Pathfinding Algorithms And Their Applications

Jonathan Huo, 8C7, Centennial Public School, Waterloo, Ontario
November 1st, 2015, to January 29th, 2016

Table of Contents

4. Bandwidth Search
5. Bidirectional Search
6. Jump-Point Search and Theta A*

vi. Applications in Gaming
1. Introduction
2. Benefits of A*
3. Scenarios
a. Exploration
b. Spying
c. Road Building
d. Terrain Analysis

d. Conclusion

C. Appendices
a. Bibliography / Works Cited

Section A - Project Process

Prior Knowledge - A, a

I began this Independent Study (IS) project with a solid foundation of knowledge on a variety of computer science (CS) topics. My interests were focused around algorithms and data structures, formal language theory, compiler design, operating system architectures, and algorithms used in gaming. Based on my knowledge of other CS algorithms, I already knew many basic graph traversal methods, but not much about pathfinding algorithms designed for gaming. From my use of the Unity game engine, I knew a small amount about A* (A Star) from reading their Application Program Interfaces (API), but not much about its internal design.

Inquisitions - A, b

My questions for this IS were centered around the A* search algorithm. Initially, I only had two questions; the first being about the usage of heuristics in A*, and the second being about how efficient different algorithms are in certain situations. As I proceeded with my research, however, my set of questions grew rapidly. I discovered that A* had many variants and optimizations, and I began wondering about the benefits of using different variants. This led to more speculation on how to implement these changes to A* in code as well.

Post-Study Knowledge - A, c

After pursuing the topic of pathfinding algorithms for two months, I have gained valuable insight on graph algorithms in computer science. I learned about the importance of heuristics in A* and how they can be modified to change the behaviour of the algorithm. I now also know about the many variants of the A* algorithm, and how to write robust implementations in code.

Section B - Research Paper

An Introduction to Pathfinding Algorithms - B, a

Introduction - B, a, i

Pathfinding algorithms are embedded at the heart of many modern technologies. From global positioning systems to packet routing protocols to game design, pathfinding is used everywhere. Essentially an elegant way of navigating mazes, these algorithms are an integral part of both computer science and mathematics.

This research paper covers both the most basic and the most complex pathfinding algorithms, as well as everything in between. The main focus is on the A* pathfinding algorithm. It delves deep into the inner workings of A*, and explains heuristics, algorithm variants, and optimizations in detail.

Note that this paper contains many references to commonly used terminology, data structures, and algorithms in computer science. The material is intended for audiences with a background in programming.

General Algorithmic Design - B, a, ii

The general design and operational flow of most pathfinding algorithms are somewhat similar. The algorithm takes in a map (graph) as input, and begins exploring nodes (vertex) from an initial node in the map until it reaches the destination node. When the destination is found, the algorithm returns the optimal path (usually the shortest) from the initial node to the end node. Since all pathfinding algorithms have the same functional goals, the only major difference between the many available is the logic behind the algorithm that prevents it from straying away from the approximate direction of the destination, as well as how to achieve the desired result with the lowest time and memory complexities.

In order to find the shortest path between any two given nodes, a wide variety of algorithms can be used, depending on what characteristics are desired of the returned path and the process used to find that path. The simplest pathfinding algorithms use brute force to find all possible paths, then compare these paths and find the shortest one. In the worst case, basic algorithms have a time complexity of $O(|V| \, |E|)$, or in some cases, $O(|V| + |E|)$, where V represents the amount of nodes in the graph and E represents the edges. However, time

complexities that have the potential to be $O(n^n)$ are generally unacceptable. In most games and real-world applications, the goal of pathfinding algorithms is not only to be as accurate as possible, but also be relatively fast. Since the complexities of simpler algorithms grow exponentially, it can lead to massive performance drops on large maps, or maps that require repeated navigation by multiple units. More complex algorithms use heuristics, dynamic programming, and computability theory to find shortest paths. By removing nodes from the candidate set that are known to damage accuracy, these algorithms can operate in time complexities as low as $O(|E| \log |V|)$ while still maintaining absolute accuracy.

All pathfinding algorithms are solutions to the Travelling Salesman Problem (TSP), and thus run in Nondeterministic Polynomial (NP-Hard) time. Because of the similarities TSP has to so many real-world problems that require pathfinding, such algorithms are optimal solutions for Global Positioning Systems (GPS), Artificial Intelligence (AI) problems, network routing protocols, and game design.

