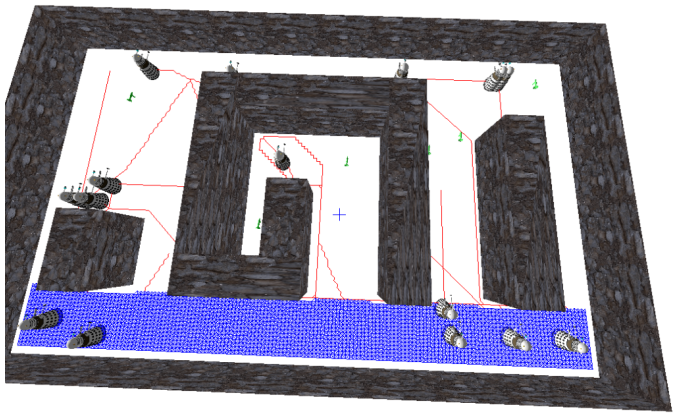

Figure 1: Dalek world with paths and part of the navigation graph drawn

1 Introduction

Our previous work [3] describes Dalek World, a framework we developed to teach a course in games programming at the School of Computing - Dublin Institute of Technology. In Dalek World, daleks need to navigate to ammunition spawn points, when they run out of ammunition. Figure 1 shows a top down view of Dalek World with shortest paths between daleks and ammunition drawn.

Path finding in the majority of commercial computer games is achieved using the A* algorithm [4]. In the second semester of our course, students learn how the A* algorithm is implemented in the game FarCry. Because the A* algorithm is so widely used in path planning, it is essential for students studying games programming to have an understanding of not only path finding [6], but also of the issues involved in creating a memory and speed optimised implementation. On our degree program, students learn the concepts of the A* algorithm in the third year AI Algorithms course. In fourth year, students learn how to implement the A* algorithm using C++ in Dalek World. This paper describes our work creating a "student friendly" (and optimised) implementation of the A* algorithm for Dalek World. Our criteria for implementing the A* algorithm in Dalek World were:

1. The implementation must be easy to understand. To support this, the C++ code should closely reflect the pseudo code and require as little "'extra"' knowledge as possible.
2. The implementation must support plug-in heuristics. This is so students can evaluate the algorithm for speed and efficiency using different heuristics, such as the Euclidean distance, the Manhattan distance and the distance squared.
3. It must support Dalek World sized navigation graphs. The world size in Dalek World is configurable, but a world size of 160 x 100 is used on the course. This results in a search space of 16000 nodes. Partitioning, triangulation and other techniques are not used at this point.
4. It must be fast and memory efficient as it will be called often and must not cause the program to "hang" as searches are carried out. As a benchmark, our goal was to make the implementation as fast as our adaptation of Buckland's implementation [2].

Our first attempt was to adapt Buckland's implementation from "Learning Game AI by Example" [2]. This implementation is both fast and flexible, however by the authors own admission, the implementation is difficult to follow. We therefore implemented our own version of the A* using various data structures from the STL to hold the open and closed lists. Our implementation is fast and memory efficient and in fact in most cases, it outperforms our adaptation of Buckland's implementation over the same search space and using the same heuristics. It also much easier to understand.

The rest of the paper is organised as follows. Section 2 presents a brief introduction to the A* algorithm. Section 3 describes our adaptation of Buckland's implementation and critiques it from the perspective of the criteria outlined above. Section 4 presents our implementation. We also present our performance data and compare our implementation with our adaptation of Buckland's. Section 5 presents our conclusions.

2 The A* Algorithm

The A* algorithm is a graph search algorithm. Pseudocode for the A* algorithm is presented in Figure 2[1]. First the search space (the world in this case) must be represented as a graph of navigable locations (nodes). Each *node* in the graph is connected to other nodes via an *edge*. An edge represents whether it is possible to travel directly from one node to another. Figure 1 shows a partial navigation graph generated using a flood fill algorithm in Dalek World.

The A* algorithm keeps two lists of nodes. An open list contains all the nodes found that have yet to be expanded and a closed list contains all nodes that the algorithm no longer needs to consider. The algorithm works by first adding a node representing the start position to the open list. The algorithm proceeds into a loop, popping off the node with the lowest f score from the open list until either the destination position node is popped or the the open list is empty. Each node that is popped (that is not the destination node) is expanded. When a node is expanded, each adjacent node is first checked to see if it is navigable. If it is navigable, the adjacent node is then checked to see if it is on the closed list. If its is not on the closed list, it is checked to see if it is on the open list. If it is on the open list, the f, g and h scores are calculated. The h score is the heuristic distance from the current node to the destination node. The g score is the cost from the source node to the current node and the f score is the sum of these two. If the node on the open list has a higher f score, it's parent and scores are updated. [2] refers to this as *edge relaxation*. Otherwise the nodes scores are calculated and it is added to the open list. Once the destination is found, the loop terminates and the path is generated by iterating from the destination to the start node via parent nodes.

