Path-finding Semester 1 Report
for CE0823a

This report looks to discuss three labs from the path-finding module during semester 1 of Computer Games Technology (G470). GPSA and hashing, the Lee algorithm and the A* algorithm have all been implemented in accordance with the lab sheets provided and the respective reflective logs are contained within this report. For the Lee and A* algorithms, the final code output have been provided to aid explanation. The following report will detail each lab in turn, discuss relevant results and give feedback on the task as a whole.

Lab. Work Reflective Log

Week 6: GPSA + hashing

Hashing is used to quickly look through a list or volume of stored values and assign values to them. It is particularly useful to ensure large amounts of, for example, images in a google image search have been correctly loaded. The algorithm can also be used to very quickly locate specific values in large data structures. The GPSA (General Problem Solving Algorithm) code provided is playing a game of noughts and crosses, if a row of three is found then that 'player' wins. Each cycle through the GPSA, nine new positions are given and values associated to each board position are given and then scanned by the hashing function. The following results will show how the code was implemented, discuss the results and give final feedback on the task as a whole.

Task – Hashing
Work done: The work was completed by taking the given GPSA code and implementing a hashing function. A bool array of 20000 elements was used to determine whether a previously generated board was the same as the current by a yes/no flag.

Discussion of results: It is clear that the large bool array is far faster than the GPSA. The ability to assign yes/no flags allows the code to process each instance almost instantaneously. Despite the altered GPSA code only using 765 elements in the 20000 element array, the performance improvement is significant. The large bool array was purposefully used as an extremity and therefore could save memory using extended hashing functions but for this experiment it was an appropriate test. The code also checks for duplicates. Although there aren't currently any

collisions through the way it was coded, had there been, the duplicates list would have been used to discard any anomalies. This can be particularly useful in large databases where multiple entries for the same item can occur and a hashing function is able to search the entire database and route out the duplicates.

Thoughts / things to remember:
This could be extremely useful when needing to make sure everything in a list has loaded correctly or has the correct values. Games programming would benefit from this by checking whether a data structure of enemies, for example, have all loaded the necessary textures etc.

Specific Points:

- Comparison between GPSA timings with large bool array hashing version:
    - As it can be seen below, the large bool array versus the GPSA code is significantly faster. The code is far more efficient and therefore is able to produce a result much quicker. This is down to the checking process simply being reduced to an array of yes/no flags. In release mode, this accounts for 0.197 seconds of array checking time. In real world situations it would a far greater time difference and therefore hashing is far more useful.

    - Large bool array
        - Debug: 2.023 sec
        - Release: 2.003 sec

    - GPSA
        - Debug: 2.6 sec
        - Release: 2.2 sec

- Any additional hashing experiments:
    - After applying the hashing function, STL hash tables were researched as an alternative method. It is again clear that this would not only clean up the code but also improve timings once again.

Things I have learnt about myself, my attitude, approach, performance or the task so far:
This task was approached with a small amount of hesitance as the task was not as clear cut as others presented previously. However, shortly after starting the task it became a lot simpler to understand where the methodology was going and how hashing could be useful. My approach to the subject is improving with each task and the added research suggestions are certainly aiding that also as well as my performance in the module as a whole.

In conclusion, it is obvious that hashing is far faster than the generic GPSA. The large bool array cut times significantly with a small alteration to the code and is far more useful. The benefits of hashing are many; quickly finding a particular reference in a large data structure, finding whether large lists of objects have correct items and values loaded and detecting duplicates like the GPSA code. It is clear that hashing would be useful in games programming and is a good base to work off of when considering STL hash tables.