In practical implementations, most, if not all pathfinding algorithms operate on the same concept. On every iteration, the algorithm performs a check on the current node being evaluated, which generally involves finding the neighbouring node that best satisfies some predefined rule (in practice, this rule is usually “closest neighbour to the goal node”). Following this, the algorithm will then apply the same set of checks to the best neighbouring node, until it finds a node that satisfies the rule to some given degree.

Commonly Used Pathfinding Algorithms - B, b

Deep-First Traversal - B, b, i

Deep-First Traversal / Search (DFS) is a simple, brute-force pathfinding algorithm that is commonly used and observed by computer scientists. The DFS algorithm was invented by French mathematician Charles Pierre Trémaux in the late nineteenth-century as a method of solving complex mazes, and is probably the earliest known pathfinding algorithm.

DFS traverses a map by exploring to the end of each branch of nodes and backtracking if a solution is not found. The algorithm starts at an initial root node and navigates down any branch in the map until it either reaches the destination or a dead end. If the latter is true, DFS backtracks up the previously traversed branch until it reaches a node in the map that has a neighbour that has not been examined. Once it finds an unexamined node, the process is repeated until a destination node is found or the graph no longer contains any unvisited nodes.

In code implementations, DFS is comprised of three sets of nodes; *unvisited*, *visited*, and *path*. The *unvisited* set of nodes represents the nodes in the map that the algorithm has not yet examined, and in vice versa, *visited* contains all nodes that have been explored. The *path* set is a stack that contains the nodes that are on the current traversal tree. As nodes are evaluated, DFS adds it to the top of the *path* stack, and subsequently, when backtracking, nodes are popped off

the top of *path*. At the end of navigation, DFS simply returns the *path* stack, which should contain all the nodes that must be visited in order to reach the destination.

Logically and asymptotically, DFS is one of the worst algorithms for map navigation. The running time for DFS is $O(|V| + |E|)$, because it is a brute force algorithm. Since there is no logic in the programming that gives it a general sense of where to navigate towards, DFS can end up traversing entire maps. Also, the algorithm does not perform any "breadth-first" navigation; DFS explores branches one at a time as opposed to multiple nodes on the same distance layer, which is a major drawback. Consider the following situation: a map has two branches, *A* and *B*, each containing 1,000 nodes. Suppose the destination node is the second node on branch *B*. Because the algorithm navigates individual branches, DFS will need to traverse all 1,000 nodes on *A* before reaching the destination on *B*. Due to these design flaws, DFS is rarely used on maps requiring intensive navigations; rather, it is more suitable for problems such as mass traversal and exploration.

Breadth-First Traversal - B, b, ii

Breadth-First Traversal / Search (BFS) is the much more commonly used relative of Deep-First Search. Invented in the late 1950's by Edward Forrest Moore, instead of navigating down branches one-at-a-time and backtracking, BFS searches for nodes on the same distance level away from the source node.

With BFS, the algorithm navigates nodes down each branch by one node on every iteration, creating a "breadth-first" expanding motion when visualized. Starting with the initial node, BFS visits all of its neighbours, and then visits the neighbours of those neighbours. This is usually applied recursively, and repeats until the set of unvisited nodes is exhausted or the destination is found. Set representation in BFS is exactly the same as in DFS; however, instead of using a stack to record the current path, each node has a parent reference that points to the node the algorithm processed before visiting it. Once the destination is reached, the path from the starting node can easily be found by backtracking through the parent references.

Although it is still a brute force algorithm, BFS is generally favoured over DFS because of the way nodes are visited. The time complexity for BFS is $O(|V| + |E|)$.

Dijkstra's Algorithm - B, b, iii

Dijkstra's algorithm is, strictly asymptotically, the fastest and most accurate pathfinding algorithm available. The algorithm was first conceived by Dutch computer scientist Edsger W. Dijkstra in 1956, while trying to demonstrate the computing capabilities of ARMAC, a 6-bit computer. The majority of research done in the field of pathfinding algorithms is largely based on Dijkstra's algorithm, with the entire branch of A* having Dijkstra's idea of tentative costs at its roots.

