Pathfinding Pac-Man: Shortest Path Optimisation Using Search Algorithms

Jack Wilson Moorin
Faculty of Computing, Engineering and Media
De Montfort University
Leicester, United Kingdom
P17190172

Adam Leonard Hubble
Faculty of Computing, Engineering and Media
De Montfort University
Leicester, United Kingdom
P17175774

Arpit Sharma
Faculty of Computing, Engineering and Media
De Montfort University
Leicester, United Kingdom
P2613237

***Abstract* — In graph theory, a shortest-path algorithm establishes the path of nodes that collectively, constitute to the minimal cost between any two vertices in any given graph. Pathfinding for intelligent agents within the domain of video game production has been investigated for much time now and is arguably the most problematic, artificial intelligence (AI) behavioral discipline to address absolute optimality for; acquiring budget-friendly computation and optimal path construction is provably challenging and desired for the evolving complexities of modernized video games. This paper anticipates the shortest-path problem relative to optimization, in exploration of the application of optimal variates of search algorithms, used in contemporary works, to address a '*Rat in a Maze*' orientation of the arcade production, Pac-Man.**

Keywords — graph theory, shortest-path algorithm, minimal cost, graph, pathfinding, intelligent agents, video game, problematic, artificial intelligence, optimality, budget-friendly computation, optimal path construction, search algorithms, Rat in a Maze, Pac-Man

I. Introduction

Computational Intelligence (CI) is the study of the design of intelligent agents, which are systems that within a given environment, exhibit behaviours regarded as being intelligent. Collectively, intelligent agents are generally proposed by the "theory, design, application and development" [1] of biologically and linguistically inspired computational paradigms, that traditionally constitute to the "three main pillars of CI", which are: Neural Networks (NN's), Fuzzy Systems (FLS's) and Evolutionary Computation (EC); however, many of the "nature inspired" computing paradigms have overtime advanced and become more relevant to applications in what is, an evolving field [2]. So much so, the paradigms in the recent era have adopted a proliferate presence within "developing successful intelligent systems, including games and cognitive developmental systems" [1]; considering the recent boom of research into the field of Deep Learning (DL), CI has become a "core method for artificial intelligence" applications, such that some of the most successfully regarded "AI systems are based on CI" principles.

Parallel to the paradigms sponsored by CI, Artificial Intelligence (AI) is also the study of the design of intelligent agents to "mimic the capabilities of the human mind" [3], led by its "pluridisciplinarity" [4], AI facilitates "learning from examples and experience, recognizing objects, understanding and responding to language, making decisions, solving problems" [3] and combinatorically applying said concepts to render computing systems "capable of solving problems that usually require the ability of human beings" [4]. Such problems typically concern a high-time complexity to resolve and natural, or biological-based outlooks, specific to visual recognition and natural language processing (NLP) for example. As the "first large scientific community" [2], many problem domains requiring intelligence to be solved has exposed AI to a range of applications concerning theorem proving, NLP, speech recognition and understanding, image interpretation and vision, robotics, and expert systems [4][5]. Despite the "several attempts to design intelligence with the same kind of flexibility as that of a human", over the fifty-years that AI has been a "defined and active field" [6], none has been met with much success [5]; for which is why the optimality of AI behaviours is currently known problematic to acquire.

In correspondence with the abovementioned quandary, the premise of this paper is to review the problem domain of graph theory, specifically within focus of the shortest-path problem, relevant to approaching path optimisation in unweighted (undirected) graph trees. For which, solutions are proposed to mitigate and address, in compliance with the array of shortest-path or search algorithms used within contemporary applications, to yield provably optimal solutions for space-navigation efficiency; this problem is contextualised for each solutions performance in the classic, arcade video game production: Pac-Man [7].

Conveniently, the paper is componentised into eight sections, one of which being this very introduction. Encompassed by the sections proceeding this passage, are discussions relevant to the contextual nature of the problem domain bespoken and works submitted to the field that are conceptually similar to the work proposed. As well, are the design choices and implementation synopses of the solutions investigated, for the software platform relevant, and lastly, are evaluative and concluding remarks concerning the optimality of the solutions submitted for the domain.

II. Background

A. Problem Definition

With regards to the definition of the problem tasked, the problem can be characterized simply, as the establishment of the "optimal path in a graph from a source vertex to a destination vertex while minimizing total cost" [8], or otherwise, as the extraction of the "shortest route (distance), the fastest speed and the lowest expenditure" [9] of traversal, from the vertices comprising a graph, that are situated between the corresponding points of origin (source) and interest (destination). Formally, the problem is renowned as the Shortest-Path Problem (SPP), exemplified as: "finding the quickest way to get from one

city to another on a road map" [8], which can otherwise be extended to the Travelling Salesperson Problem (TSP) [10], defined as: "finding the route for the shortest tour between cities with the condition that, each city is visited only once and the returning point of the tour will be the exact starting point of the tour". Although, providing the nature and scope of work proposed for the Pac-Man application, TSP cannot be strictly adhered to given the narrow structural arrangement and limited adjacencies of coin collectibles (the cities) occupying (traversable) nodes in the graph of the game; thus, TSP considerations are neglected from the applications state, which subsequently informs that the solutions submitted are strictly shortest-path algorithms.

B. Field Research

Finding the shortest path between a point of origin and a point of interest in a graph is an "important problem" [11], whose solution has many applications. Ranging from car navigations systems [12], traffic simulations [13], transport scheduling [14], internet route planning [15], web searching [16], and vital to our investigation, direction orientated path planning within video games [17]. In the context of video games, an optimal path between "any two locations is the least cost path rather than the shortest path"; given the contrasting example of a horizontal pathway and a pathway mapped to the concavity of a mound that are of equal length (geographical distance), relative to time, the cost incurred from ascending, and descending said mound would inevitably be more expensive than traversing along a horizontal plane, only. Currently within the field and context in focus, solutions submitted for pathfinding either provide a "high speed search by sacrificing accuracy or produce an optimal path but using more time and resources", such that acquiring absolute optimality in path planning remains a contemporary subject of study.

Scientists and mathematicians have long studied the problem domain, that is pathfinding [18]. As a "classical graph theory algorithm problem", the problem was first solved by Dijkstra's algorithm in 1959 [19], which is well-recognised and remains to be one of the best "among various algorithms for the shortest-path problem" in the present day [20]. Evans and Minieka [21] proposed that most path algorithms are bound by three categories: "the single-source shortest-path algorithms, the all-pairs shortest-path algorithms, and the *k*-shortest-path algorithms". Summarily, single-source shortest-path algorithms compute the shortest paths from a specified point of origin (source) to a point of interest (destination), or to all other vertices comprising the graph. Whilst all-pair shortest-path algorithms compute the shortest-paths between every pair of vertices comprising the graph, and *k*-shortest-path algorithms not only compile the optimal path between specified vertices but until k^{th} best paths are computed. From which the pathfinding or planning problem can be characterised as optimally routing through "graphs that contain sets of vertices and edges", representing traversable nodes and their adjacencies.

Relevant to video game productions, pathfinding "generally refers to finding the shortest route between two end points" [22], with the support of shortest-path algorithms, typically of the single-source type. Almost all video games "require pathfinding to make the game more human-like" [23] and such it has become imperative to apply optimised pathfinding algorithms, since there are many "real-time games" being developed nowadays, that require solutions to be able to "solve pathfinding problems on a more complex environment with limited time and resources" [22]. As games are one of the "popular forms of entertainment" [23] for which has extended to digital platforms such as "mobile phones", there has been an "increasing interest" in pathfinding in games, which is why the previous "two decades have seen a growing trend" towards applied AI in video games. Such that pathfinding is one of the most prevalent "applications of game research among AI techniques" at present, where "dozens of revised algorithms have been introduced successfully" [22] to the field. Undoubtedly, games can be much more fun and engaging "when the agents in the game are smart enough to take the shortest path" [23], which arises the most "common" pathfinding problem in video games, an agent's movement. Given that video games are recognised as "excellent experiment" features for pathfinding research, we propose the classic arcade production: Pac-Man, as our catalyst for identifying the optimal shortest-path algorithm, via comparative analysis; the algorithms elected for the investigation proposed, express relevance to the recommendations of contemporary works existing in the field already. Reasoning for the production's nomination derives from its many adaptations to related works already existing in the domain [24][25][26].

III. Literature Review

A. Common

The role of pathfinding is the most 'visible' [27] problem in the domain of AI and video games. Intelligent pathfinding is relevant for all games in which the movement of the object under consideration, is calculated by the computer, instead of being player-controlled. Objects can identify to be anything in a game, such as a person, vehicle, or a miscellaneous item, which for the investigation proposed, is Pac-Man, an agent who traverses through a two-dimensional grid upon finding the shortest path to the closest coin collectible, in absence of the enemies or ghosts featured in the original orientation of the game.

For discovering the shortest path from the Pac-Man avatar to the target ahead of its movement using the 'plan before move' [27] strategy, a literature review is purposed for identifying a series of optimal, shortest-path algorithms specific to the application of unweighted graphs, within a two-dimensional space (matrix), with known points of origin (source) and interest (destination). This survey was conducted over the course of several weeks, to compile promising solutions for the problem domain bespoken.