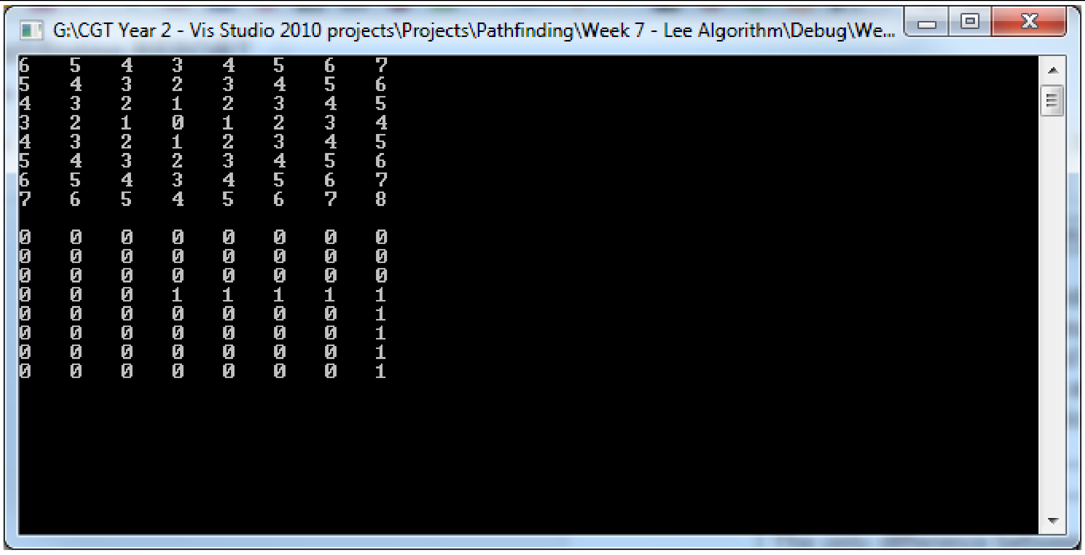
Lab. Work Reflective Log

Week 7: Lee Algorithm

The Lee Algorithm uses wave propagation and backtracking to find the shortest path from a starting point on a 2D grid to the desired target. Using a list to store the current nodes, it cycles through each node in turn and gives it a distance value from the start. Once the target has been found, the code will generate a 'shortest path' by backtracking through the nodes - finding the next shortest distance to the start each time. The following results will show how the code was implemented, discuss the results and give final feedback on the task as a whole.

Task – Lee algorithm
Work done: The work provided was to implement the Lee Algorithm to the GPSA code provided. This was done by starting from scratch and building up to the desired path-finding result using wave propagation and backtracking. Wave propagation circles each node in turn finding its neighbours and assigns them a distance value from the starting node. From this, the back-tracking function can find the shortest path from the target to the starting node by counting down the distances given to each node.

Discussion of results:
The algorithm successfully scans each node on the board in turn and gives it a distance value from the starting node. When the target node has been found, it again successfully traces the shortest distance back to the start and displays the desired result. Obstacles are no problem for the algorithm as it is able to move around the blockages and continue its path to the target. The provided graphical output shows the board with distance values inserted by the wave propagation and then again with the path highlighted.


G:\CGT Year 2 - Vis Studio 2010 projects\Projects\Pathfinding\Week 7 - Lee Algorithm\Debug\We...

```
6       5       4       3       4       5       6       7
5       4       3       2       3       4       5       6
4       3       2       1       2       3       4       5
3       2       1       0       1       2       3       4
4       3       2       1       2       3       4       5
5       4       3       2       3       4       5       6
6       5       4       3       4       5       6       7
7       6       5       4       5       6       7       8

0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       1       1       1       1       1
0       0       0       0       0       0       0       1
0       0       0       0       0       0       0       1
0       0       0       0       0       0       0       1
0       0       0       0       0       0       0       1
```

The algorithm moves in a NESW pattern in its search for the target. Using a double for loop and several conditions to prevent illegal moves, the algorithm is able to find the target in seconds.

Thoughts / things to remember:
The Lee Algorithm, although fast and effective, is basic in terms of what it can achieve. Not only this, but it wastes time by scanning the entire board for the target. Regardless, STL lists are easy to use and not code-intensive, therefore efforts should be made to use STL at all times despite the drop in performance it often has.