Dijkstra's algorithm computes the shortest path between the start and destination nodes by visiting nodes with the lowest tentative distance costs, which are calculated by adding up the traversal costs of all nodes that are in the currently visited path. The actual navigation process of Dijkstra's algorithm is similar to a guided BFS. Starting from the initial node, the algorithm explores neighbours with the lowest tentative distance between it and the destination. This process is then repeated until either the goal node is reached or the algorithm visits every single node in the map. Below is the algorithmic process for Dijkstra's algorithm:

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3."

For set representation, Dijkstra's algorithm uses two arrays of nodes. The first array, *open*, contains all the nodes that are candidates for examination. At the beginning of the algorithm, every node except for the starting node is in the *open* set. Once Dijkstra's algorithm begins exploring, nodes that are visited are removed from the *open* set and added to the *closed* set. Anything within *closed* is not evaluated again.

Although Dijkstra's algorithm is guaranteed to return the absolute best path every time, it does significantly more work-per-cycle than other algorithms that implement heuristics for strategic candidate elimination. Because of this, Dijkstra's algorithm is usually used on paths that require infrequent or single-unit navigation, such as networks in packet routing protocols.

The A* Pathfinding Algorithm - B, c

Introduction - B, c, i

The A* (A Star) search algorithm is arguably the most important algorithm used for pathfinding, and is known for its accuracy and speed. The original A* algorithm was developed by Peter Hart, Nils Nilsson, and Bertram Raphael of Stanford Research Institute (SRI) in 1968, as a faster variant of Dijkstra that uses heuristics to reduce the amount of work the algorithm does - making it ideal for usage in games and other programs requiring high performance.

Algorithmic Design - B, c, ii

A* combines information from both Dijkstra's algorithm and an estimated heuristic cost function to run fast, and yet maintain a high degree of accuracy. The core of the algorithm is based on a simple mathematical equation: *fCost* = *gCost* + *hCost*; where *gCost* represents the tentative distance-to-goal cost of a node returned by Dijkstra's algorithm, and *hCost* is the approximate distance-to-goal cost of a node given by the heuristic function. The above equation is applied on every iteration of the algorithm, during the stage where it tries to find the optimal node to visit. In A*, the ideal neighbour node to visit is the one with the lowest *fCost*. This leads to the algorithm exhibiting interesting and varied behaviour in certain conditions. When *hCost* returns zero, *fCost* evaluates to *gCost* + *0*, turning A* into Dijkstra's algorithm. As the heuristic function begins to return results that get closer to *gCost*'s output value, A* will stop visiting nodes that are known to increase the cost of the path by more than the heuristic estimate. Lastly, when *hCost* becomes substantially greater than *gCost*, the accuracy of the path decreases, but the algorithm runs faster. A* achieves its speed and accuracy by keeping a balance of the above situations.

Heuristics - B, c, iii

Introduction - B, c, iii, 1

The unique part of A* that makes it faster and smarter than other algorithms is the heuristic cost function. Heuristics are used by A* to improve its ability to eliminate nodes that are unworthy of visiting, as well as to change the general behaviour of the function; allowing for dynamic navigation based on factors such as map design, CPU speed, and time limits. This makes A* a very versatile algorithm, because it enables the pathfinder to specify constraints and navigational factors such as movement cost, unit avoidance, and heuristic imbalance.

Heuristic Functions - B, c, iii, 2

Depending on certain navigational factors, different heuristic functions may provide more benefits than others. On maps where nodes are not represented on a coordinate grid, the heuristic will usually be a BFS that runs before the main algorithm. Because non-coordinate maps do not allow for vector-based distance evaluations, the BFS simply assigns each node a distance value that represents the number of edges between it and the destination node. A*, however, is normally applied to tiled vector maps, so heuristics that work with vectors coordinates are used. On such maps, there are four major heuristics:

- Manhattan Distance for tiled maps allowing four directions of movement
  - *distance = D * (destinationNodePosition - currentNodePosition)*
- Diagonal Distance for tiled maps allowing eight directions of movement
  - *distance = D * (destinationNodePosition - currentNodePosition) + (D2 - 2 * D) * minimum (destinationNodePosition - currentNodePosition)*
- Euclidean Distance for maps allowing any direction of movement
  - *distance = D * squareRoot ((destinationNodePosition^2 - currentNodePosition^2))*
- Hexagonal Manhattan Distance for hexagonal maps allowing six directions of movement
  - *distance = D * ((destinationNodePosition3D - currentNodePosition3D) / 2)*

Each of the above heuristic functions operate on the same principle of subtracting the position of the current node being examined from the position of the destination node, which returns the linear distance between the two. The simplest heuristic of the four is Manhattan, which takes the distance between *destinationNodePosition* and *currentNodePosition* and fixes it by some value *D*; this scaling factor should be set to match the *gCost* function.