Algorithm Completeness – the ability of the algorithm, representing the probability at which it is "guaranteed to return a solution" [28], or in other words, its competency to unfailingly find the target every time one is assigned, and subsequently generate a fully-connected path. In video game productions, algorithmic completeness is a vital characteristic, as without it, a game would promote behavioural defects of undesirable natures; perhaps leading to hardware or software malfunctioning, that resultingly degrades a player's experience and their continuing engagements with it.

Space complexity – the number of total cells or nodes visited by the algorithm for path planning, as a function of

depth, where the branches (adjacencies) from each node are represented by cells within a grid formation; this is dependent upon the number of different cells that Pac-Man "can reach in any sequence of actions" [29]. According to the structural arrangement of the games environment, each cell in the grid can have a maximum of four neighbouring cells. Therefore, the maximum number of branches to each node in the generated graph, cannot exceed four. As for the dimensionality of the grid, the environment of the game is composited up of thirty rows, which is greater than the number of featured columns, that supports twenty-eight; the maximum depth of the graph will always be less than thirty nodes in length, given this layout. But, the space complexity is still high enough to be taken into consideration. An algorithm offering less space complexity or otherwise, considering fewer nodes for path planning, is preferred.

Time complexity – similarly, the total amount of time consumed by the algorithm to traverse the graph, is a function of the number of branches expanding each node in the graph, as well as the graphs depth; thus revealing the relative time taken by the algorithm to reach the target node specified. Therefore, the algorithm operating with lesser time complexity is favourable.

B. Adam Hubble

Since the advent of the computer, researchers have invested more of their attention and time into the "optimal path selection problem" [30], otherwise known to be the shortest-path problem, which as one of the "well-studied topics in computer science" [31], specifically in graph theory, has been diversified across many fields concerning video games, robotics, route planning, traffic control and routing protocols [32], to name a few contemporary examples. Consequently, there has been a recent "surge of research in shortest-path algorithms due to said problems numerous and diverse applications" [31], in attempt of alleviating sub-optimal approaches to resolving the problem. Generally, the shortest-path problem is represented via graphs, for which a graph can be characterised as a "set of vertices and edges", such that the edges connect to the vertices to form a graph tree; where along the edges of said graphs, it is possible to traverse from one vertex to another, in attempt to route through the graph's topology. Relative to the "lengths" of the edges employed by a given graph, as otherwise recognised as "weights", the lengths of the edges are representative of the "cost between two vertices in a graph", normally used for "calculating the shortest path from one point to another point". Typically, pathfinding anticipates the minimisation of the cost or "path length from start to goal" [32] destinations, for which there exists "many algorithms" [31] to address, where each is purposed for compiling the "shortest route (distance), at the fastest speed and with the lowest expenditure" [9] of traversal. Translated to video game applications, path planning presents to be a significant behaviour of a non-player controlled (NPC) character or agent, in finding the "shortest, fastest, and cheapest way possible to navigate from one place to another within the game environment" [33]; environments within such games are commonly represented by grid-based maps [26].

Recognised as one of the "best known and widely used pathfinding algorithms" [32], A* (pronounced a-star), proposed by Hart, Nilsson, and Raphael in 1967 [34], exists as an extension to Dijkstra's algorithm [19], by adding a heuristic value that "estimates the distance from the start node to the goal node" concerned in a path. Undeniably, various search algorithms were published prior to its emergence, one of which being the Depth-First Search (DFS) algorithm [34]. But since its "success" in the field of pathfinding problems and specifically game AI [22], it has been known for "many researchers to have focused on variants of A* algorithm", such that many "revised pathfinding algorithms" have been introduced; the success of the algorithm is acknowledged to be governed by its "more convenient capabilities compared to others" [23], that fundamentally enable the algorithm to establish a "feasible path between two points in a short period of time" [34]. Firstly, A* is "guaranteed to find a path from the start to the goal if there exists a path" [22], and it is considered "optimal" if the estimated cost calculated is an "admissible heuristic", meaning that the heuristic cost $h(n)$ is "always less than or equal to the actual cheapest path cost from n to the goal", otherwise known as the geographical distance $g(n)$. Lastly, A* is recognised to make the "most efficient use of the heuristic" calculated, where no other search algorithm uses the same heuristic function to find an "optimal path that examines fewer nodes than A*". Hence A* algorithm is regarded as the "most advanced" [25] and "provably optimal" [22] algorithm available for the interest of the problem domain being explored.

Extending the study upon the A* algorithm, a heuristic approach to path planning infers that rather than a "exhaustive expansion" of a graph, like that of Dijkstra's algorithm, which states that "all possible states must be examined", only the states or nodes comprising the graph that "look like better options, are examined"; thus, the advantage of A* is that it "does not need to traverse all nodes, but instead proceeds in the direction of the desired road (the target node that needs to be experienced)" [30]. This aligns adequately with the progressive absence of coin collectibles in the grid of the game's environment, that Pac-Man would 'eat' overtime, from recursively navigating to and from their corresponding nodes; the heuristic function used by A* is purposed to estimate the "cost from any nodes on the graph to the desired destination" [22], only. By introducing the heuristic approach, A* algorithm "improves the computational efficiency" of Dijkstra's algorithm, "significantly". Notably, if the estimated cost returned by the heuristic function is known to be exactly equal to the actual cost, only the nodes comprising the best path are selected and thus, no other node in the grid is expanded. Implying that a "good" heuristic that can accurately estimate the cost to a specified goal node, may allow the algorithm to operate "much quicker". Although, using a heuristic that "overestimates" the actual cost to the goal node, typically results in a "faster search with a reasonable path" being generated nonetheless, as the search "pushes hard" on the closest nodes to the goal; this resorts to fewer nodes being explored, when compared to "non-overestimation heuristic approaches".

However, it must be acknowledged that the A* algorithm "often finds the optimal path needed because the heuristic function is not suitable"; therefore, meaning that the "success rate of using the A* algorithm to select the optimal path is not very high" [30]. Although A* algorithm has the "shortest time in theory" [35] and is arguably about as "good as a search algorithm as you can find so far" [22], it also sponsors a series of performance defects that must be

addressed. Importantly, A* algorithms spatial growth is "exponential" [35], where the time complexity of the algorithm is $O(b^d)$ and space complexity is $O(b^d)$ [36], such that when searching in larger game environments, A* becomes "expensive in terms of execution times when the number of nodes in the map increases" [34]; resulting in the search process and games continuity alike, advancing "more and more slowly" [35]. Also, maintaining focus upon the scenario presented, A* algorithm requires a "huge amount of memory to track the progress of each search" [22], that in terms of memory allocation, can "rapidly change to the environment" [34]; consequently causing "excessive" memory leaks, before "producing the solution". Due to these limitations, variants of the A* algorithm have later since been "introduced to solve the pathfinding problem", hence the birth of the popular: Anytime Repairing A* (ARA*) [37], Theta*, D*, D* Lite, Field D*, AD* and Iterative-Deepening A* [36] search algorithms, that each attempt to "reduce space requirements in A*" [22] or to hasten calculations in "dynamic environments" [32]. However, given the domain of Pac-Man and the productions relatively small environment, space complexity and the resultant time complexity is not of much concern for the A* algorithm, in which "obstacle density" is sought to be the only contributing factor.

Providing its popularity in the game industry, Cui and Shi [22] regard A* algorithm as the "provably optimal solution for pathfinding" in modern video games, where despite being easy to understand, its implementation in a "real computer game is non-trivial". The authors reveal the algorithms relevance to popular, contemporary production titles, like that of Civilisation V and Counter-Strike, alongside the "classic real-time strategy game", Age of Empires. Concluding with the A* algorithm being performatively optimal for the application of video games, for which is why it has become to be the "most popular algorithm in pathfinding", wholesomely. The authors further pledge ways to "improve the performance of A*" and acknowledge its "huge success" as being the catalyst for the efforts that have already been made by relevant researchers in the field, to optimise and revise the architecture of it. Assuming the space complexity of the Pac-Man production, the authors suggest that A* is in fact a candidate solution, as it will have "less work to do, and less work will allow the algorithm to run faster".

Meanwhile, Kapi, Sunar and Zamri [33] also announce A* algorithm to be the "most prominent pathfinding algorithm", applied to "grid maps" in video games, that has "dominated the field for decades". The authors identify A* algorithm to "outmatch" various contemporary algorithmic designs, like that of Bee algorithm and Ant Colony Optimization (ACO) within "complex environments"; despite A* algorithm's memory consumption being "three times" worse on average. Although it is considered a "classic pathfinding solution", the authors remark that A* algorithm is "still being implemented and benchmarked, and it is further optimised in most of the current researches". Thus, extending its relevance to the investigation proposed by this work.

Furthermore, also extending the favourability of A* algorithm in maze solving applications, Barnouti, Al-Dabbagh and Naser [38] to explore the application of A* for the nature of strategy orientated video games. The authors acknowledge A* algorithm to be a compound of a uniform-cost search and a heuristic search, that is "widely used in pathfinding and graph traversal" applications. The authors continue to remark that A* does not only find a path between a given source and destination but finds the "shortest path quickly". Concluding that A* is a very capable search algorithm, whose overall performance is "acceptable and able to find the shortest path between two points". As previously seen, they too refer to the algorithm as the "most popular algorithm in pathfinding", given its successes.