Specific Points:

- Problem representation
    - The problem faced was simple - start at a given point in a board and reach the target. It would be solved using data structures and lists. The board is a 2D array which is simply filled with integer values brought from the working set deque which has a struct data value. The struct ‘coord’ contains an x and y value to give co-ordinates to each node in the board. This allows the code to move throughout the board and know where it is.
- Choice of data structure for wset
    - Deque was chosen for this task due to its ability to have elements added or removed from both the front and back of the queue. This was particularly useful when it came to backtracking. After adding the nodes to the queue one way through wave propagation, backtracking was able to simply pop each node individually in the opposite direction.

Things I have learnt about myself, my attitude, approach, performance or the task so far:
This task took a long time to grasp. The limited instruction on the lab sheet, complicated GPSA code and little help found on google etc

made this task quite difficult. However, it pushed to broaden knowledge and understanding of path-finding and therefore was worthwhile. I feel that my attitude towards pathfinding has changed along with my approach to the subject and opened up the possibility of utilising path-finding in general games programming.

In conclusion, the Lee algorithm is a basic but effective algorithm for maze routing problems. Being able to set a start and end for the code allows it to quickly calculate the shortest path through wave propagation and backtracking. It may waste time by searching the entire board but its simplicity is very important to remember. The Lee algorithm is a good stepping stone in the direction of efficient path-finding and a good base to move onto the A* algorithm.

Lab. Work Reflective Log

Week 8 & 9: A* Algorithm

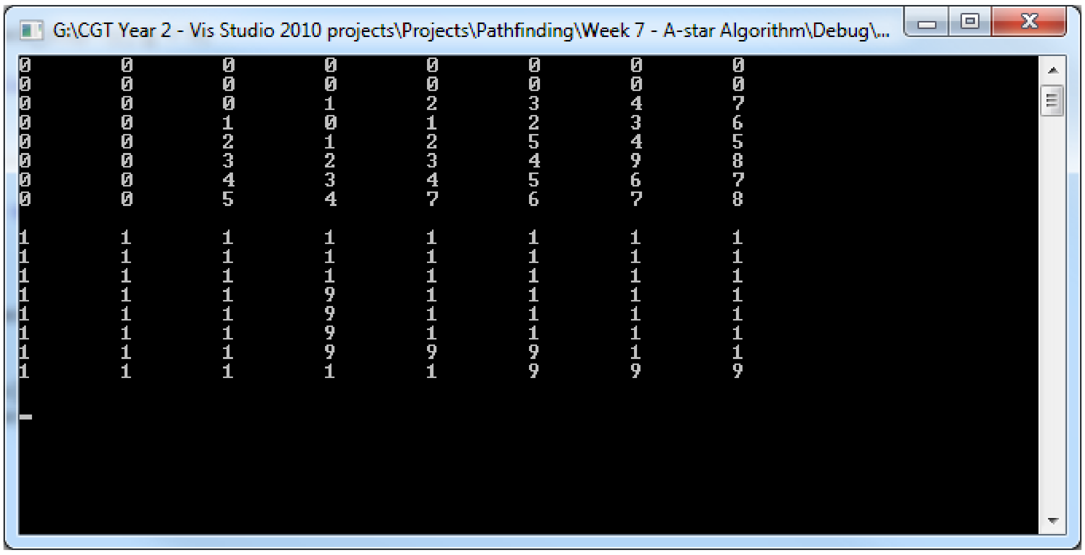
The A* algorithm is an extremely efficient method for moving through, for example, a board from a specified start point to a target. Despite being very similar to the Lee algorithm, A* introduces heuristics and is therefore able to calculate the shortest distance to the target from every node it comes across. This allows the algorithm to avoid the majority of the unnecessary nodes and head in the direction of the target from the start. The following results will show how the code was implemented, discuss the results and give final feedback on the task as a whole.