Diagonal Distance deals with maps where nodes have eight adjacent neighbours (horizontal, vertical, and diagonals). In this heuristic, the algorithm computes the amount of steps taken without traversing diagonals, and subtracts from it the amount of steps taken with diagonals. The benefit of taking paths with diagonals is that it reduces the number of nodes visited *2 * D* times, but the cost of traversal increases by a factor of *D2*. In cases where *D = 1* and *D2 = 2*, the heuristic is called the Chebyshev distance function, and when *D = 1* and *D2 = squareRoot (2)*, the heuristic is known as the octile distance function.

With Euclidean Distance, units on the map are free to move in any angle, so the heuristic function computes the Pythagorean vector distance between *currentNodePosition* and *destinationNodePosition*. Using Euclidean Distance causes the performance side to suffer, however. Since all angles of movement are permitted, Euclidean heuristics will return distances significantly shorter than Manhattan or Diagonal would. This causes the balance between *gCost* and *hCost* to be shifted towards *gCost* (since *hCost* is smaller than it would be normally). When this happens, the performance-heavy Dijkstra's algorithm becomes dominant - causing A* to run slower.

On maps that are split into hexagonal tiles, the Hexagonal Manhattan heuristic is used. Rather than taking the two-dimensional (2D) hexagonal distance between nodes, the function converts all hexagons into cubes in three-dimensions (3D), and computes the much simpler vector-distance between the cubes. Since 2D-hexagons have the same number of adjacent nodes (faces) as a 3D-cube does, it is mathematically equivalent to represent the hexagons as 3D objects.

Cost Functions - B, c, iii, 3

The concept of cost functions is an integral part of the A* algorithm. In many situations, maps contain nodes that have navigation costs, which represent how expensive it is to traverse through a node. With this, the best path may no longer be the one that visits the least amount of nodes - a better path may be one that contains more nodes but has a lower overall traversal cost. This enables the algorithm to eliminate nodes that have a cost higher than the expected or average traversal cost. To implement cost-based node elimination, it is common for *fCost* to evaluate a cost function within *hCost*. In an example map, there might be two types of nodes, *quick* and *slow*, each having a traversal cost of one and five, respectively. Since the cost function prefers to navigate across nodes with a lower traversal cost, A* will search five times more often on *quick* nodes than *slow* ones.

Implementation and Analysis - B, c, iv

Outline - B, c, iv, 1

All A* implementations follow a similar design pattern. There are two sets of nodes, *open* and *closed*, which represent the candidate nodes and visited nodes, respectively. When the algorithm begins navigation, the *closed* set only contains the starting node, since all other nodes have not been examined yet. As A* runs, it builds up a tree of partial paths known as the navigation tree. Each branch of the tree represents a path that A* has considered visiting, and each node on the navigation tree represents a node in the map that has been examined. The *open* set contains the leaf nodes of the navigation tree; this can be thought of as a “frontier of exploration”. All visited nodes also have a reference to the node that the algorithm processed before them, which allows for A* to retrace the path through the parent references once it reaches the destination node.

In code, the A* function is comprised of a main outer loop and multiple secondary inner loops. The main loop iterates through the *open* set until a path to the destination node is found, or until the open set is empty. If the latter is true, then the algorithm exits and returns an error. Otherwise, the outer loop will find the node in *open* with the lowest *fCost* value and evaluate it. This node is known as the *current* node or *head* node. If *current* is the goal node, then A* will

trace the path from *current* to the starting node via the parent references. Otherwise, the *current* node is added to the *closed* set and its neighbours are examined by a secondary loop. If the neighbour being evaluated is already in the *closed* set, then it is ignored unless its *fCost* value can be changed. Otherwise, the neighbour has its parent reference set to the *current* node and is added to the *open* set. The tentative *gCost* value for the neighbour is updated as well.