Additionally, Rafiq, Kadir and Ihsan [39] recognise A* algorithm to be "one of the most popular techniques used for pathfinding" in video game development, given "its accuracy and performance". The authors reveal that the algorithm has been applied in "several video game genres", such as real-time strategy games, role-playing games, racing games, and turn-based strategy games, in the context of NPC's. They continue to reason its prominence in the field, due to its "simplicity", for which is why it has and continues to be "chosen by programmers to solve pathfinding problems"; alongside what the authors acknowledge the A* algorithm to have achieved, in finding the "minimum solution by finding the shortest path" between any given two nodes. The authors in support of their popularity investigation, determine that A* algorithm between the period of '2010 – 2018' was the most "popular technique applied to pathfinding in game development", opposing metaheuristic techniques such as Genetic Algorithms (GA) and ACO, despite their "constant increase in usage as pathfinding algorithms" and being deemed to be "better than heuristic techniques", in terms of "time and memory usage". However, the authors conclude from their investigations that "improved A* algorithms" in fact return the "fastest path in the shortest time", overlooking the "better performance" of metaheuristics overall. A few of many existing variants of the A* algorithm that the authors declare are: IDA* and Hybrid A*. Providing the space complexity of the Pac-Man production proposed for our experimentation, metaheuristic techniques are sought to be excessive, for which the authors acknowledge to be comparatively beneficial for more "complex maps", alternatively.

C. Jack Moorin

The pathfinding functionality of shortest path problem solutions has applications within video games for artificially controlled characters [58] but also within the decision-making process behind classical AI planners when considering potential rewards over a given planning horizon which can potentially be mapped into a standard shortest path problem and therein solved using a pathfinding algorithm [59]. This is discussed within the 2015 paper 'Classical Planning with Simulators: Results on the Atari Video Games' where Lipovetzky, Ramirez and Geffner discuss the use of traditional pathfinding algorithms within AI Planning. By formatting a given planning horizon as a hypothetical weighted graph pathfinding algorithm can be applied to find the optimum route, which in the case of AI Planning is the combination of decisions resulting in the "maximum possible reward". Within this application, the authors consider Dijkstra's pathfinding algorithm for use in this approach to optimally solve the ideal route. Whilst the ability of the algorithm to always find the optimum route is praised, they define it as being inefficient over large state

spaces as well as other blind search methods such as Breadth-First Search and do not consider it for the purpose of their AI Planner. This trade-off of performance in favour of ability makes Dijkstra an interesting choice for an algorithm implementation when regarding the application of video games as though performance is often favoured due to their real-time nature, it could be used as a benchmark by other faster algorithms such as A*.

The 2011 paper 'A*-based Pathfinding in Modern Computer Games' by X Cui and H Shi also finds fault with the application of Dijkstra's algorithm, stating that it was "soon overwhelmed by the sheer exponential growth in the complexity of the game" [60] and provides numerous suggestions of algorithms which vary upon the structure of A* as potential replacements. While this of course does again report on the inefficiency of Dijkstra's algorithm in the face of largely more complex graphs, the paper considers notably more recent and highly complex games as being the source of the need for algorithms better suited to them. While Dijkstra's algorithm has now been left behind in the face of the requirements of recent games with more complex graphs, Dijkstra was used initially within earlier games to great effect despite being rendered obsolete over time. When considering the comparatively low complexity of the unweighted graph within the discussed Pac-Man game, Dijkstra could perhaps operate as efficiently as the other implemented algorithms and otherwise provide a useful comparison between algorithms such as the often more efficient but less optimal A* and those that always output the optimum route between nodes but are faced with the problem of becoming drastically slower in the face of growing complexity.

Despite the previous paper's dismissal of the use of Dijkstra's algorithm when pathfinding over larger weighted graph, in the 2018 paper 'Comparative Analysis of Pathfinding Algorithms A*, Dijkstra, and BFS on Maze Runner Game', the authors Permana, Bintoro, Arifitama and Syahputra find Dijkstra's algorithm to be conversely optimal for pathfinding when being implemented upon an unweighted graph structured as a two-dimensional grid, the same configuration used within the Pac-Man game [61]. Within the paper the A*, BFS and Dijkstra algorithms are implemented within a video game application and their performance tested in finding a path from a given start node to a target node after the space within the graph has been obstructed with the addition of numerous obstacle blocks. The performance of each algorithm is trailed against a series of obstacle layouts, each further limiting the available space within the graph. In direct contrast to the previously noted inefficiency of Dijkstra's algorithm in comparison to that of A*, within the implementation of an unweighted graph as a two-dimensional grid Dijkstra repeatedly outperforms both the A* and BFS algorithms and is chosen by the authors in the paper's summary for implementation within the game. This paper is of course highly advocating the use of the Dijkstra algorithm and is especially practical for the purposes of this study given the large similarity of the authors application of the algorithm to the application intended for those algorithms implemented within this study.

Dijkstra's algorithm conceived by Dutch computer scientist Edsger W. Dijkstra in 1956 and first published by him in the paper 'A Note on Two Problems in Connexion with Graphs' [62] three years later in 1959 is an algorithm commonly used to solve the shortest path problem [63]. Originally the algorithm only discovered the optimal path between the start and target nodes within a graph however, the more commonly used variant of the algorithm today, and also the variant implemented within the Pac-Man variant, defines a shortest path tree for a given source node by finding the shortest path between that node and every other within the graph [64].

In the 2001 paper 'The PN*-search algorithm: Application to tsume-shogi' [65], Seo, Iida and Uiterwijk propose the use of a new proof-number (PN) search algorithm, PN*, for application within a computerized version of the Japanese 'Shogi miniature problem', 'Tsume-Shogi' in which players attempt to 'checkmate' their opponents king within a given Shogi board layout, similar to the western equivalent of a chess problem. The proposed PN* algorithm attempts to improve upon the performance of already existing PN algorithms, namely the Breadth-First variant of the PN-Search by using "methods such as recursive iterative deepening, dynamic evaluation, efficient successor ordering, and pruning by dependency relations", which transforms the approach of the algorithm into that of an iterative-deepening depth-first search. The 'Tsume-Shogi' problem, similar to the previously discussed Maze Runner Game, has many parallels with the pathfinding problem within the proposed Pac-Man game. Both problems are applied within two-dimensional grid graph structures and consider the optimal movement of an entity within the graph. This paper of course advocates the use of the developed PN* algorithm and its iteratively deepening depth-first structure within the 'Tsume-Shogi' problem and creates considerable favour for the idea of the implementation of a similar algorithm within the study documented by this paper.

D. Arpit Sharma

For the task of finding the shortest path from the Pac-Man's current location to the closest coin, several algorithms were considered based on the criteria described in the above section. Bidirectional Breadth-First Search algorithm and Bidirectional Dijkstra algorithm were given priority because of the desirable characteristics they possessed discussed in detail in this section.

The BFS algorithm was invented in 1945 by Konrad Zuse. One of its reinventors, Edward F. Moore, redesigned it in 1959 for finding the shortest path in a maze [54], which is the very same problem, that is being tested in the experiment. It is still one of the classic algorithms used for the problem of maze solving.

For the game used in the experiment, as all the nodes have an equal weight of a single unit, the grid can be considered as an unweighted graph as the cheapest path is always the shortest one. BFS is a suitable for unweighted graphs. As the next closest coin can be in any direction, an undirected search was needed and because the Breadth-First Search 'works on both undirected and directed graphs' [56], it was preferred. The algorithm searches in all directions and terminates exploration once it finds the target.

The time complexity is very crucial for video games as it amounts to undesirable lags as it determines, how much 'time the program will take' [58] to compile. If the computation time for path planning is high, it will slow

down the game and affect its playability. The time complexity of the unidirectional Breadth-First Search in big 'O' notation is $O(b^d)$, where 'b' and 'd' are described in the previous section. To decrease the time complexity, to make the algorithm quicker, a bidirectional Breadth-First Search algorithm was preferred as it has a much lesser time complexity of just $O\left(b^{d/2}\right)$. The reason is that instead of just searching from a single starting node, the search begins from the target node as well, at the same time. The termination condition is met, whenever both the searches have traversed any common node in the graph-tree.

The space complexity of the algorithm is proportional to the 'storage the program will take' [58] while pathfinding as it is the number of nodes, the algorithm must traverse to find the shortest path from starting node to the target. Using the similar notation as used above, the space complexity of the unidirectional Breadth-First Search is $O(b^d)$. On the other hand, the bidirectional Breadth-First Search algorithm has the space complexity of just $O\left(b^{d/2}\right)$ as the search begins from both the starting node and the destination node, the graph is explored to a much lesser extent than it would be if the search had begun from a single end.

Algorithm completeness is one of the most important characteristics of an algorithm when it comes to video games. Within the application of video games, it its generally more important for a solution to always be found, no matter the incurred overhead than to use a less complex method with a lower chance of ultimately finding it. Since, it explores the nodes layer by layer, space and time complexity offered by the algorithm is considerable, but it guarantees to find the target node even in an infinite graph. Although, generally, the graph generated by the algorithm while path planning is finite because of the very limited and fixed size of the maze, but even if it had been more complex, it would still eventually find the target. Therefore, the Breadth-First Search is a complete search unlike the Depth-First search, which sometimes gets trapped inside part of the graph away from the goal and never returns [55]. It makes it a good shortest path searching algorithm. The bi-directional variant is also a complete search and guarantees the solution.

