Tank War

Product Owner
Adam Hubble

SCRUM Master
Josh Howitt

Team Members
Alfie Horton Joseph Hill

Software Design Decisions

A* search

Acknowledging all the path finding techniques that could be used, we had chosen to primarily implement A* search, believing it to be the most accurate and able to calculate the shortest path, as a guided search method which equips a heuristic function. In knowing of heuristic function capabilities, of being able to calculate an estimate of the minimum cost between a starting and end (goal) node, via ranking and the creation of alternative paths at every branch step; prevails our understanding of fewer nodes needing to be visited to reach the goal node. From such, we envisioned a more challenging opponent to the player with the implementation of the A* search technique. From which we had assumed that the bot tank would exhibit behaviours that seemed more ‘player-like’ and often in comparison to the other graph search techniques, this is from knowing the efficiency of the paths it calculates. Alike the player, the bot tank would attempt to traverse the shortest distances to reach an objective in view and therefore would assume to have more frequent encounters with its objectives; each encounter would determine the presented behaviour(s) of the tank. In summary, we had hoped of presenting a more challenging opponent to the player from its use.

Moreover, our initial heuristic function selected was Euclidean without square root, this was due to its ease of implementation and speed which was apparent due to a low demand for computational processing; the exclusion of a square root function enables this. But, alongside the exclusion of the square root function we had also accepted the possible compromise of accuracy as a result of its ‘trade-off’ for speed; however, our findings determined that there was no significant difference in accuracy between the other Euclidean (with square root) and Manhattan Heuristic functions. Collectively, we concluded that A* search in conjunction with the Euclidean without square root function, was the most suited and therefore most advantageous for our project, and that the project was more speed orientated as opposed to accuracy given its flat terrain concept.

Depth-first Search (DFS)

Additional to A* search, Depth-first Search (DFS) has been implemented as a technique being one of “several different AI techniques” in accordance to the assignment specification; it is necessary to include several techniques to determine the best path finding algorithm. Given this context, its implementation (comment blocked) purposes as a comparative technique to the A* search query, which serves as our pre-eminent technique. Acknowledging the capabilities of Depth-first Search, the technique requires less time to calculate pathing in comparison to A* search. However, Depth-first Search proves to traverse (visit) many more nodes, in order to reach the goal node of the path calculated, which undoubtedly hinders the tanks behaviours as an opponent. Moreover, in relation to the expense of time used to calculate paths, the difference between A* search isn’t significant in comparison, but due to DFS’s compromise for path efficiency (accuracy) and sometimes inability to even create paths, disadvantages the technique when compared to A* search.

Breadth-first Search (BFS)

Alongside the A* search and Depth-first Search (DFS) techniques, Breadth-first Search (BFS) has also been implemented within the program code, as another instance of “several different AI techniques” used. As mentioned within the project management documentation, its implementation was only purposed for providing a comparison between the other existing techniques, from which we are able to determine the better technique for the bot tank and its behaviours to perform as an opponent. Regarding the capability of BFS, the method alike DFS requires less time to calculate paths in comparison to A* search. But unlike DFS, BFS traverses (visit) fewer nodes in order to reach the goal node of the path calculated. This is comparing to DFS, whereas A* search traverses within the same number of nodes as BFS, but the path takes longer to be traversed. Similarly to the results seen between the A* search and DFS methods, A* search once again utilises more time to calculate paths, but the difference relative to calculation is too insignificant to be noticed in real-time execution. Alike DFS, BFS is also disadvantaged by its compromise for path efficiency (accuracy), and so summarises A* search to be the superior path finding algorithm.

Finite State Machine (FSM)

In extension to the numerous pathfinding techniques implemented within the project program, a series of Finite State Machines have been incorporated to govern the behaviours of the bot tank; in which each state of both state machines has a unique behaviour associated and are performed based on given conditions. Conditions determine the states that are being transitioned to and from, which are relative to the types of stimuli inside of the tank's proximity and its attributes; the conditions operate as virtual sensing capabilities to the tank. Furthermore, the FSM's partially conform to a hierarchical arrangement in focus of their states and sub-states; this is presented by the prioritisation of states that has been used to provide scalability across the FSM's. Prioritisation is managed by a series of standalone or nested 'if else' statements, containing Boolean conditions. All of such are components which make up the conditions used to determine the tanks state(s).

Acknowledging the ease of implementation and adjustment of a Finite State Machine, we had chosen to adopt FSM's to accommodate our needs for being able to implement path finding techniques with ease. In correlation of its use, aspects of debugging have been made simpler also, the application of states has enabled the isolation of problems, quickening problem resolution in result. To note, system performance also promises stability from its use, this is simply due to the low computational overhead which Finite State Machine's incur.

Moreover, the FSM's have two core capabilities, one to alternate movement behaviours of the bot tank; the other to alternate its mounted turret behaviours. Relative to movement, some of the behaviours include 'searching', 'following' and 'stopping'; meanwhile turret behaviours involve 'searching aim' and 'aiming'. All states are performed in conjunction with the A* search technique, and wholesomely act as a thought process for the bot tank, which configures a challenging opponent to the player. Without doubt, our inclusion of FSM's has hastened our project completion as well as enabled desirable functionality/ the maintainability of program code. Thus, we conclude our use for Finite State Machines to be a sensible software design decision.

Implementation of Program Functionality

Regarding the implementation of program functionality, three additional classes were created to enable the path finding, behaviours and sensual awareness of the bot tank. Initially, we had created the 'map' and 'mapNode' classes. For which were used to calculate paths for the bot tank, when using different path finding algorithms; an exampling method being the A* search query. Additionally made to these classes was the 'newTank' class, which contains the Finite State Machines and methods for calculating the values each FSM uses. All of such enables the bot tank to exhibit behaviours depending on its transitioned-to-state, which is based upon the objects that the tank senses around itself in addition to the values of each class variable.