Pseudo-Code - B, c, iv, 2

Below is the pseudo-code for the original version of the A* search algorithm. Note that this A* function assumes that the heuristic produces a monotonic (consistent) output for the same node and thus does not reevaluate the *fCost* of any node in the *closed* set.

```
function A*(start,goal)
       ClosedSet := {}         // 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 := map with default value of Infinity
       g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
       f_score := map with default value of Infinity
       f_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)

              OpenSet.Remove(current)
              ClosedSet.Add(current)

              for each neighbor of current
                     if neighbor in ClosedSet
                     continue              // Ignore the neighbor which is already evaluated.
                     tentative_g_score := g_score[current] + dist_between(current,neighbor)
// length of this path.
                     if neighbor not in OpenSet   // Discover a new node
                            OpenSet.Add(neighbor)
                     else if tentative_g_score >= g_score[neighbor]
                            continue              // This is not a better path.

                     // This path is the best until now. Record it!
                     Came_From[neighbor] := current
                     g_score[neighbor] := tentative_g_score
                     f_score[neighbor] := g_score[neighbor] +
heuristic_cost_estimate(neighbor, goal)
```

```
        return failure

function reconstruct_path(Came_From,current)
        total_path := [current]
        while current in Came_From.Keys:
                current := Came_From[current]
                total_path.append(current)
        return total_path
```

Performance - B, c, iv,

Performance is a major problem for many pathfinding algorithms. Although the original A* algorithm is aimed to be performance-friendly, many further optimizations can be made:

- Decrease graph size by using waypoint nodes instead of a tile map
- Improve heuristic accuracy and speed
- Make set manipulation operations easier by using specialized data structures and algorithms

To decrease the map size, it is recommended that the pathfinder uses a waypoint-marked navigation mesh rather than a grid of tile nodes. With this approach, the size of the map can be reduced dramatically, since A* will only navigate by jumping between waypoints, rather than traversing individual nodes. Using waypoints can prove to be extremely useful, especially on maps containing large open areas. Reducing the amount of nodes that A* needs to visit can also be done by storing open areas in quadtrees. With such a data structure, the algorithm can quickly skip over areas containing no obstacles.

One method commonly used to improve the speed of the heuristic is using a hierarchical map approximation. This can be visualized as placing a coarse, estimated grid over the actual tile map, and having the heuristic use the coarse grid to find paths quicker.

Set representation is one of the major chokepoints restricting the A* algorithm. On every iteration, A* reads, writes, and analyses data retrieved from both the *open* and *closed* sets. In most implementations, three fundamental set operations are performed: the main loop finds the best node in the *open* set (remove-best), the inner loop checks if neighbouring nodes are in the *closed* set (membership test), and at every stage, A* inserts and removes nodes between the two sets (insertion). On maps where the heuristic is not monotone, an increase-priority function for nodes with changing *fCost* values must be implemented as well. Selecting the proper data structure to manage and represent *open* and *closed* is key. If implemented using a basic array or linked list, A* easily reaches time complexities of $O(n^2)$. Thus, it is important to use more robust data structures such as heaps and priority queues, since bad set representation can easily drop performance rates on large maps.

Set Representation - B, c, iv, 4

Unsorted Arrays and Linked Lists - B, c, iv, 4, a

On a standard unsorted array, all three underlying operations run in an unacceptable time complexity of $O(n)$. Membership testing requires linear search, insertion requires all elements to be shifted, and increase-priority takes $O(n)$ to find the node to increase and $O(1)$ to change the value. Using a linked-list can reduce insertion time to $O(1)$.

Sorted Arrays - B, c, iv, 4, b

With a sorted array, membership testing and increase-priority are both reduced to $O(\log n)$ time. Since all elements are sorted, binary search can be used on the collection. Remove-best becomes $O(1)$, because the best node is sorted to the end of the array. Insertion, however, remains at $O(n)$. The one major drawback of using a sorted array is that every time the array is accessed, it must be sorted.

Binary Heap and Fibonacci Heap - B, c, iv, 4, c

Binary and Fibonacci heaps are the best choices for set representation in most situations. Rather than using pointers that refer to child nodes, each node on one of these two heaps is assigned an index value, which allows for fast random access. Both insertion and remove-best can be executed in $O(\log n)$ time. Increase-priority takes $O(n)$ for locating the node and $O(\log n)$ to increase its priority. Although Binary and Fibonacci heaps must be sorted as well, it only takes $O(\log n)$ time as opposed to $O(n \log n)$ on arrays.