[1] Adapted from [5], [2], [7]

```
function findPath(Node start, Node end) {
  List open, closed;
  open.empty();
  closed.empty();
  start.f = start.g = start.h = 0;
  open.push(start);
  while(! open.isempty()) {
    bestNode = open.popLowestFScoreNode();
    if (bestNode == end) {
        \\Iterate from end to start via parents
        \\Construct the path;
    }
    closed.push(bestNode);

    parent = bestNode;
    foreach adjacentNode in bestNode.adjacentNodes() {

      if (adjacentNode.isTraversable()) {
        if (! closed.exists(adjacentNode)) {
          if (! open.exists(adjacentNode)) {
            adjacentNode.h = heuristicCostTo(end);
            adjacentNode.g = parent.g + costTo(adjacentNode);
            adjacentNode.f = adjacentNode.g + adjacentNode.h;
            adjacentNode.parent = parent;
            open.push(adjacentNode);
          } else {
            // Edge relaxation
            openNode = open.find(adjacentNode);
            adjacentNode.g = parent.g + costTo(adjacentNode);
            if (adjacentNode.g < openNode.g) {
              openNode.g = adjacentNode.g;
              openNode.parent = parent
            }
          }
        }
    }
}
```

Figure 2: A* Pseudocode

3 Buckland's A*

Our first attempt at implementing the A* in Dalek World was to adapt Buckland's A* implementation from "Programming Game AI by Example" [2]. In his book, the author first describes Dijkstra's shortest path algorithm and continues by incorporating a heuristic to choose the next node to expand. This is the feature of the A* algorithm that gives it its efficiency over Dijkstra's algorithm. The author describes his implementations as follows:

> *"The implementation of Dijkstra's shortest path algorithm can be gnarly to understand at first and I confess I've not been looking forward to writing this part of the chapter because I reckon it's not going to be any easier to explain."*

Buckland's implementation is indeed hard to understand and is implemented in no less than eleven C++ classes. It is however fast and flexible due to several

factors:

1. It uses templates for nodes and edges, therefore can be used with graphs of any type, (not just navigation graphs).
2. The heuristic is also a template, meaning that any heuristic can be used with the algorithm.
3. It uses its own data structures rather than the STL for the open and closed lists, including an implementation of an indexed priority queue for the open list. It has been suggested that greater efficiency will be achieved if the STL is not used for the data structures (our implementation suggests this is not necessarally the case).

On the other hand:

1. It is difficult to understand and requires eleven classes and many hundreds of lines of code to implement (see table 1). We suggest therefore it is not a good implementation for learning purposes.
2. In his demos, Buckland pre-generates the graph. This is the approach we follow in our adaptation of Buckland's implementation also. It uses a flood fill algorithm to pre-generate all the nodes. In Dalek World, this results in a graph of 16000 nodes. Were the world to be bigger, the graph would consequently increase. For example for a world of size 500 * 500, the graph would be 250000 nodes. We suggest therefore, that it would be more memory efficient were an alternative approach to be adopted that did not require that the graph be pre-generated.
3. The flood fill algorithm used first adds every single node to the graph, even those that are not navigable and then tags non-navigable nodes as invalid. We suggest that this is also inefficient as many nodes are added and stored unnecessarily.

Table 1: Classes used to implement Bucklands's A$*$ in Dalek World

Class	Description
SparseGraph	A sparse graph that uses node and edge templates
GraphNode	Base class for a graph node
NaveGraphNode	A node in a navigation graph
GraphEdge	An base class for an edge connecting two nodes
NavGraphEdge	An edge connecting two nodes in a navigation graph
GraphHelper	Generates a graph using the flood fill algorithm. Also draws the graph and generates a Path from the Graph_SearchAStar class
Graph_SearchAStar	This class implements the A* algorithm
PriorityQ	A heap based priority queue
PriorityQLow	A 2-way heap based priority queue implementation
Path	A container for a vector of waypoints
Heuristic_Euclid	Implements the Euclidian distance heuristic

Table 2: Classes used to implement A* in Dalek World

Class	Description
Node	A navigation graph node
PathFinder	Implements the A* algorithm
Path	A container for a vector of waypoints

4 Our A* Implementation

Having adapted Buckland's implementation, we felt that an alternative "student friendly" implementation had to be developed, which was equally as efficient but perhaps sacrificed some of the flexibility in order to be easier to understand. We therefore developed our own A* implementation using the STL. Our implementation has several features that make it a better fit for teaching:

1. It is implemented using just three classes. (See Table 2).
2. To use the implementation to calculate a path requires just as little as two lines of code. This compares with eleven for our adaptation of of Buckland's implementation.
3. It is implemented using the STL to hold the open and closed lists and therefore does not require the coding of any additional data structures.
4. It supports configurable cost and heuristic functions.
5. It does not require a pre-generated graph. Instead, nodes are added to the graph as required as the algorithm proceeds.