Throughout the development process of the project, we had progressively approached problem resolution and or the overcoming of program errors. In the first instance, we had focused on achieving the bot tanks turret to aim at every objective in the map space, whist preserving the movement functionality from the 'DumbTank' class. We had approached such by comparing the difference in position between the bot tank and the closest objective to it in the map space, and then further calculating the angle between the bot tank and objective. This was calculated from the vector distance between the closest objective and the bot tank, via the 'calcAngle()' method.

Later, we had initiated the implementation of the A* search technique, in attempt to replace the 'DumbTank' movement with pathfinding to a given goal node. Proceeding its implementation, we had then focused upon the Finite State Machines, which was used to alternate the goal node that the A* search query created a path towards. This was relative to the conditions controlling the transition between the states of the bot tank. In doing so, has enabled the bot tank to exhibit several behaviours that places the tank in more advantageous positions, situationally. Furthermore, the turret of the bot tank has been configured to aim towards its relative objective (that's closest) and search around itself; this is dependent on its weapon state. As previously mentioned, we had progressively tested and bug-fixed the program, this had consumed a significant amount of time throughout the program's development, especially as we were unable to alter any of the existing functionality, we could only add or replace functionality where eligible.

However, summarising our program functionality, we were able to implement all our targeted pathing techniques, which were the A* search and DFS methods and additionally were able to implement the BFS algorithm. From which, we have concluded that the A* search query calculates the shortest path to the goal node from its use of a heuristic function, when instructed by the FSM's, instigating it to be the most appropriate algorithm for the bot tank. Opposingly, BFS reveals to calculate paths that are not entirely efficient for speed, as the paths calculated were not always the shortest possible path to each goal node. However, it posed a similar degree of accuracy to A* search than DFS. Of which, DFS calculated paths that were unnecessarily longer in terms of the number of nodes traversed, which made the bot tank much easier to defeat; the bot tank took longer to discover objectives to destruct, and the paths calculated after entering new behavioural states often caused the tank to rotate around in the opposite direction, this occurred when tank has partially destroyed a player base.

Testing Regime

Throughout our project, program code has been progressively tested in the form of test-driven development, having been conducted in such way, has ensured the discovery and further removal of any found bugs; we have used a series of tests (see below) to accomplish this. In accordance to the specification, we wanted to provide program functionality at an advanced level where no "flaws" or "significant flaws" were apparent and believe that the continuity of Smoke, Blackbox, Unit and performance profile testing has achieved this.

In the acknowledgement of testing, as a group we had recognised testing to be significant in the attempts of achieving proper program functionality; a successful program build. This ideal was based upon our other concurrent module 'Advanced Object-Orientated Programming in C++', from which we adopted the skills used to conduct and construct the tests, for this regime. Thereby the deliverable program code, should show to be error-free and be playable from a performance perspective. Which in context of the player tank, will present a stable and proficient opponent.

Testing [**Smoke testing**]

Given our understanding of the types of program tests, we had purposed Smoke testing to determine 'window' controlled events alike the opening, closing and updating state of the program window; simply Smoke testing has been used to demonstrate the programs working, upon program execution (compilation), during program runtime and at the point of program termination. We have nominated this test method to investigate whether any of the functionality additional to the pre-existing functionality provided, has encouraged the program to encounter errors or perform differently to our expectations during these events.

Case	Summary	Process	Actual result(s)	Expected result(s)	Passed?
1	Window opens upon being compiled and executed.	Build the project solution and execute (CTRL + F5).	Window opens upon program execution. No errors.	Render window opens error-free, continues to execute program.	🟩
2	Window renders all object; program is in playable state.	Wait for console window and render window to appear from program execution, observe for images and colouring drawn to window.	All objects are drawn to the render window.	All objects drawn to render window.	🟩
3	Debug draw mode can be toggled during program runtime.	Program processes key pressed 'TAB', revealing all objects in render window space.	Debug draw mode can be toggle continuously during program runtime.	Debug draw mode can be toggled throughout program runtime.	🟩
4	Window processes changes within objects (updates) during runtime.	Tank objects move in the relative window space upon processing keyboard input or pathing technique. Tank objects rotation and position updates (visual changes since execution).	Tank objects update continually, relative to keyboard input and pathing technique.	Tank objects position and rotation changes, accordingly to key input or pathing technique.	🟩
5	Window renders game over state with text before program end.	All enemy bases destroyed, or all players' ammunition is depleted, text object overlays existing objects in render window.	Text object overlays render window when conditions are met.	Text object overlays all object existing in current render window view.	🟩
6	Window closes upon being exited/ game completion.	Navigate to close window icon in corner of render window (ESC).	Window closes upon termination. No errors.	Window closes error-free, returning to application code.	🟩

From our findings, our program functionality has maintained the working order of the program in context of all the window events specified. The program is proven and also experienced to have been error-free from any unexpected program crashing before, during and ending program runtime. Conclusively all test cases were successful.

Testing [**Blackbox testing**]

In further focus of testing, we had also conducted a series of Blackbox testing, for which we intended to test for the functionality and behaviours of the bot tank. And so, we led investigations into the relation between the Finite State Machine states, the bot tanks sensing capabilities as well as the functionality achieved within each of its pathing techniques. To note, investigations were conducted progressively, and were recorded within our 'Discord' group conversation; whereby Joseph (Tester) and Alfie (Programmer), communicatively updated and overcame the undesired or lacking behaviours of the bot tank.

Meanwhile, all the investigative factors mentioned within the progressive testing, are components which determine the bot tanks behaviour(s) overtime; exhibiting desirable behaviours in the programs deliverable state was the objective of the project. And so, we had nominated Blackbox testing, as a method to investigate whether the bot tank exhibits such desired behaviours and whether it performs well as an opponent (using A* search), in the programs deliverable state.