Task – Traceback algorithm, heuristic, priority queue and A*
Work done: The work completed consisted of implementing the A* algorithm. This meant using a priority queue, heuristics and features from the Lee algorithm. Both the A* and Lee algorithms utilise queues and back-tracing algorithms, meaning they are very similar in structure with the main difference being the inclusion of heuristics for A*.

Discussion of results:
The only difference between the A* algorithm and the Lee algorithm is the inclusion of heuristics. It allows the algorithm to calculate the distance between the current node and the target at any point.

A* utilises a priority queue instead of a deque used by the Lee algorithm. This is so the algorithm can focus the direction towards the target as apposed to searching the entire board through wave propagation. The priority queue has been set up to prioritise nodes with the smallest 'f-cost'. Where the 'f-cost' is calculated by adding the current path length and the estimated distance left to travel to the target.

The back-tracing function uses the nodes parents to find the shortest path. When a node was added to the open list, it was assigned a parent (the previous node it came from). It is then able to simply follow the parent pointers until the start is reached and a graphical output is displayed.



G:\CGT Year 2 - Vis Studio 2010 projects\Projects\Pathfinding\Week 7 - A-star Algorithm\Debug\...

```
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       1       2       3       4       7
0       0       1       0       1       2       3       6
0       0       2       1       2       5       4       5
0       0       3       2       3       4       9       8
0       0       4       3       4       5       6       7
0       0       5       4       7       6       7       8

1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       1       1       1       1       1
1       1       1       9       1       1       1       1
1       1       1       9       1       1       1       1
1       1       1       9       1       1       1       1
1       1       1       9       9       9       1       1
1       1       1       1       1       9       9       9

_
```

Thoughts / things to remember:
A* and the Lee algorithm are essentially the same with the addition of heuristics for A*. It is far more efficient since it does not need to search the entire board for the target. Heuristics finds the distance between the current node and the start and estimates the distance between the current node and the target, through this and the priority queue the algorithm is able to find the shortest path to the target.

A*, although more complicated than the Lee algorithm is far better and far faster at discovering the shortest path to the target. Because of this, it is more desirable in real world applications. Games could make good use of this over the Lee algorithm from the time and memory saving aspect alone.

Specific Points:

- Trace-back
    - The trace-back function is extremely efficient since it simply has to follow a list of pointers. Each time a node was added to the open list through the original target-finding process a pointer was added to the node it came from.
- Heuristic
    - The main difference between the Lee and A* algorithm. Heuristics calculates the current distance from the start

    - and the estimated distance to the target. This allows the algorithm to directly look for the target instead of searching through the entire board with wave propagation.
- Choice of data structure for wset
    - A 2D array of Nodes was set up to store board positions and any relevant information required. The struct allowed for multiple instances to be created and assign each node in the board specific values. This meant seeking certain board positions was made very easy along with being able to set what is happening in each node whether it be a blockage, starting node or target with one line of code.
- Priority queue
    - Priority queues are extremely useful when needing to find a certain condition-meeting value as quickly as possible. It was used in this case to provide the algorithm with the lowest heuristics cost of each node every time it moved. It did this by organising the new nodes by 'f-cost' and produced the lowest value as the next node to visit.

Things I have learnt about myself, my attitude, approach, performance or the task so far:
This task has allowed me to further broaden my knowledge of path-finding and heuristics. Both my attitude and approach to the subject have improved dramatically, making me appreciate pseudo code and planning far more. My performance of the task certainly was better versus my performance during the Lee algorithm task. I feel that with a little more practice, the A* algorithm could be very easy to use in games programming.

In conclusion, the A* algorithm is an extremely efficient method for moving through a board. It is able to move toward the target by way of heuristics and a priority queue. Back-tracing is made easy through the use of parent pointers which are implemented every time a new node is reached. Obstacles are no match for A* and when curved paths are required it is able to accommodate those also. Lastly, A* can be used in a wide variety of situations and therefore is a good candidate when path-finding is required in games.