Since, it had been found out at this stage that the bidirectional Breadth-First Search algorithm is good for making optimum paths, and for video games in general, another algorithm to consider for making a bidirectional variant of, was the classic Dijkstra for the experiment as its very similar to Breadth-First Search in working as it also explores the entire area while searching, because of which its bidirectional variant is also a 'complete' algorithm. Although, its bidirectional variant can be quicker at times, but is not always guaranteed to return the shortest path unlike the typical bidirectional BFS. The reason is that the algorithm may terminate on a common node leaving the alternative, more optimal node at times.

However, its longest possible path can still be a lot shorter than the possibility of sub-optimal path calculated by the Bidirectional Depth-First Search algorithm. Although the results expected from this algorithm are not always optimal, it can still be a competitive algorithm for the pool of algorithms selected for this experiment and for future studies regarding the optimization of the algorithms.

The algorithm works by 'dividing two graph matrix' [57] and then conducting independent searches. The results are combined at the instance, the algorithms search space intersects. Both the units use the 'shared memory' in the program being run.

The IDA* from the A* algorithms which have been proven algorithms in the gaming world as discussed in the above sections, have been 'derived' [59] from Dijkstra, making its bidirectional variant, Bidirectional Dijkstra, alongside Bidirectional Breadth-First Search, one of the classic candidates to be preferred for the experiments involving video games.

IV. Solution Design

A. Common

Following the decision to implement numerous pathfinding algorithms within the Pac-Man variant, the requirements of accurately testing and trialling each of the implemented algorithms to compile results reliable enough to inform further observations regarding their suitability for use in the shortest path problem becomes a notably large undertaking and threatens to push the time constraints of the study. Fortunately, the entirely digital format of the proposed Pac-Man game and the full automation of the Pac-Man character within the game following an algorithms implementation, creates the possibility of extending Pac-Man's automation to cover not juts collecting every coin node within the maze environment but also with regard to its own testing and compilation of results regarding the overall performance of each of the implemented algorithms. Considering the definition of a new maze environment and repositioning of Pac-Man within the level, following the resetting of a level encountered within the original Pac-Man game due to the player either having successfully completed the previous level or having fallen victim to one of the non-player character ghost enemies within the grid, a potential aim for the Pac-Man variant implementation would be to include this functionality within the implementation of Pac-Man using an implemented pathfinding algorithm. By being able to correctly reset the environment and position Pac-Man at a new starting position with the implemented variant, enables the repeated resetting of the maze environment and repositioning of Pac-Man required to implement an iterative testing architecture of each algorithm's performance with Pac-Man being positioned at each of the possible starting points within the grid. This would allow the testing of each algorithm to not only be carried out entirely autonomously without the need for oversight or involvement by those working on the project but also dramatically increase the amount of result data able to be compiled during the study as if correctly implemented, iterative testing functionality would enable the Pac-Man variant to trial and compile results regarding each algorithms performance considerably faster and more efficiently than if the testing process were completed by hand. Increasing the number of results compiled during the testing process further increases the validity of the results collected and of any observations made following their compilation and analysis as the repeated carrying out of trials remains a commonly used practice within all fields of study regarding testing processes caried out where the outcome of the test is unknown to the person or people completing them. Repeating trials allows them to confirm the accuracy of their tests as any results that were affected

by one or more external factors changing the outcome of the test would become more apparent when considered in combination with other tests not affected in the same way as the erroneous results would not fit the established trend.

Typically, within the scientific field, tests are repeated a minimum of three times to ensure their accuracy. For the purposes of this study, provided none of the implemented algorithms are in any way stochastic and implement an entirely deterministic approach to generating a path between the provided start and target nodes, repeatedly trialling the performance of an algorithm with Pac-Man beginning at the same starting position is unnecessary as provided conditions remain the same between tests, the same path will be generated by the algorithm with little to no variance between the time taken to compute the result. Therefore, instead of wasting time doing this the Pac-Man variant when determining the performance of each algorithm at collecting every coin node within the graph, any autonomous testing functionality should instead trial the algorithms performance with Pac-Man beginning at each of the potential starting positions within the graph to ensure the algorithms performance is tested over a variety of graph configurations.

With the aim of further extending the suggested iterative testing functionality, a further improvement would be for Pac-Man to trial each of the implemented algorithms in sequence, allowing for the creation and compilation of results for all of the algorithms in one go. Implementing testing in this way would make the process entirely autonomous and provide the easiest and most efficient solution to reduce the overall time spent compiling results on each algorithms performance.

B. Adam Hubble

Proceeding from the review of the literature studied, and relevant to the algorithms elected for the individual contribution to the investigation proposed, A* algorithm and its bidirectional counterpart, Bidirectional A*, were selected in correspondence with the general findings of heuristic searchers being optimal within simpler environments; similar to that of the structural arrangement that Pac-Man engages. As not explicitly referred to within the review conducted, Bidirectional A* was also elected as one of many suggested, improved variants of A*, given its offering for space and time complexity reduction [42], despite its minimalistic advancement from the original, unidirectional implementation.

As revealed by the series of works submitted to the field of pathfinding, specifically for video game productions, the A* algorithm undoubtedly remains to be regarded as one of the most popular and provably optimal search algorithms in contemporary research, regardless of it being defined as a classic appeal to graph traversal optimisation. Its application within the Pac-Man game was sought optimal, providing its best-first search capability according to heuristic evaluation, for exploiting the most promising nodes in the grid, to reach a specified goal node. For which as previously realised, A* is competent in guaranteeing the optimal shortest path, as recognised as a complete search algorithm, assuming that the heuristic is of an admissible measure.

In continued mention of heuristics, supporting the relevant authors claims regarding the performance variability of the algorithm when using different heuristic functions, it is sensible to further propose the implementation of multiple, commonly used heuristic methods, to satisfy the comparability of the investigation. Given that the A* algorithms bidirectional variant utilises heuristic capabilities also, heightens the uncertainty of outcomes produced by the experiments led, in concentration of the iterative benchmark projected for purpose.

C. Jack Moorin

Following the review of the papers discovered relevant to the study in the previous section of the report, the algorithms chosen to be implemented for the individual contribution to the project were the Dijkstra and Iterative Deepening Depth-First Search algorithms. Of the two potential variants of Dijkstra, the variant used to create a shortest path tree, using Pac-Man's position as the source node.

As discovered from the literature review, Dijkstra's algorithm is known for being notably less efficient than other pathfinding algorithms, especially when implemented across highly complex graph structures, however, given the simplicity of the maze environment within the Pac-Man game as a relatively small and unweighted graph, Dijkstra could potentially perform outperform other algorithms known for being more efficient, due to the fact that it will always output an optimal path to a target node which could result in Pac-Man collecting every coin node within the graph faster.

Unlike Dijkstra, IDDFS does not always output the optimum path however, it is expected to perform more efficiently than Dijkstra as it is notably less complex. This will make for an interesting comparison between the two within the study as one approach favours performance while the other favours efficiency.

D. Arpit Sharma

After conducting the review of the papers mentioned in the previous section, looking out for the desired characteristic for the shortest pathfinding algorithm for the problem being used in the experiment, an unweighted two-dimensional maze, where the search had to be undirected as the direction of the goal is unknown, Bidirectional Breadth-First Search algorithm and Bidirectional Dijkstra algorithm were finalized. The algorithms were used to plan the path for Pac-Man to get to the next closest coin, in the game.

Based on the paper reviewed, the Bidirectional Breadth-First Search algorithm is thought to be one of the most efficient and promising algorithms based on its space and time complexity and algorithm completeness. Also, found from the papers was that the bi-directional BFS algorithm was originally implemented for maze solving by one of its reinventors, making it an eligible algorithm for the experiment. Most importantly it always guarantees the shortest path, which is one of the main goals of the experiment.

But the second chosen algorithm, Bidirectional Dijkstra is not expected to yield as good results as the first algorithm as it does not always promise the shortest path. However, in terms of algorithm completeness, it is found to be as promising as the previous algorithm, which is very important to prevent glitches in video games.

V. Solution Implementation

A. Common

Following the decision to implement the trialled algorithms within the maze environment of the popular Pac-Man game, a variant of the game was implemented using the chosen programming language, Python [47], and the Visual Studio [48] integrated-development environment. The implementation of our Pac-Man variant, of course borrowed heavily from its's predecessor, with respect to both the mechanics of the original; a player-controlled character: Pac-Man, is directed through a grid-based maze environment with the aim of collecting as many of the power pellets spread throughout the maze as possible whilst also avoiding the multiple ghost enemies also in the maze, and visually: with the assets created for and used in our variant being either heavily inspired by or taken directly from the original game. For the purposes of our study, further added to this base implementation of the game was functionality enabling the user to select from a list of defined pathfinding algorithms, an algorithm to be used by an entirely computationally controlled instance of Pac-Man, to traverse the maze environment and collect each power pellet. To further enable the testing and result compilation necessary of each defined algorithm, iterative testing functionality was implemented enabling repeated trialling of either a specific algorithm or each algorithm in sequence with the total time taken for Pac-Man to entirely traverse the environment and the number of nodes it traversed trialled and output to a file for each possible starting point within the maze.