Case	Summary	Process	Actual result(s)	Expected result(s)	Passed?
1	AI tank searches randomly on the left side of the map for enemies if it has reset or the conditions for higher priority states are not met.	AI tank resets and no higher priority state conditions are met.	AI tank enters the SEARCHING state, turret rotates constantly looking for enemies, A* search query calculates a random path to a random node of the left side of the map and the AI tank follows this path.	AI tank enters the SEARCHING state, turret rotates constantly looking for enemies, A* search query calculates a random path to a random node on the left side of the map and the AI tank follows this path.	🟩
2	AI tank stops, rotates perpendicular to	Enemy base enters AI tank's vision and no	AI tank enters the STOPPED state, turret is aimed at the base and the tank shoots at the base, destroying it. If the AI tank is in a	AI tank enters the STOPPED state, turret is aimed at the base and the tank shoots at the base, destroying it. If the AI tank is in a	🟩

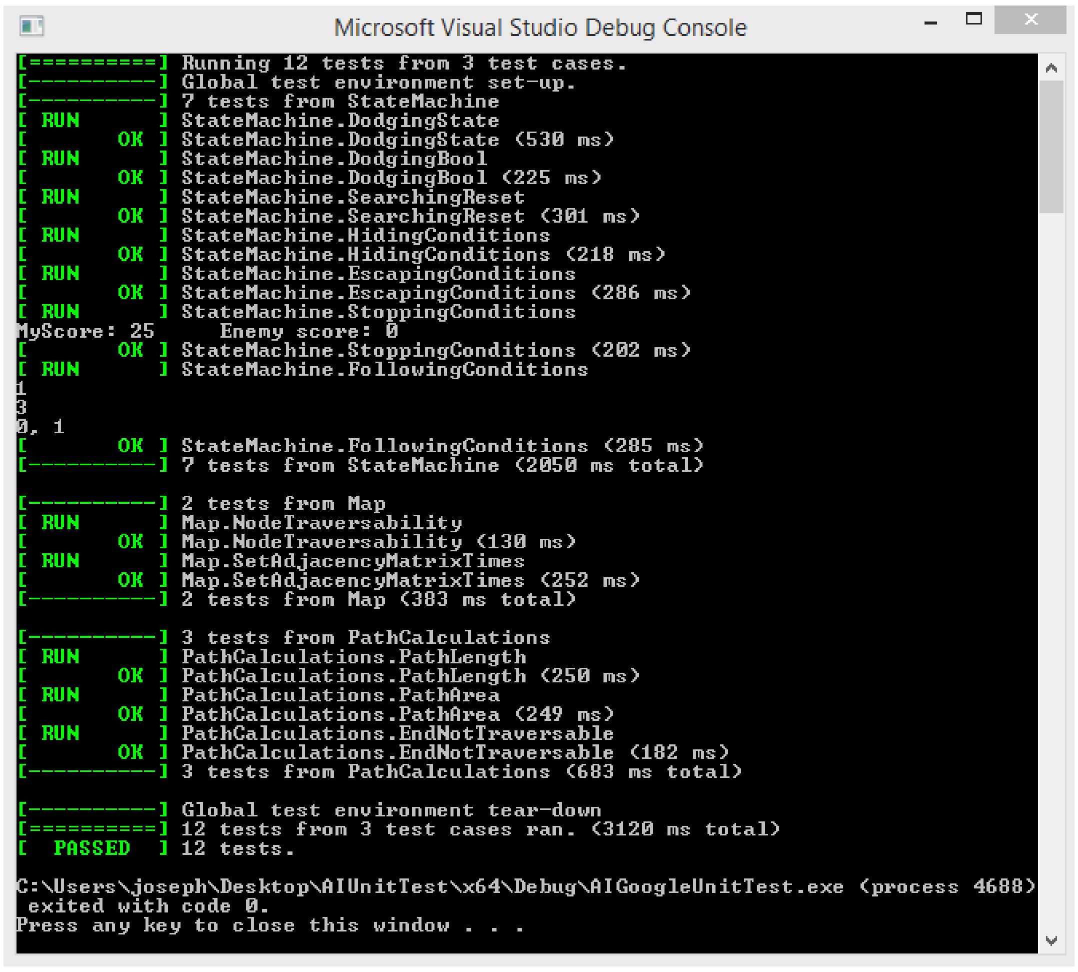
	and then shoots enemy base when it's in view.	higher priority state conditions are met.	higher priority state, the STOPPED state won't be entered.	higher priority state, the STOPPED state won't be entered.	🟩
3	AI tank stops, rotates perpendicular to and then shoots enemy tank when it's in view.	Enemy tank enters AI tank's vision and no higher priority state conditions are met.	AI tank enters the STOPPED state, turret is aimed at the enemy tank, tank rotates perpendicular from the enemy tank and the AI tank shoots at the enemy tank, destroying it. A* search query calculates a path to the enemy object's position, which will be used for the FOLLOWING state. If the AI tank is in a higher priority state, the STOPPED state won't be entered.	AI tank enters the STOPPED state, turret is aimed at the enemy tank, tank rotates perpendicular from the enemy tank and the AI tank shoots at the enemy tank, destroying it. A* search query calculates a path to the enemy object's position, which will be used for the FOLLOWING state. If the AI tank is in a higher priority state, the STOPPED state won't be entered.	🟩
4	AI tank dodges enemy projectile if in its path and it's in view.	Enemy projectile enters AI tank's vision, and the nodes containing the bounding boxes for the path of the projectile intersects with the AI tank's node.	AI tank enters the DODGING state. Tank moves forward until it no longer sees the projectile.	AI tank enters the DODGING state. Tank moves forward until it no longer sees the projectile.	🟩
5	AI tank attempts to hide in the corner of the map when its ammunition is depleted.	AI tank's ammo is depleted, and no higher priority state conditions are met.	AI tank enters the HIDING state, A* search query calculates a path to the closest cornering node in the map, and the tank follows this path, stopping once it reaches its goal node. If the AI tank is in a higher priority state, the HIDING state won't be entered.	AI tank enters the HIDING state, A* search query calculates a path to the closest cornering node in the map, and the tank follows this path, stopping once it reaches its goal node. If the AI tank is in a higher priority state, the HIDING state won't be entered.	🟩
6	AI tank attempts to escape from an enemy tank when one is in view and its ammunition is depleted.	AI tank's ammo is depleted, and enemy tank enters AI tank's vision.	AI tank enters the ESCAPING state, A* search query calculates a path to an adjacent cornering node, anticlockwise to its relative current cornering node of the map, and the tank follows this path. If the AI tank reaches the goal node it is escaping to, the tank will leave the ESCAPING state and enter the HIDING state.	AI tank enters the ESCAPING state, A* search query calculates a path to an adjacent cornering node, anticlockwise to its relative current cornering node of the map, and the tank follows this path. If the AI tank reaches the goal node it is escaping to, the tank will leave the ESCAPING state and enter the HIDING state.	🟩
7	AI tank moves towards an enemy object that has left its vision.	AI tank is in the STOPPING state, no enemy object is in the AI tank's vision and no higher priority state conditions are met.	AI tank enters the FOLLOWING state, causing the AI tank to follow the path calculated when in the STOPPING state. If the AI tank is in a higher priority state, the FOLLOWING state won't be entered.	AI tank enters the FOLLOWING state, causing the AI tank to follow the path calculated when in the STOPPING state. If the AI tank is in a higher priority state, the FOLLOWING state won't be entered.	🟩
8	AI tank follows a newly calculated path if it gets stuck on a base or another tank.	AI tank has not moved for several frames when it is supposed to be moving.	AI tank enters the STUCK state, causing the AI tank to calculate a new path, it then enters another state depending on conditions met and then follows the newly calculated path that will go around the object causing the AI tank to be stuck.	AI tank enters the STUCK state, causing the AI tank to calculate a new path, it then enters another state depending on conditions met and then follows the newly calculated path that will go around the object causing the AI tank to be stuck.	🟩
9	AI tank aims at the closest enemy object, prioritising enemy tanks over bases, when one is in view.	AI tank is in the STOPPING state.	AI tank enters the AIMING weapon state, calculating which enemy object is the closest, prioritising enemy tanks. AI tank then aims its turret towards this enemy object.	AI tank enters the AIMING weapon state, calculating which enemy object is the closest, prioritising enemy tanks. AI tank then aims its turret towards this enemy object.	🟩
10	AI tank searches around its body using its turret.	AI tank is in either of the SEARCHING, HIDING, DODGING, FOLLOWING or ESCAPING state.	AI tank enters the SEARCHINGAIM weapon state, causing the turret to rotate constantly in one direction. If the AI tank is in a higher priority weapon state, the SEARCHINGAIM state won't be entered.	AI tank enters the SEARCHINGAIM weapon state, causing the turret to rotate constantly in one direction. If the AI tank is in a higher priority weapon state, the SEARCHINGAIM state won't be entered.	🟩