Analysis - B, c, iv, 5

Asymptotically, the best choice for set representation is a Binary or Fibonacci heap. However, under different operating specifications, one type of data structure may outperform another, even if it has a greater time complexity. With low set sizes, a linear increase in time complexity may actually be faster than an algorithm that increases logarithmically.

Variations - B, c, v

Beam Search - B, c, v, 1

Beam search is a variation of A* where the open set has a limit on size. When the amount of nodes in the open set exceeds the limit, nodes with the worst chances of yielding a good path are ignored (dropped). While this model does increase efficiency by elimination, the catch is that the open set must be sorted for the algorithm to determine the priority of each node. Beam search is usually implemented using a Fibonacci heap for fast set manipulation.

Iterative Deepening - B, c, v, 2

Iterative deepening (ID) is a technique that can be used to dynamically change the balance between A*'s speed and accuracy. ID is typically applied to strategy games where the computer examines possible future moves until it finds a set of moves that it is satisfied with, such as chess and checkers.

In this variation, A* takes two extra input parameters, *cutOff* and *increase*. *cutOff* specifies the initial *fCost* drop value, and *increase* is the amount that *cutOff* will grow on every pass. The first time the algorithm navigates through the map, any nodes with a *fCost* higher than *cutOff* are ignored (not added to the open set) - causing A* to visit very few nodes on its first iteration. On each subsequent navigation, the algorithm increases the cutoff value. If the path returned by the next iteration is very different or substantially better than the previous one, IDA* will continue to increase *cutOff* by *increase* until it is satisfied with the path.

Dynamic Weighting - B, c, v, 3

With dynamic weighting, A* evaluates a scaling function, *weight >= 1*, that is associated with the heuristic. This function is generally implemented in the form of *fCost = gCost + (weight * hCost)*. The *weight* function assumes that at the beginning of the search, moving quickly is important. As the algorithm gets closer to the destination, the *weight* function decreases its output value and the heuristic becomes less powerful.

Bandwidth Search - B, c, v, 4

Bandwidth search reduces the amount of nodes A* visits by dropping nodes that are proven to be detrimental to the accuracy of the path. This modification of the algorithm assumes that the heuristic function is an overestimate, meaning that *hCost* returns values greater than the actual distance-cost of a node. However, the algorithm also conjectures that the heuristic will not overestimate by some value *e*. Thus, as with most other variations of A*, Bandwidth search works best with a more accurate heuristic. This gives it the unique "banding" property; all nodes with a *fCost* that is $e + d$ (for some heuristic *d*) greater than the *fCost* of the best node in open can be ignored, since it is proven that they will not be on the best path.

Bidirectional Search - B, c, v, 5

Bidirectional A* separates the pathfinding process so that the two instances of the algorithm navigate towards the *head* node of the other algorithm and meet in the middle. By splitting the amount of nodes processed by one algorithm across multiple threads, implementing bidirectionality reduces the overall running time of A*. With this model, instead of having one large navigation tree on the map, there are multiple small trees that interlace with each other.

Naturally, there are many approaches that can be taken when incorporating bidirectional searches. The first variation of bidirectional A* is the front-to-front navigation model. Instead of choosing a single node that has the lowest *fCost*, front-to-front chooses a pair of nodes, *a* and *b*, which have the lowest $(gCost (a , start) + gCost (b , goal)) + hCost (a , b)$ value. This effectively joins the two searches together, without having all of the computations being done on one instance of the algorithm.

Another solution to the bidirectional navigation problem is using retargeting. Retargeting bidirectional A* works by alternating the search destination used by each instance of A*. For a specified number of iterations, one instance will navigate towards the *head* node of the other. This process then alternates between the two running algorithms, until they meet in the center.

Jump-Point Search and Theta A* - B, c, v, 6

Jump-Point A* (JPA*) and its relative, Theta A* (TA*) , are used in situations where calculation is costly. Instead of computing A* on every single node, JPA* and TA* generalize paths by creating waypoints on nodes that are located on intersections. With this, the algorithm can skip across open areas and navigate directly to waypoint nodes, thus reducing the amount of time needed to calculate paths.