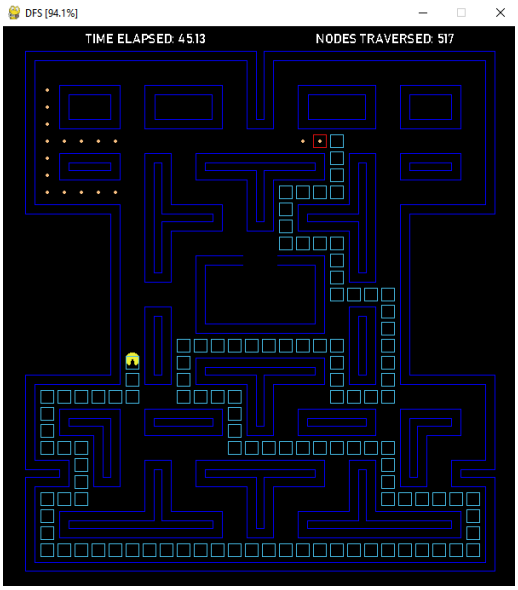


Figure 1: Graphical visualization of the Pac-Man game variant, implemented for the purposes of this study. The figure illustrates Pac-Man using the Depth-First Search algorithm to find a path between itself and each of the coin collectibles within the maze.

For the purpose of ease, the pygame [49] python library was used in development of the Pac-Man variant. Designed bespoke for the purpose of the creation of video game applications, the pygame library provided functionality for the creation of the creation of the game screen window and the drawing of the game's graphics to it as well as processing of the keyboard input events used to control the game. Use of the pygame library enabled the creation of the Pac-Man variant in a notably shorter amount of time than would have otherwise been possible and therefore meant that more time was able to be dedicated to the development and trialling of each algorithm.

The creation of the graphical assets used in the Pac-Man variant not available online, was handled using the Piskel online sprite editor [50]. An entirely free, online, and open-source application, Piskel enables the creation of static and animated pixel art sprites and further exporting them to various file types, enabling their use in other applications. For the purposes of this project, Piskel was used in the creation of the frames used in the animation of the Pac-Man character and the four Non-Player Character ghost enemies.

To enable easy editing of the maze environment navigated by Pac-Man during the game, the environment is constructed from an external text file, walls.txt, which is loaded into the application as its started and its contents used to define the two-dimensional array used both for collision detection by the Pac-Man character when traversing the map and by the algorithms when determining a path through the maze. The file itself contains a twenty-eight by thirty grid of characters, each of which defines the purpose of its respective cell within the maze. The character 'C' for example denotes the corresponding cell of the maze environment contains a coin, the misnomer by which our own variant of the Pac-Man game refers to the power pellets of the original, and when the game is loaded will cause a coin to be created within the corresponding maze cell. Using an external text file to define the maze environment used within the game makes it easier to edit the game environment as to change the purpose of a grid cell the relevant character within the file must simply be changed to refer to the desired purpose of the cell.

To assist in the implementation of the various pathfinding algorithms used, a debug mode was implemented, which when activated by pressing the 'tab' key, draws a two-dimensional grid onto the game environment which can be used as a reference by the programmers to easily work out the grid reference of any cell in the maze. This is useful during implementation as the algorithms can be set to output the paths they create or the cells they check, in order for the programmer to understand how the algorithm is attempting to search the grid. By coupling this approach with the grid provided by the debug-mode the programmers can easily visualise the search process and will therefore have a better chance at understanding the cause of any potential errors with the algorithm.

Also implemented to assist in understanding how each of the algorithms searches the grid is the drawing of Pac-Man's current path within the maze also drawn on top of the game environment, so that it's clearly visible the path the algorithm has generated for Pac-Man to use in traversing the maze. The maze cell containing the coin targeted by the pathfinding algorithm has an overlaying red square drawn on it and the cells making up Pac-Man's path through the maze have a blue overlaying square drawn on them. This allows programmers to easily see if the implemented algorithm is correctly finding the shortest path to the targeted coin or if the generated path is taking some other convoluted route erroneously and the algorithm's implementation needs to be checked.

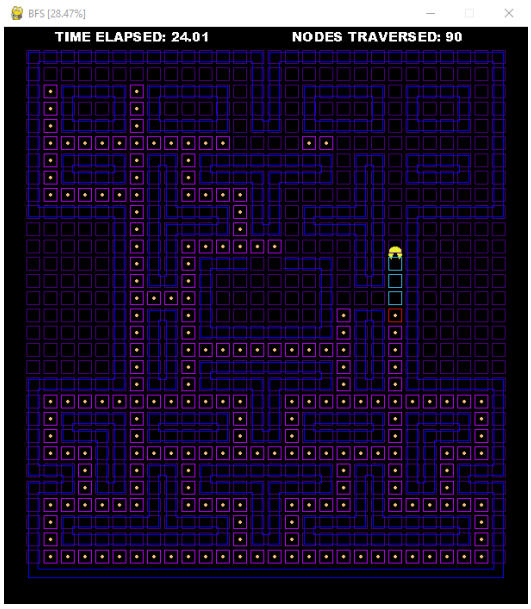


Figure 2: Graphical visualization of the Pac-Man variant during runtime, showcasing the debug mode implemented.

Figure 3: Graphical visualization of the Pac-Man variant, at the start of runtime. Illustrating the menu-environment of the application where algorithms can be executed in isolation or sequentially.

To ensure the comparability of the algorithms implemented as part of this study, an implementation of the Breadth-First Search algorithm without pathfinding functionality is used to determine the coin node to be targeted by the chosen pathfinding algorithm. This implementation of using the BFS algorithm to determine the target and then the pathfinding algorithm to determine Pac-Man's path is used for the testing of all of the pathfinding algorithms despite the implementation of BFS with pathfinding functionality as the first of the studied algorithms as many of the other implemented pathfinding algorithms such as Dijkstra do not contain their own search functionality and require knowledge of the location of a targeted node within the grid to inform the process of their pathfinding. Therefore, despite the fact that both the implemented BFS algorithm and the other studied algorithm with search functionality, Depth-First Search, could both be used to locate a coin node as well as determine a path through the grid, the performance of these algorithms will be comparable to those without search functionality as the only varying factor will be the algorithm's pathfinding functionality.

The iterative testing functionality developed enables either a chosen single pathfinding algorithm or each in sequence to be repeatedly trialled and have its performance results compiled and output to a file completely autonomously. When trialled the selected pathfinding algorithm will be used to navigate Pac-Man through the maze environment, collecting each of the coins spread throughout the grid. The performance of each algorithm is trialled repeatedly with a total of two-hundred and eighty-five tests being carried out per algorithm with Pac-Man starting at each of the possible starting positions within the maze. The total time taken for Pac-Man to collect every coin and the number of nodes it traversed in doing so for each trial is output to an external text file for the algorithm currently being trialled. If all the algorithms are being tested in sequence, the first algorithm to be implemented as part of this study, Best-First Search, will be trialled initially and once its trials are complete the next algorithm in the list will be trialled until each of the algorithms has been tested.

With respect to the contribution of each study member to the implementation of the Pac-Man variant, it is worthy to note that each member contributed equally to the development of games initial state, the supporting functionality discussed regarding debugging and iterative testing, as well as the implementation of the relatively simple Breadth and Depth-First Search algorithms, that were rendered complete during the early development phase of the project. Following the initial development, each member had elected and subsequently implemented two search or pathfinding algorithms, that each discovered to be worthy for the investigation said, as reviewed within the Solution Design section. The individual development of each of these algorithms is detailed below, specific to each member of the study.

B. Adam Hubble

*A** – as the algorithm traverses through the graph or grid-based map representation of the game's environment, resembling a maze, A* follows a path of the lowest known cost (least distance travelled in the shortest time), whilst maintaining a sorted, priority queue of alternate path nodes along its route of traversal. At any given point in the environment, if a node of the path currently being traversed features a higher-cost (more expensive to route) than another encountered (visited and known) node, the algorithm sensibly abandons the higher-costing node and instead, traverses the lower-costing node. This procedure recurs until the traversing node becomes the goal node, representing the destination of the path. Summarily, A* conforms to a best-first search, which explores a graph by expanding the most promising nodes according to cost that is compiled using a heuristic evaluation function, denoted as $f(x)$, and finds the least-cost path from a given starting node (the source) to a specified goal node (the destination). The heuristic function it utilises, a "distance-plus-cost" [40] evaluation method, purposes to determine the order in which the search visits or explores nodes in the graph tree,

or respectively the grid. For which, the function is known to be a summation of two subsidiary functions, denoted as:

$$f(x) = g(n) + h(n)$$

Where $g(n)$ represents the path-cost function, representing the cost or actual distance from the starting node to the current node of traversal. And $h(n)$ represents an admissible, heuristic estimate of the distance to the goal node, from the current node of traversal; for a path to be considered the least-cost path from start to goal, the heuristic "must not overestimate the distance to the goal".