Summarising our findings, the bot tank conforms to all the desired behaviours, as shown to all 'pass'. Whereby the tank is fully capable of searching for enemy bases and the playing tank (seek), able to destroy the player tank as well as it's bases upon seeing them, to dodge projectiles casted from the player tank (avoid), to hide momentarily when its ammunition has depleted (flee), to remember and attempt to follow the player tank and or it's last known location (follow), to detach itself from player bases (when stuck) enabling the tank to move once more, to prioritise the destruction of the player tank or base relative to the distance between them and it, and to also identify all of such objectives, via the use of the Object-Bounding Box (OBB) assigned to the tanks, nodes and casted projectiles . All of which behaviours, are correctly transitioned to and from in accordance to their priority arrangement. Therefore, from our observations (Blackbox testing), the bot tank successfully exhibits sensing capabilities at a sufficient level via the use of Finite State Machines and further demonstrates error-free pathing from the use of the A* search technique. Overall, the bot tank is successfully presented as an opponent to the player.

Testing **[Unit testing]**

Additionally, unit testing has been used to simplify debugging processes throughout the development of the program. Whereby bugs were easily identifiable from the test cases that have been setup to be refactored, which were utilised to eliminate all possible errors, in attempt to establish a solution quicker. More so, unit testing provides the capability of testing aspects of the program within isolation and is convenient to use regarding its small and simple code requirement to create test cases. All of which was considered as a suitable method for testing the bot tanks functionality, alike Blackbox testing, however, instead is within focus of the accuracy of the tank's behaviours. We measured this accuracy using Boolean logic and numeric driven test cases.