Our implementation creates graph nodes on the fly as required. Thus it does not require the graph to be pre-generated. A node is represented using an instance of the Node struct in Figure 3.

We need to access nodes on the open list by retrieving the node with the lowest f score, and also by position vector (to check if a node is on the list). We need to access nodes on the closed list just by position vector.

We first used the STL `list` template data structures to hold the open and closed lists. A list has a member function `sort`, which sorts the elements by a configurable sort order. To sort the list it is necessary to provide a class functor which compares two nodes. To retrieve nodes by position, `list` has a member function `find_if`. Using this function with a comparison functor argument, it is possible to retrieve a matching element, (to check if a node exists on the list).

Using STL `lists`, our implementation was as much as ten time slower than our adaptation Buckland's im-

Figure 3: A graph Node

```
struct Node
{
  D3DXVECTOR3 pos;
  float f;
  float g;
  float h;
  Node * parent;

};
```

Figure 4: Data structures used by our implementation

```
priority_queue<Node*, vector<Node*>, NodeLessFunctor>
_open;
map<D3DXVECTOR3, Node*> _openMap;
map<D3DXVECTOR3, Node*> _closed;
```

plementation. Profiling of our implementation revealed that our implementation spent up to 70% of its execution time sorting the open list and retrieving elements from the open and closed lists. This confirms [8]'s assertion that the representation of the underlying search space used will have an impact on the performance and memory requirements of a path finding system.

We then optimised the implementation in two ways. The first optimisation we performed was to use an STL `priority_queue` to hold the open list. A priority queue is a data structure in which only the largest element can be retrieved (popped) [1]. An STL `priority_queue` can be set up using a class functor to compare elements. This facilitates program defined sort order. The class functor we used is shown in Figure 5. Using this class results in the priority queue being sorted in ascending f score. Because elements are inserted *in order* into a priority queue, there is no need to sort the data structure. The second optimisation we used was to keep a *second* copy of the open list in an STL `map` data structure using node position vectors as a keys. As entries in both data structures are pointers, there is a minimal memory overhead involved in keeping a second copy of the lists in a second data structure. This eliminated the need for the sequential search. Declarations of all the data structures used are presented in Figure 4.

Figure 6 compares search times for our adaptation of Buckland's implementation with our implementation using the STL `priority_queue` and STL `map` data structures. To generate this data, twenty searches were performed using various start and end points in Dalek World. The program was then executed ten times (performing the same searches each time) and the times taken by each A* implementation were averaged. We observed that in fourteen of the twenty searches, our implementation either outperformed our adaptation of Buckland's A*, or was at least as fast. Overall our algorithm was faster by 34% that in fourteen of the twenty searches, our implementation either outperformed our adaptaion

Figure 5: Class functor used to sort the `priority_queue`

```
class NodeLessFunctor
{
    Node * p;
    public:
    NodeLessFunctor() { p = NULL;}
    NodeLessFunctor(Node * p) : p(p) {}
    bool operator()(Node * f1, Node * f2)
    {
      return (f1->f > f2->f);
    }
};
```

of Buckland's A*, or was at least as fast. Overall our algorithm was faster by 34%.

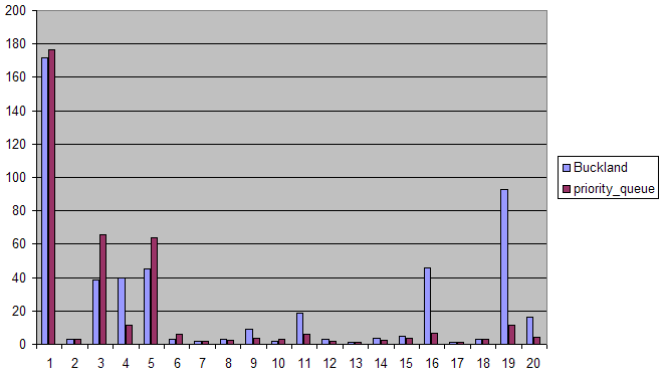

Figure 6: A comparison of average search time for our adaptation of Bucklands A* algorithm with our implementation based on the `priority_queue` and STL `map` data structures