For the algorithm's implementation, initially, given the grid-based representation of the game's environment, a two-dimensional array representing its structural arrangement is instantiated, to distinguish between the positions of nodes occupied by obstacles and nodes that are traversable by the Pac-Man avatar. Given the uniformity of the traversable cells in the original production, the grid adopts the schematic of an unweighted and undirected graph, where nodes comprising the map are binarily classified as the values of '1', representing the occupation of obstacles, and '0' representing their absence; this is iteratively addressed by a 'for-loop' statement. In knowing of this configuration, the path-cost of the algorithm can simply be acknowledged as a representative of the number of nodes that it is comprised of. Meanwhile, the start and goal nodes of each path are instantiated to be relative to the position of Pac-Man in the grid, at the time of path computation, and the position of the goal also; both the start and goal nodes configured for every path compiled are bound by a class, namely 'Node', for which they are objects of. Abstracting the properties of each node comprising the algorithms procedure, was sought beneficial for maintaining the robustness of the codebase and for ensuring cost evaluations are performed accurately, where each node as an object of the class, has an associated position, parent node, path-cost (g), heuristic estimate (h) and final cost (f) value. In continued mention of the class, therein features two magic methods [41], otherwise known as Dunder (double underscore) methods, that are unique to the python programming language, and enable comparison-wise statements to be autonomously invocated for efficiency practises; thus, allowing the theoretical nature of the algorithms procedure to be better realised, from an enhanced prospect of simplicity. To note, the 'Node' class is also borrowed by the bidirectional variant of A* proposed and would also be compatible with other heuristic-driven algorithms, in future works.

Preceding the algorithmic operation of A*, the algorithms priority queue variables are instantiated, as a set of open and closed lists of nodes, each represented by an empty array variable initially, representative of the list of "nodes that are children of already expanded nodes, but have not been expanded themselves, yet" [37], and the list of "nodes that have already been processed". Where at each step of the procedure, the node with the lowest cost $f(x)$ is removed from the priority queue or open list, becoming the current node of traversal; the $f(n)$ and $g(n)$ values of the current node's neighbours are then updated, before they are appended to the queue for processing in the next iteration. This sequence of operations continues to recur until the node removed from the queue is the goal node, which can be identified by the least cost and f value compared to any other node in the queue; subsequently, the f value of the node traversed is then also acknowledged as the cost of the shortest path, given that its h value in an admissible heuristic, would equate to the value of zero (no distance to the goal from the current node).

To orchestrate this scheme, initially, the start node of Pac-Man's path is appended to the open list, for neighbourhood processing. In achievement of the algorithms recursive state, a 'while-loop' declaration is then placed and conditioned by the open list, for which continues to iterate the procedure until no nodes are contained by the list, such that the length of the list equates to zero. Therein, for each cycle of the procedure, the nodes comprising the queue are sorted by ascending order relative to their f values, where the node with the least cost surfaces to the top of the queue; this capability is addressed by the language-provided 'sort' method in compliance with the 'Node' class's iterative operator, contained by its '__it__' Dunder method. Proceeding from the sorting process, the current node is then assigned to the properties of the least-cost node in the open list, before the least-cost node is then removed from the open list and oppositely appended to the closed one; this serves to identify the current node as processed, for which prevents its cost being calculated multiple times.

Prematurely, the position of the current node is then compared via 'if-else' statement, with the position of the goal node, for reducing the potential time complexity of the algorithm; if the condition presents to be 'true', the path from the start node to the goal node is thus known and can be constructed for Pac-Man's traversal operation(s). For the path's generation, an empty array variable is instantiated (the path), where within a subsidiary 'while-loop' declaration, the positions of the parents of the current node traversed can be appended to it; through backtracking, the algorithm is orderly able to construct the path traversed, from the goal node to the start node. Upon its compilation, the start node is then appended to the path, as not already considered, before the order of the path is then inversed to accommodate for the reversed cycling of its generation. Resultingly, a fully-connected path is processed and returned by the containing method, namely 'AStarSearch', which terminates the procedure and preserves operational time (reduced time complexity) by ignoring further statements in the corresponding method.

However, if the condition presents to be 'false', such that the current node traversed is not the goal node, then alternatively a series of directions representative of neighbouring nodes are populated in an array variable, relative to the current node's adjacencies, for which can only be of up to four directions: up, down, left, and right. Each neighbouring node's position is simply calculated via an addition operation, that sums the current nodes positional values with those of each potential direction calculated; this is orchestrated by a 'for-loop', that iterates through each potential direction representing the location of a neighbouring node in the grid, as contained within the corresponding array variable. For each neighbouring node, assuming its calculated position exists within the boundaries of the grid, and its value within the grid does not resemble an obstacle '1', thus meaning that the node is traversable, as well as the node not existing in the closed list already (its

cost has not been calculated previously), then aligned with the active heuristic estimation metric, the neighbouring nodes h value is calculated. This is achieved via invocating the heuristic evaluation method, that is configured with four popular distance measures (discussed later). Beyond the neighbouring nodes cost being registered, then for all nodes contained in the open list, if the neighbouring node is uniquely positioned in the grid and its cost is less expensive than any of the contained nodes, it is then appended to the queue. Upon being appended, in the following iteration of the procedure the node will be considered for expansion, in finding the shortest path between the start node and the goal node specified. All the above-mentioned conditions are addressed by a series of nested 'if-else' statements, accompanied by a Boolean state, for adhering to simplicity.

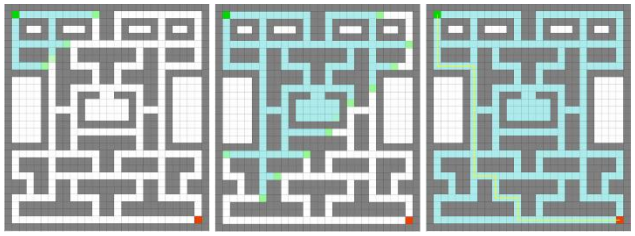


Figure 4: Graphical visualization of the path expansion and deduction phases, of the A algorithm. Relative to environment of the Pac-Man game.*

*Bidirectional A** – as opposed to searching from the start node to the goal node directly, A* algorithms bidirectional variant enables two searches to be conducted in parallel. Such that one of which searches originates from the start node whilst the other originates from the goal node; the most "ideal state is that two search programs will meet in the middle, then the time is only half of their original cost" [42]. However, if the conditions modelling the searches intersection detection is "bad", it is possible that the "search time can be doubled", alternatively. Operationally, bidirectional search's purpose to narrow the expansion of nodes in a graph tree, which typically enlarges over the course of a unidirectional search.

For its implementation, as an arguably simplistic variant of A*, the adaptation can adopt all functionality declared prior for the algorithmic procedure of A*. Differently, the path expansion and deduction schemes of the algorithm are abstracted into distinct methods, to appropriate the re-use of the available functionality, for effectively addressing the "forward and backward" modes of search that the algorithm simultaneously entertains. For which, two declarations of start nodes, goal nodes, open lists and closed lists also exist, to cater for the algorithm's polarity, that consequently redefines the condition of the algorithms recursive state, as the procedure instead recurs until both instances of open list contain no nodes to be expanded; unless a node is common to both searches, thus there being a point of intersection. Unlike Unidirectional A*, the termination condition of Bidirectional A* not only accounts for when each search reaches its target, formerly as the goal node, but when two searches intersect such that they "meet at the same node" [43].

Upon one of three components of the termination condition being satisfied, the paths generated by each mode of search up until the point of intersection, or the goal node(s) alternatively, is deduced from the all the nodes visited by the algorithm. Identical to the A* algorithm, an empty array variable is instantiated for each search (the paths), where within a subsequent 'while-loop' declaration, the positions of the parents of the current node traversed can be appended to it. Through backtracking, the algorithm is able to construct two paths, from either search's start node to its corresponding goal node, or to the node at which the searches intersect. Proceeding from either path's population, and in the company of a series of conditional and iterative statements, all nodes comprising either searches path that defer from the common pathing focus or in case of error, that surpass the point of intersection, are removed from consideration to ensure that the least-cost path is found. In which both paths can then be merged, to find the shortest path between the common start and goal node of the search; here, the start nodes of each search operation are appended, if not already concerned in the path compiled.

Oppositely, if none of three termination conditions are satisfied, alike Unidirectional A*, Bidirectional A* then continues to compile a series of promising neighbouring nodes, via invocating the relevant path expansion method for each mode of search, until the termination condition is eventually fulfilled. Thus, Bidirectional A* also guarantees that a path is generated but its optimality is questionable, considering that it "does not guarantee the route found to be optimal, if the search ends when the forward and backward search meet in the middle" [44].

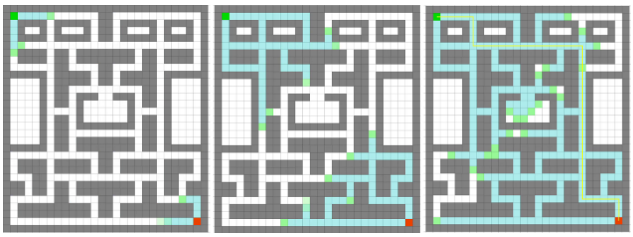


Figure 5: Graphical visualization of the path expansion and deduction phases, of the Bidirectional variant of the A algorithm. Relative to environment of the Pac-Man game.*

Heuristic Evaluation – as informed search algorithms and thereby users of heuristic functions, A* and its bidirectional counterpart can find the shortest path through a search space, using a series of heuristic methods purposed for estimating how close an agent (Pac-Man) is to the specified goal; that in return, determines the order in which the search visits nodes in the grid. Notably, it must be acknowledged that the "time complexity of A* depends on the heuristic" [40] active in its search.