A priming test case is that of the bot tanks 'dodging' behavioural state, which is alternated based upon its detection of Object-Bounding Box (OBB) intersection, between the bot tank and an oncoming projectile; projectiles are casted from the player tank. The tanks sensing capability in this given example needs to be measured, ensuring that the bot tank moves away from a projectile when it enters 'DodgingState'; the collision between their Object-Bounding Boxes (see below). Hence its nomination as a test method.



```
[==========] Running 12 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 7 tests from StateMachine
[ RUN      ] StateMachine.DodgingState
[       OK ] StateMachine.DodgingState (530 ms)
[ RUN      ] StateMachine.DodgingBool
[       OK ] StateMachine.DodgingBool (225 ms)
[ RUN      ] StateMachine.SearchingReset
[       OK ] StateMachine.SearchingReset (301 ms)
[ RUN      ] StateMachine.HidingConditions
[       OK ] StateMachine.HidingConditions (218 ms)
[ RUN      ] StateMachine.EscapingConditions
[       OK ] StateMachine.EscapingConditions (286 ms)
[ RUN      ] StateMachine.StoppingConditions
MyScore: 25      Enemy score: 0
[       OK ] StateMachine.StoppingConditions (202 ms)
[ RUN      ] StateMachine.FollowingConditions
1
3
0, 1
[       OK ] StateMachine.FollowingConditions (285 ms)
[----------] 7 tests from StateMachine (2050 ms total)

[----------] 2 tests from Map
[ RUN      ] Map.NodeTraversability
[       OK ] Map.NodeTraversability (130 ms)
[ RUN      ] Map.SetAdjacencyMatrixTimes
[       OK ] Map.SetAdjacencyMatrixTimes (252 ms)
[----------] 2 tests from Map (383 ms total)

[----------] 3 tests from PathCalculations
[ RUN      ] PathCalculations.PathLength
[       OK ] PathCalculations.PathLength (250 ms)
[ RUN      ] PathCalculations.PathArea
[       OK ] PathCalculations.PathArea (249 ms)
[ RUN      ] PathCalculations.EndNotTraversable
[       OK ] PathCalculations.EndNotTraversable (182 ms)
[----------] 3 tests from PathCalculations (683 ms total)

[----------] Global test environment tear-down
[==========] 12 tests from 3 test cases ran. (3120 ms total)
[  PASSED  ] 12 tests.

C:\Users\joseph\Desktop\AIUnitTest\x64\Debug\AIGoogleUnitTest.exe (process 4688)
 exited with code 0.
Press any key to close this window . . .
```

Providing the test case results, it is inevitable that the bot tank demonstrates functionality that is accurate as we had configured for. In which, all the following test cases ‘passed’ and further identify that the tank is a proficient opponent, from which all desired behaviours are exhibited. Below outlines each test cases purpose and its simulated conditions.

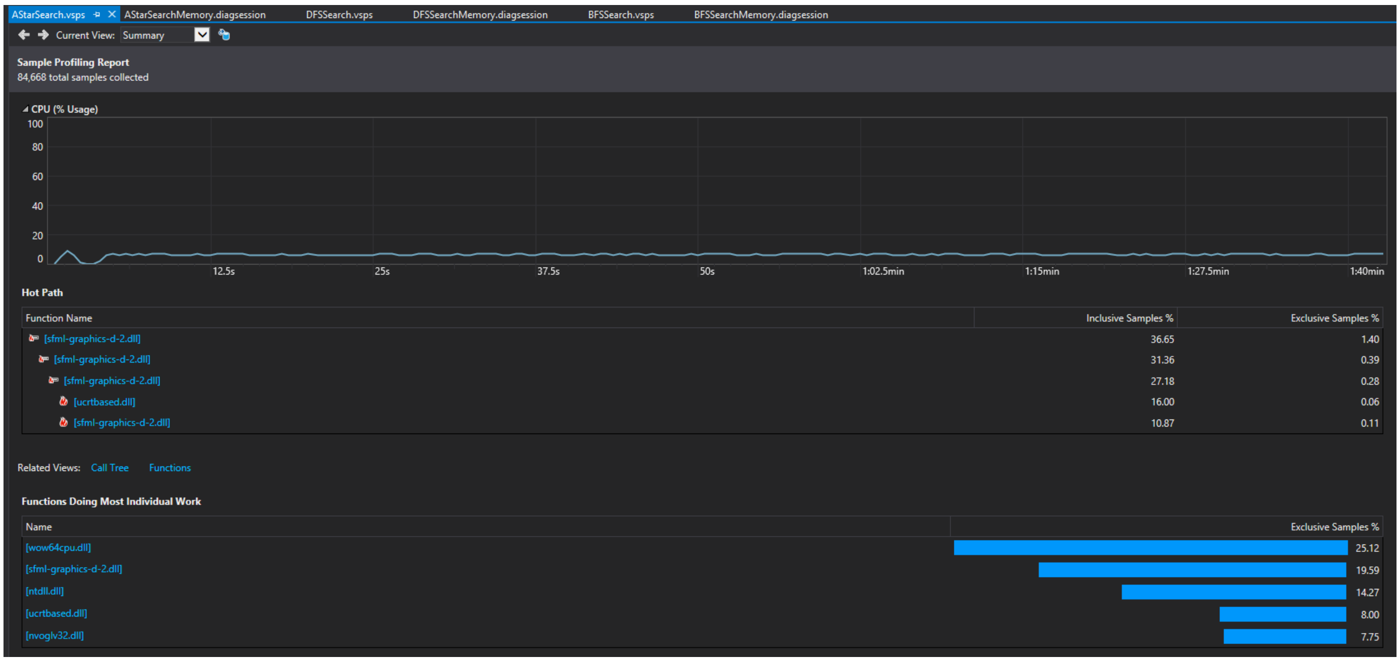
Test case	Test purpose
[StateMachine] DodgingState	Simulates a player projectile being projected near the bot tank, tests if it causes the bot tank to enter the DODGING state correctly, moving the tank out of range.
[StateMachine] DodgingBool	Tests if Boolean variable bDodging is set correctly when a projectile is seen, this makes sure the bot tank keeps dodging as long as it can see a player projectile after it was in its path.
[StateMachine] SearchingReset	Simulates the tank respawning, tests if its state is reset to SEARCHING correctly, the tank moves in search of the nearest objective.
[StateMachine] HidingConditions	Simulates when the tanks ammunition depletes, tests if the state is set to HIDING correctly, the tank moves to a corner of the map which the player tank is adjacent to.
[StateMachine] EscapingConditions	Simulates when the tanks ammunition depletes and sees the player tank, tests if the state is set to ESCAPING correctly, the tank continues to move to corners of the map adjacent to the player tank.
[StateMachine] StoppingConditions	Simulates when the tank sees the player tank, tests if the state is set to STOPPING correctly, the tanks position does not change.
[StateMachine] FollowingConditions	Simulates when the tank sees the player tank and it leaves its view for more than twenty (20) frames, tests if the state is set to FOLLOWING correctly, the tank moves towards the player tanks last seen location.
[Map] NodeTraversability	Tests the map.traversable() method, ensuring that it returns true for traversable objects (UNKNOWN, PLAYERSHELL) and returns false for non-traversable objects (OWNBASE, PLAYERBASE, PLAYERTANK).
[Map] SetAdjacencyMatrixTimes	Tests the setAreaTraversable() method against the setMapTraversable method() for speed, ensuring that the setAreaTraversable() function is faster, so that it can execute whilst playing.
[PathCalculations] PathLength	Tests each pathfinding algorithm's path length, ensuring that the path calculated by the A* search technique, is shorter than the path calculated by DFS, but is the same length of path as calculated by BFS.
[PathCalculations] PathArea	Tests each pathfinding method (A* search, DFS and BFS), ensuring A* search query calculates the most direct and efficient path of the three, and the calculated path therefore utilises the least area space.
[PathCalculations] EndNotTraversable	Tests if the program correctly prevents a goal node from being in a non-traversable node, preventing the program from unexpected crashing.