When implementing JPA* / TA*, there are two approaches that are commonly used. The first method of applying JP is by preprocessing the map and creating waypoints when an intersection is found (this is usually done with a modified BFS). Every time the preprocessing algorithm reaches an intersection node, it will record it as a waypoint, and begin finding waypoints that are reachable from the current one. When marking down waypoints, it is very important to record the cost of navigating to the waypoint. This way, JPA* can determine how expensive it is to cross the nodes that connect waypoints. However, the first approach is only beneficial for use on static (non-changing) maps that require repeated navigation, since the preprocessing algorithm requires more resources than JPA* itself. The other way of achieving waypoint-based navigation is to only add the most distant visible nodes to the open set. With this computation model, each step is more complex to calculate, but there are less of them in total.

Applications in Gaming - B, c, vi

Introduction - B, c, vi, 1

The most frequently observed use of pathfinding algorithms is in game design. Every single type of video game, ranging from Real-Time Strategy (RTS) games to First-Person Shooters (FPS) all have AI software packages and API's that are designed for complex pathfinding. Because the amount of situations in gaming that require map navigation is gigantic, pathfinding algorithms have evolved hand-in-hand with the gaming industry to produce newer, and more robust techniques for solving intricate problems related to navigation.

Benefits of A* - B, c, vi, 2

Of the many pathfinding algorithms available for use, A* is preferred over others because it offers an edge when it comes to satisfying certain game design principles. In video game programming, keeping performance in check while maintaining high-realism is a major challenge. Since many modern games are extremely graphics-intensive, using algorithms that are less costly on performance can dramatically improve rendering speeds. Because of this, A* is the go-to algorithm when implementing pathfinding in video games - it provides an almost perfect balance between speed and accuracy.

Another reason why A* is favoured by game developers is because it is incredibly adaptable. The heuristic function that the algorithm uses can be easily modified as to change the properties of returned paths. With this versatility, A* can produce paths that are needed in every gaming scenario.

Conclusion - B, d

Pathfinding is an incredibly diverse and useful field of research in computer science and mathematics, with applications in a variety of real-world situations. Algorithms used for pathfinding not only represent methods of navigating maps - they are also accurate mathematical models of solutions to problems in computational theory. Today, programmers and theoretical computer scientists alike are constantly finding new algorithms for solving practical pathfinding problems. The study of pathfinding algorithms continues to remain a valuable and ever-growing subject.

Section C - Appendices

Bibliography / Works Cited - C, a

Online Sources:

- "A* Search Algorithm." Wikipedia. Wikimedia Foundation. Web. 14 Dec. 2015.
- "Algorithms - GeeksforGeeks." GeeksforGeeks. Web. 14 Dec. 2015.
- "Bellman-Ford Algorithm." *Wikipedia*. Wikimedia Foundation. Web. 31 Jan. 2016.
- "Breadth-First Search." *Wikipedia*. Wikimedia Foundation. Web. 01 Feb. 2016.
- "Depth-First Search." *Wikipedia*. Wikimedia Foundation. Web. 01 Feb. 2016.
- "Dijkstra's Algorithm." Dijkstra's Shortest Path Algorithm in Java. 2 Nov. 2009. Web. 14 Dec. 2015.
- "Dijkstra's Algorithm." *Wikipedia*. Wikimedia Foundation. Web. 31 Jan. 2016.
- Graham, Fan Chung, and Lincoln Lu. "CSE 202 Greedy Algorithms." *04greed - 04greed.pdf*. University of California, San Diego, 2002. Web. 13 Jan. 2016.
- "Granberg, Aron."Get Started With The A* Pathfinding Project." A* Pathfinding Project: Main Page. Web. 17 Dec. 2015.
- "Iterative-Deepening A*." *Wikipedia*. Wikimedia Foundation. Web. 01 Feb. 2016.
- "Pathfinding." Wikipedia. Wikimedia Foundation. Web. 4 Jan. 2016.
- Patel, Amit. "Amit's A* Pages From Red Blob Games." Amit's A* Pages. Amit Patel. Web. 14 Dec. 2015.
- "Search Algorithm." *Wikipedia*. Wikimedia Foundation. Web. 01 Feb. 2016.

Video Sources:

- Lague, Sebastian. "[Pathfinding Tutorial Series] The A* Algorithm." *YouTube*. YouTube, 16 Dec. 2014. Web. 17 Dec. 2015.