For comparative sake, the application features four well-known heuristic functions for grid-based representations of maps; this was sought to be useful for investigating the resultant behaviours of Pac-Man, specifically for realising a supposed trade-off between speed and accuracy. The heuristic functions sponsored by the work submitted are Manhattan, Euclidean, Octile and Chebyshev-type distances, each of which methods propose a unique set of operations, optimal for separate environmental conditions and traversal capabilities of agents. For simplicity, all heuristic methods are conditioned by a series of 'if-else' statements, bound by one method, namely 'Heuristic', where either heuristic calculation can be invocated relative to a string variable that is 'passed-by-value'. For their

implementation, a series of math-related expressions are simply replicated [45].

Purposed for reducing the relative space and time complexities of each algorithm proposed by the experimentation presented, Manhattan distance claims a presence in each of the relevant methods, as a "good heuristic" for when a grid only allows horizontal and vertical movement. With the support of an 'if-else' statement, rather than invocating the procedures of the algorithms featured, when the start node is adjacent to the goal node, Manhattan distance is used to determine whether a path containing the start and goal nodes only, should be compiled instantaneously; this is satisfied upon the distance between said nodes being estimated to the value of one. Also recognised as city block distance, a distance value of one infers that the distance between the start and goal nodes is estimated to be one cell relative to the grid, thus, meaning that the nodes form an adjacency and can further be traversed too directly. This mechanism although not officially recognised within the field, enhances the fluidity of Pac-Man's traversal, without deferring from the original implementations of either algorithm supported by the study.

C. Jack Moorin

Dijkstra – requires that the shortest path between the starting node and each of the other traversable nodes within the graph is discovered to ensure the path between the start and target node is the shortest possible. On the basis that the graph used for our implementation of the Pac-Man game, the maze environment defined as a two-dimensional grid within the code, is an unweighted graph the distance of the shortest path between Pac-Man's starting point and the target node can be referred to as the number of nodes traversed by the path between the two. To hold this distance for each node a two-dimensional array is defined that is the same size as the array used to define the maze environment. The start node's distance is set to zero as it is already the node that the search will begin at and added as the first item in a queue. This queue is then iterated through with each iteration adding the traversable nodes neighbouring the currently iterated node to the list and setting their distance to be one higher than that of the currently iterated node as the neighbouring node is only a single traversal away from it.

Once the list has been fully iterated every traversable node within the graph will have had a distance assigned equal to the minimum number of nodes that need to be traversed in order to reach it. By using these distances, it is possible to work backwards from the node of the targeted coin in order to find the shortest path to it from the starting node. A second queue is defined to hold the discovered path and the target node set as the first element. Looping until the target node has been set to the position of the start node, the first of the nodes neighbouring the current target node to be checked that has a distance value one less than that of the target node is appended to the path and becomes the new target node. This looped process will continue until the start node has been reached and which point the path queue will contain the shortest path between the target and the start nodes. Once it has been reversed it will then provide Pac-Man with the path from its current node to the target node.

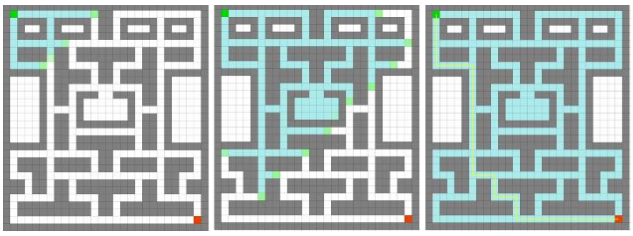


Figure 6: Graphical visualization of the path expansion and deduction phases, of Dijkstra's algorithm. Relative to environment of the Pac-Man game.

Iterative Deepening Depth-First Search – makes use of two functions in order to handle the recursive nature of the algorithm. The algorithm's 'root' is the contents of the 'IDDFSearch' function and is used to repeatedly call the second function, 'DLS', which performs the actual depth limited search functionality of the algorithm, with incrementally greater provided search depth until the search has discovered the target node. The 'DLS' function is defined as an 'inner function' within the 'IDDFSearch' function as the 'DLS' function needs access to the path list defined within the enclosing 'IDDFSearch' function. Before defining the inner 'DLS' function, the 'IDDFSearch' function defines the aforementioned path list as well as a, currently empty, list of nodes that have been already checked during the search process to avoid the algorithm repeating over itself. As the 'DLS' function can call itself recursively potentially multiple times, it is necessary for these to be defined outside of the function, so they are not overwritten following each execution of the function.

The 'DLS' function takes in the parameters of the node to be searched next and the remaining depth left of the search. If the remaining depth is zero, the provided node is checked and if it is the target node returns it. If it not the target node however, the 'None' keyword is returned indicating that the target node was not found. If the remaining search depth is greater than zero, the 'DLS' function recursively calls itself upon the nodes neighbouring the provided node with one less search depth than was input to the current instance of the function. This recursive calling of the function is what gives the function its depth-first structure as, with sufficient search depths, the search function will be called on the start nodes first neighbour, which will then call the function on its first neighbour and so on. If the function returns here with the keyword 'None' then the completed search was unsuccessful, and the search algorithm is then called on the next neighbouring node. If a node index is returned however, then the search was successful, the current neighbouring node is appended to the path and is returned by the function. When the instance of the function called within the 'IDDFSearch' 'root' returns the search with the current search depth has been completed. If this instance of the function returns a node index, then the target node must have been found and the start node is appended to the path as its final node. If the 'None' keyword has been returned however then the current iteration of the search must not have been successful, the search depth is incremented by one and the search algorithm called again on the start node.

D. Arpit Sharma

Bidirectional Breadth-First Search – the algorithm has been implemented to find the shortest path between the starting node and the destination node in an unweighted

grid, implying that the shortest path in length between then nodes is also the optimum. Two units of unidirectional Breadth-first algorithm have been used as subparts and are run simultaneously, from both the starting node (Pac-Man's location) and the target (next closest coin location) towards each other.

The ordinary Breadth-first algorithm uses the graph theory for path planning by exploring the nodes in a layer-by-layer fashion. The algorithm starts at the root node and explores all the adjacent nodes. Then, for each adjacent node, it explores its neighbouring nodes one by one untill it reaches the target. It uses a First in First Out, 'FIFO queue' [51] for the nodes in the graph obtained by expanding their parent nodes. The unexplored or unvisited nodes are kept in a linked list or queue called 'open' and after being visited are placed in the 'closed' queue. The code contains two methods for the two phases to perform the path planning: 'space exploration', and 'shortest path deduction'. Figure 6 shows the visualization of these phases, where blue cells are the cells that have been explored till that point, and the yellow line is the final shortest path calculated by the algorithm.

In the space exploration phase, each unit performs its blind search generating a graph tree by traversing through the neighbouring nodes. At first, using the FIFO principal, the first entered node is popped from the 'open' queue. Then it explores all the unexplored neighbouring nodes (that is if they are not inside the 'closed' queue), before they are then appended to the open and closed queues. Next each branch formed between the current and neighbouring node is added to the graph tree array. The method returns the generated graph tree. When the end of the graph tree is reached, meaning that there are no more neighbouring nodes to explore, the open queue starts emptying.

The above phase runs simultaneously for both units. The exploration phase ends if any of the individual units explore the graph tree fully or a common node explored by them is found by checking the intersection of both closed queues, and hence, the algorithm enters the phase of path deduction.

During the path deduction phase, the shortest path is deducted by each unit one by one by traversing the generated graph tree by moving in the opposite direction, from the target node till the start node is reached.

The path calculated by the forward-moving BFS is added to the reversed path returned by the reverse moving BFS (to arrange the path in the proper sequence) and returned as a complete shortest path needed for the Pac-Man to traverse.

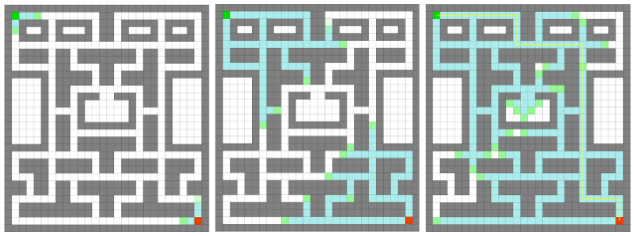


Figure 7: Graphical visualization of the path expansion and deduction phases, of the Bidirectional variant of the Breadth-First Search algorithm. Relative to environment of the Pac-Man game.

Bidirectional Dijkstra – it also uses two units of the ordinary Dijkstra's algorithm that has been used in the experiment as a complete algorithm, as a subpart, one for beginning the search from starting node (the Pac-Man's current location) to the target node and the other unit for searching from the target node to the Pac-Man, in parallel, to find the shortest path between them.