Testing [**Performance profiling test**]

Regarding performance profiling, to ensure the code functionality being implemented was adequately playable, we had additionally undergone a series of performance profiling tests over the duration of the program development. Specifically, both Central Processing Unit (CPU) and Random-Access Memory (RAM) profile tests were conducted, both of which purposed to indicate the consumption of computational processing and memory throughout the cycle of execution. Based upon the results gathered throughout the project, we were further able to optimise code efficiency; one example is when changing the adjacency matrix to make a new path, it checks only for nodes within a specific proximity of the path start and end nodes. In result, is less time and processing expensive in comparison to analysing the entire map, for every iteration of a new path made. Notably, the improved adjacency matrix calculations were only applicable by the A* search algorithm, as it computes the most direct path to the goal node and will not enable the traversing of the bot tank in any nodes exceeding the calculated path area.

Using performance profiling testing has enabled us collaboratively to understand the difference that numerous path finding techniques has, on relative computational performance. It can further be utilised as a measure of speed, contrasted to the accuracy of pathing each technique provides; hence why it was another nominated method of testing.

[**CPU usage**] A* search:



Sample Profiling Report
84,668 total samples collected

Hot Path

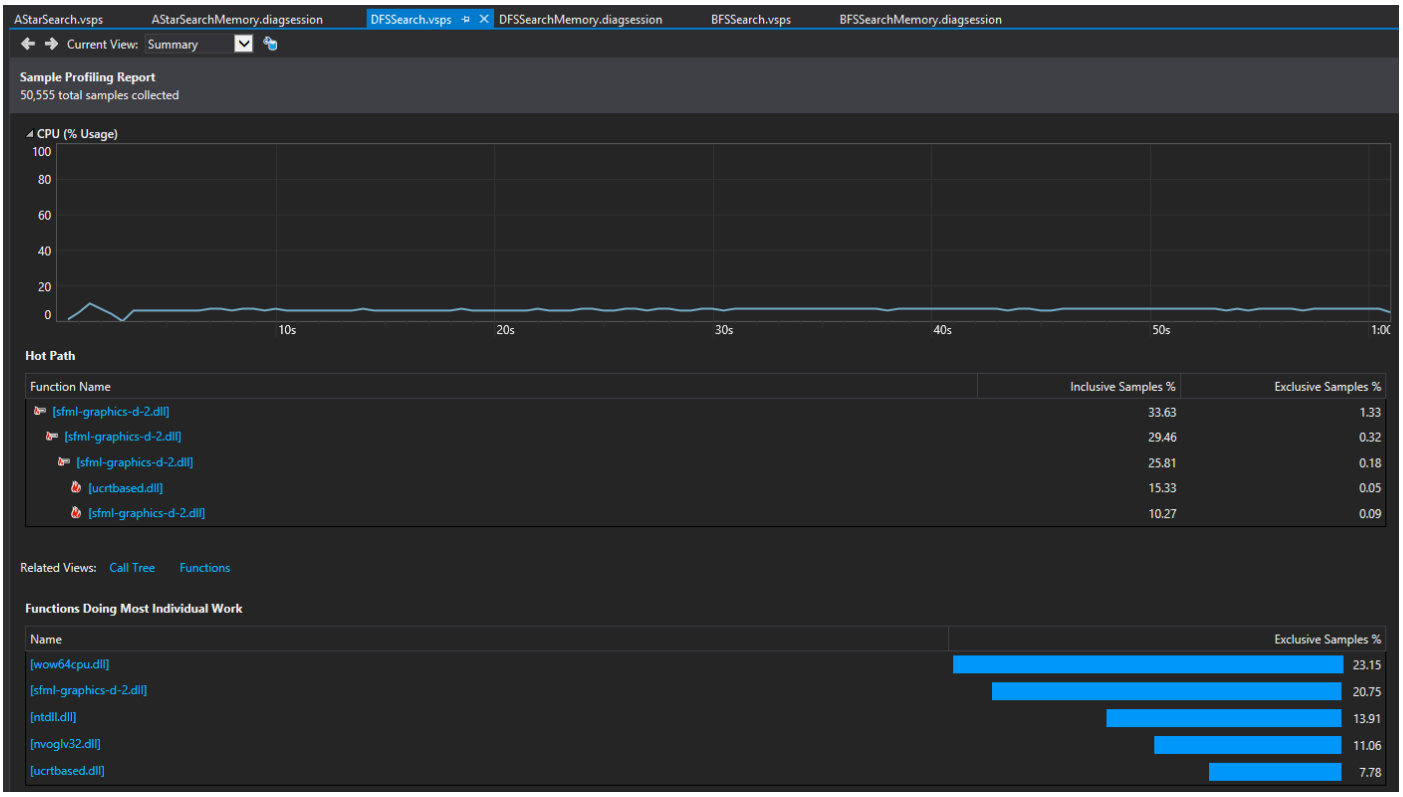
Function Name	Inclusive Samples %	Exclusive Samples %
[sfml-graphics-d-2.dll]	36.65	1.40
[sfml-graphics-d-2.dll]	31.36	0.39
[sfml-graphics-d-2.dll]	27.18	0.28
[ucrtbased.dll]	16.00	0.06
[sfml-graphics-d-2.dll]	10.87	0.11