Figure 7 compares search times for our implementation using the STL `priority_queue` and STL `map` data structures using three heuristics, the Euclidian distance, the Manhattan distance, the distance squared and using no heuristic. Again, twenty searches were carried out using various start and end points, the program was executed ten times and the average search times were profiled. As expected, using no heuristic, the number of nodes expanded was greatly increased and therefore the time taken by the search to complete was higher. What is interesting to observe from this data is that using the Euclidian distance heuristic is often fastest, despite the fact that the computational overhead for this approach would seem to be greatest. We speculate that this may be because to calculate the Euclidian distance, the DirectX API call `D3DXVec3Length` is used and this call must be hardware accelerated.

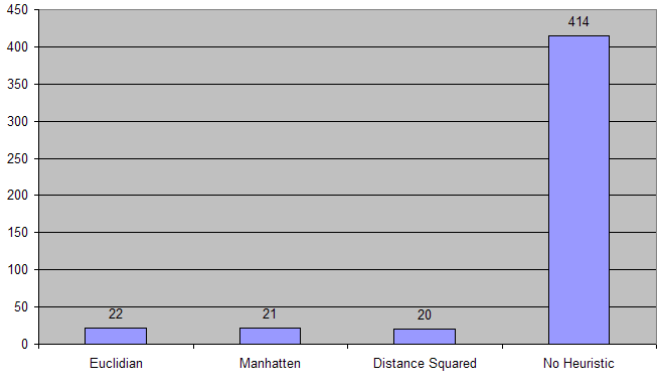

Figure 7: A comparison of search times in milliseconds using different heuristics

5 Conclusions

This paper presented our implementation of a student friendly yet efficient implementation of the A* algorithm. We first adapted Buckland's implementation of the A* algorithm in our game framework Dalek World. We conclude that this implementation is both flexible and efficient. However, we further conclude that due to the complexity of the implementation, it is not ideal for use in a teaching environment. We then described our own simplified implementation of the A* algorithm, generating graph nodes on the fly and using the STL `list` template data structure to hold the open and closed lists. We conclude that based on data gathered from search executions, this implementation is also not suitable because it is too slow. Finally we presented our optimised implementation of the A* algorithm using the STL `map` and `priority_queue` data structures to hold the open and closed lists. We presented data demonstrating that in Dalek World, it is on average the fastest of the three implementations. We can therefore say that it is possible to create a highly optimised implementation of the A* algorithm using data structures from the STL, if appropriate data structures are chosen. We also conclude that using three classes as opposed to eleven, our implementation is more suitable for learning the A* algorithm.

On the other hand, we acknowledge that our implementation lacks the flexibility of Buckland's implementation. We also conclude that due to the fact that our implementation queries the world repeatedly to check if a node is navigable, it is susceptible to optimisations in this area.

All the code used for teaching and testing this paper can be downloaded from http://www.comp.dit.ie/bduggan/games.

6 About the Authors

Bryan Duggan is a lecturer in the school of computing at the DIT in Kevin St. He hold a first class honours degree in computer science and software engineering from the University of Dublin (studied at the DIT) and a masters degree in Information Technology for Strategic Management (DIT). He is presently working on a PhD with a working title of "Modeling Creativity in Traditional Irish Flute Playing". He lectures on games programming and music technology in the DIT School of Computing.

Fred Mtenzi received his B.Sc degree from University of Dar es salaam - Tanzania in 1989. He also received his M.Sc and PhD from University College Dublin - Ireland graduating in 2000. Currently, he is a lecturer in the School of Computing at the Dublin Institute of technology - Ireland. His research interests include design and implementation of heuristic algorithms for combinatorial optimisation problems, security issues in mobile devices, games and healthcare systems and design of power aware protocols for Mobile Ad Hoc Networks.

References

[1] Leen Ammeraal. *C++ for Programmers*. Wiley, 2000.

[2] Mat Buckland. *Programming Game AI by Example*. Worldware Publishing Inc., 2005.

[3] Bryan Duggan. Learning games programming using "dalek world". May 2005.

[4] FarCry. *FarCry AI Bible*. Crytek, 2003.

[5] Mario Grimani and Mathhew Titelbaum. Beyond a*. *AI Game Programming Wisdom 2*, 2005.

[6] IGDA. *IGDA Curriculum Framework*. Crytek, IGDA.

[7] Patrick Lester. A* pathfinding for beginners. accessed from http:www.policyalmanac.orggames-astartutorial.htm, 2005.

[8] Paul Tozour. Search space representations. *AI Game Programming Wisdom 2*, 2005.