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[wow64cpu.dll]	25.12
[sfml-graphics-d-2.dll]	19.59
[ntdll.dll]	14.27
[ucrtbased.dll]	8.00
[nvoglv32.dll]	7.75

[**CPU usage**] Depth-first Search:



Sample Profiling Report
50,555 total samples collected

Hot Path

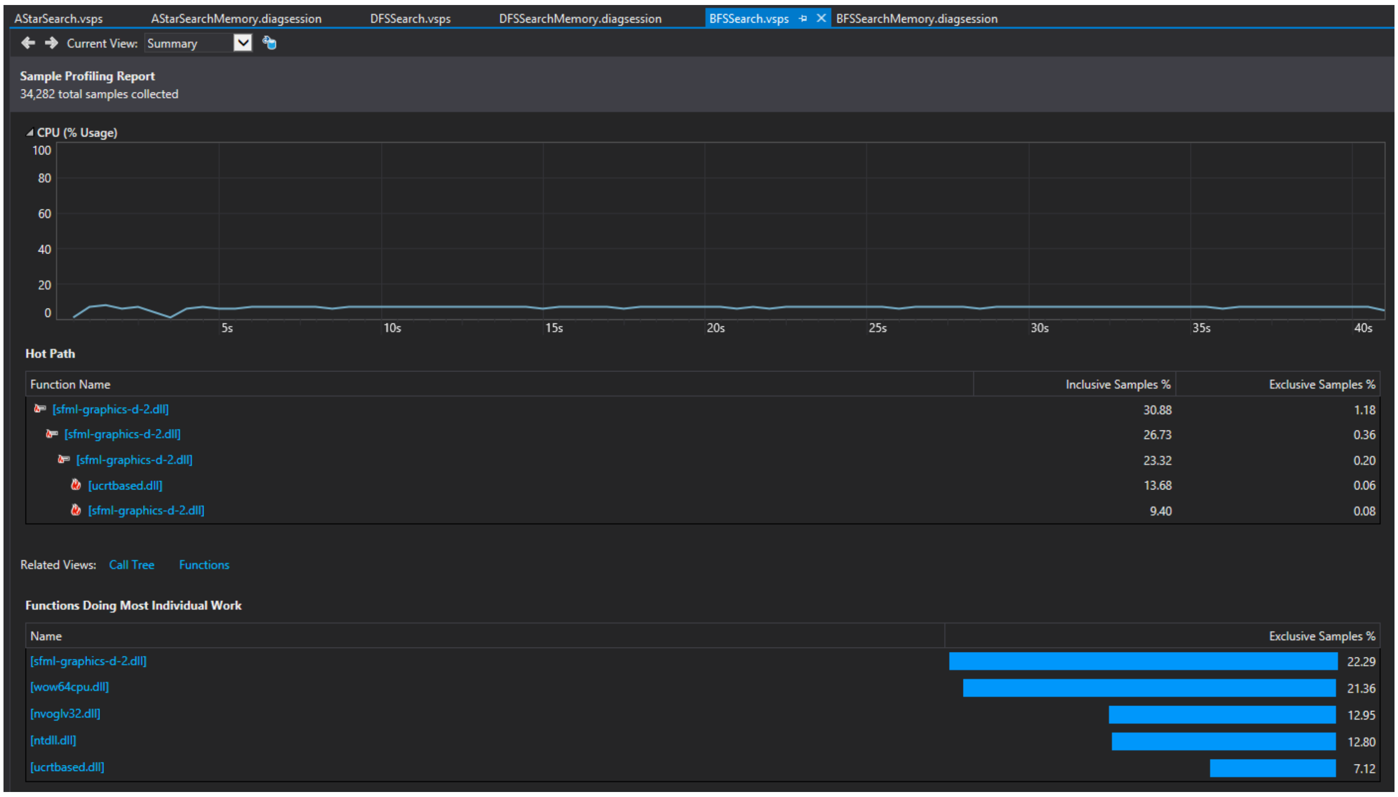
Function Name	Inclusive Samples %	Exclusive Samples %
[sfml-graphics-d-2.dll]	33.63	1.33
[sfml-graphics-d-2.dll]	29.46	0.32
[sfml-graphics-d-2.dll]	25.81	0.18
[ucrtbased.dll]	15.33	0.05
[sfml-graphics-d-2.dll]	10.27	0.09

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[wow64cpu.dll]	23.15
[sfml-graphics-d-2.dll]	20.75
[ntdll.dll]	13.91
[nvoglv32.dll]	11.06
[ucrtbased.dll]	7.78

[**CPU usage**] Breadth-first Search:


AStarSearch.vsps
AStarSearchMemory.diagsession
DFSSearch.vsps
DFSSearchMemory.diagsession
BFSSearch.vsps
BFSSearchMemory.diagsession
Current View: Summary
CPU (% Usage)
100
80
60
40
20
0
5s
10s
15s
20s
25s
30s
35s
40s
Related Views: Call Tree Functions

Sample Profiling Report
34,282 total samples collected

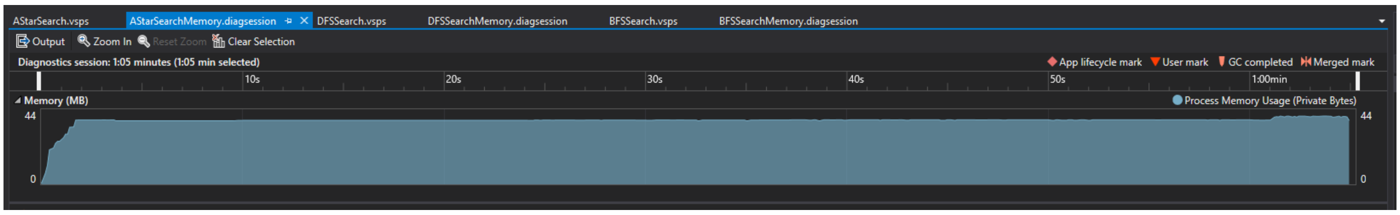
Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
[sfml-graphics-d-2.dll]	30.88	1.18
[sfml-graphics-d-2.dll]	26.73	0.36
[sfml-graphics-d-2.dll]	23.32	0.20
[ucrtbased.dll]	13.68	0.06
[sfml-graphics-d-2.dll]	9.40	0.08

Functions Doing Most Individual Work

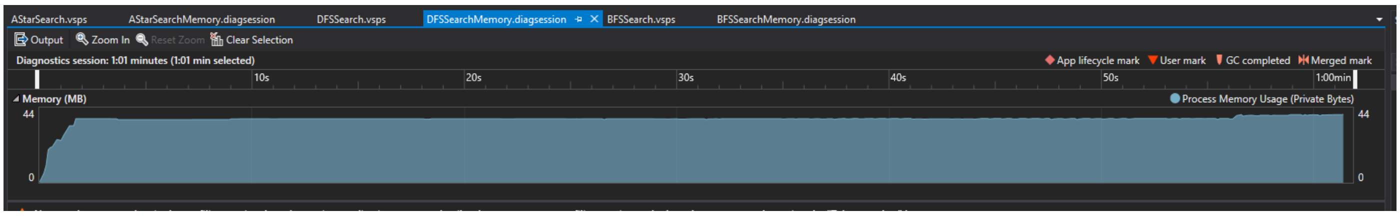
Name	Exclusive Samples %
[sfml-graphics-d-2.dll]	22.29
[wow64cpu.dll]	21.36
[nvoglv32.dll]	12.95
[ntdll.dll]	12.80
[ucrtbased.dll]	7.12

[**RAM usage**] A* search:



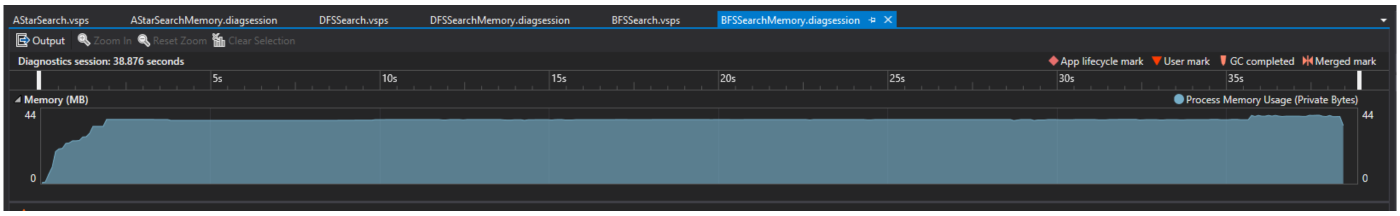
Diagnostics session: 1:05 minutes (1:05 min selected)

[**RAM usage**] Depth-first Search;



Diagnostics session: 1:01 minutes (1:01 min selected)

[**RAM usage**] Breadth-first Search:



Diagnostics session: 38.876 seconds

Without doubt A* search is more computationally stressing in comparison to both Depth-first Search (DFS) and Breadth-first Search (BFS) throughout program runtime. Through which A* search shows more inconsistency in the usage of computational processing (CPU) in comparison to both DFS and BFS techniques. The CPU usage when A* search is equipped fluctuates between 6-7% usage per averaging 1.5 second interval; DFS and BFS both share a similar fluctuation rate, in which DFS fluctuates CPU usage between 6-7% per averaging 3-4 second intervals, whereas BFS fluctuates CPU usage between 6-7% also, per averaging 4-5 second intervals. Relative to the most efficient utilisation of CPU, the ranking of each technique from most to least is: BFS, DFS and then A* search. Inevitably, Breadth-first Search provides the most stable processing performance, this is assumed to be because of BFS's ability to quickly and inexpensively calculate paths in comparison.

Moreover, regarding memory, all three path finding techniques have used a similar amount of Random-Access Memory (RAM) over the course of program runtime. In which, all the path finding methods have utilised 41MB of computational memory on average, and at most had used 44MB (due to the overlaying text loading into memory). The amount of memory occupied was as expected and purposed as an indicator of the objects and resources loaded into memory during runtime. As none of the functionality used to implement each technique, involved the loading of files and objects into memory during runtime, the memory usage remained identical in comparison.

Overall, the performance profiling results reveal an insignificant difference between the path finding methods, which therefore suggests that either of the A* search or Breadth-first Search techniques are better suited for the bot tank. Both techniques calculate paths that would lead the tank to reach a goal node quicker, this is in comparison to Depth-first Search and enables a faster orientation of gameplay to the player. Clearly, the path finding techniques within this given scale of program, have no noticeable influence on computational performance in real-time. And as this testing method concludes, A* search is once again proven to be a suitable path finding technique for the bot tank.