For the implementation of each unit, the code is divided into two phases, 'space exploration' and 'path deduction'. Figure 7 illustrates the visualization of these phases. As the unidirectional Dijkstra uses 'tentative distance' for calculating the distance from the current node to the neighbouring nodes by slowly 'relaxing' [52] them as it performs the exploration. So, initially, the tentative distance of the current source is zero (as the distance of the current node from itself is obviously, zero) and is added as the first item to the priority queue. All the rest of the unexplored nodes have a tentative distance of infinity at this stage. Slowly the nodes are relaxed with the incremental increase in one unit distance as the depth of the generated graph tree increases, and as it is an unweighted graph (the single edge distance, the distance between adjacent nodes is considered equal to one unit). Each node in the graph tree has a tentative distance attached to it which will simply be the minimum number of nodes required to reach that node. The distance belonging to a node will keep on updating recursively as long as shorter distances to that node are found in the maze environment during the exploration phase. A two-dimensional array has been used to keep a record of these distances (each node representing a cell in the maze environment). The above phase runs simultaneously for both the unidirectional units and the termination condition is met even if any of them traverses the graph tree fully or if they share a common explored node at any point.

Next comes the path deduction phase, during which, each unit deduces the path from the graph tree, starting from the target node, and selecting the neighbour from all the neighbours with the lowest single edge distance from it recursively till the start node is found. The phase runs for both the units one by one, and the path returned by both units is combined after reversing the path returned by the reverse traversing unit to make their ends meet properly.

Finally, that complete path is reversed (because the nodes were appended to the path oppositely during the phase deduction phase) and is returned as the shortest path that Pac-Man must traverse.

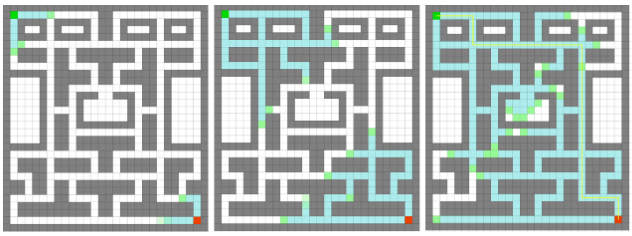


Figure 8: Graphical visualization of the path expansion and deduction phases, of the Bidirectional variant of Dijkstra's algorithm. Relative to environment of the Pac-Man game.

VI. Solution Evaluation

A. Common

Proceeding from the implementation of each of the algorithms studied, the iterative testing functionality

implemented for the Pac-Man variant, was utilised to recursively exercise the performance of each algorithm and compile each's results in the format of an external text file (extension .txt). These results were then transferred to a spreadsheet file (extension .xlsx) for processing, where the mean and standard deviation of the total nodes traversed during each test iteration, and the total time taken by Pac-Man to complete each of said tests, could then be calculated for sensibly rendering conclusions.

The data compiled and representative of each algorithm is conveniently compressed into the table featured beneath this passage, which is then referred to, for evaluating each algorithms performance relative to the Pac-Man production.

Shortest-Path Algorithm	*Nodes Traversed (Average)*	*Nodes Traversed (Standard Deviation)*	*Compilation Time Average (Seconds)*	*Compilation Time Standard Deviation (Seconds)*
BFS	*391.236*	*9.860*	*18.695*	*0.527*
A* (Manhattan)	*391.899*	*10.665*	*18.928*	*0.705*
A* (Chebyshev)	*391.924*	*10.665*	*19.136*	*0.605*
Bidirectional BFS	*391.236*	*9.860*	*19.140*	*1.298*
A* (Octile)	*391.924*	*10.665*	*19.169*	*0.799*
Bidirectional A* (Octile)	*402.528*	*11.817*	*19.214*	*0.694*
A* (Euclidean)	*395.069*	*9.613*	*19.237*	*0.556*
Dijkstra	*391.236*	*9.860*	*19.257*	*0.774*
Bidirectional A* (Chebyshev)	*397.045*	*12.075*	*19.280*	*0.675*
Bidirectional A* (Euclidean)	*404.608*	*9.197*	*19.412*	*0.456*
Bidirectional Dijkstra	*391.236*	*9.860*	*19.422*	*0.505*
Bidirectional A* (Manhattan)	*403.097*	*10.537*	*19.554*	*1.180*
IDDFS	*404.847*	*18.020*	*19.656*	*0.996*
DFS	*641.243*	*89.806*	*31.834*	*4.837*

Table 1: The results compiled for the testbed configured, displaying the performance of each of the implemented algorithms, ranked relative to the time of their operations and cumulative path-cost incurred.

B. Adam Hubble

Proceeding from the iterative nature of experimentation led for this investigation, numerical results concerning the number of nodes traversed by the Pac-Man avatar and the computational time taken for Pac-Man to navigate all traversable nodes in the grid, were compiled, for the number of traversable nodes available. Which is representative of the number of experimental runs executed (two-hundred-and-eighty-eight), per algorithm constituting to the study. Through applying concepts of statistical analysis to the self-populated dataset, mean and standard deviation statistics could be derived for better portraying a relative, performance comparison between each algorithms search(es).

*A** – in correspondence with the numerical results displayed in *Table 1,* comparatively, the A* algorithm proves to be somewhat optimal with the likes of the Breadth-First Search algorithm. As is evidenced by A* amassing an average of '391.899' nodes traversed, with an averaged compilation time of '18.928' seconds, compared to BFS's '391.236' nodes traversed and compilation time of '18.695' seconds. Noticeably, the heuristic method applied to the A* algorithm to achieve such result was Manhattan distance, which as previously told to be a good heuristic for grid-based maps, fulfils its expectations as being accurate for the horizontal and vertical restraints of Pac-Man's traversal capability. Whereby, it is observed that the Chebyshev, Octile and Euclidean distance metrics are performatively degrading; given by their increased number of nodes traversed and time taken for Pac-Man to attain all coin collectibles, on average. However, the Chebyshev and Octile variants yield a similar performance to the Manhattan variant, respective of the Euclidean variant as performatively being the worst. As does the bidirectional variants of BFS and A* when the active heuristic method concerns Octile distance. Although A* is regarded as being the provably optimal algorithm across various applications, as already explored, its suboptimality in path-cost when relative to the BFS algorithm, can be considered a defect of the heuristic function that it employs. Where acknowledged prior, it is sometimes possible for the algorithm to overestimate the actual path cost, for which the heuristic is then not considered admissible, as the path returned may not be representative of the shortest path possible. This may occur due to the unweighted nature of the grid that Pac-Man traverses through, providing that it would be possible for every node to be calculated as being the same distance from the goal.

Moreover, relating to the deviation between each experimental run of the algorithm, all heuristic variants of the A* algorithm offer a degree of consistency in their search, where unexpectedly, A* in use of the Euclidean distance metric yields the smallest deviation in nodes traversed '9.613' (nodes) and compilation time '0.556' (seconds) amassed, across all experimental runs. Generally, it can be surveyed that the A* algorithm is somewhat consistent by comparison to the other featured algorithms, where only BFS and the bidirectional variants of Dijkstra's algorithm and A* are able to compete. Such that the A* algorithm when using Euclidean distance, ranks fourth out of all other algorithms and their variants. Inevitably, the heuristic function being invocated appears to affect the performance of the A* algorithm, as is evidenced by commonality extracted from the results. That is heuristics with higher rates of consistency within their search, cause Pac-Man to traverse more sub-optimal paths, than the heuristics that demonstrate less consistency.

Providing the space complexity and unweighted nature of the game's environment, it is feasible that the A* algorithm can be a sub-optimal solution, relative to compilation time, due to the memory expense incurred from heuristic calculations in a simple environment; for which BFS can empower within, due to the lack of heuristic involvement on its search. Despite the outcome presented, the performance of the A* algorithm should not be overlooked, as within more complex and vaster environments, the algorithm claims its optimality, when aligned within the literature reviewed. As well, on average, A* shares the optimal path-cost of '391' nodes traversed,

which assumes an equivalence with BFS and both variations of Dijkstra's algorithm, relative to their completeness.

*Bidirectional A** – in continued mention of *Table 1*, A* algorithms bidirectional variant is seemingly non-optimal, for which is one of the least-performing algorithms of the study, ranking sixth (at best) and twelfth (at worst). As is backed by the numerical results compiled, Bidirectional A* when using the Octile distance heuristic, amasses an average of '402.528' nodes traversed and a compilation time of '19.214' seconds, for enabling Pac-Man to obtain all the coin collectibles in its environment. These scores are comparatively higher to the likes of BFS, A* and Dijkstra's algorithm, as previously entertained. Once again, the heuristic method active in the algorithms search vastly affects its capability of rendering an optimal path, where between each of the heuristic methods sponsored by the investigation, the deviation in average nodes traversed accumulates '7.563' nodes. Evidentially, the deviation is substantial when compared to results obtained for A*, that only deviates between an average of '3.17' nodes traversed; hence, the performance of Bidirectional A* deviates twice as much, on average, relative to the cost of the path generated. Thus, the performance of Bidirectional A* is sought unreliable and negligent to path optimisation, in the context of our investigation.

Unusually, the Octile variant of the algorithm is performatively best, in which traverses fewer nodes on average and within a shorter period; this behaviour is not anticipated, when considering that the metric is regarded optimal for diagonal-driven movements of agents. Where inversely, the better expected heuristic: Manhattan distance, is proven to be performatively worse instead, in terms of average compilation time '19.554' (seconds) and the deviation '1.180' (seconds) therein; thus, opposing all expectations of outcome, that presents no correlation with the performance of unidirectional A*. Immediately, given the deviation amount announced, it is once again plausible that the bidirectional variant of the A* algorithm too experiences heuristic overestimation, thus returning paths not considered admissible; this is sensible to assume, providing that the paths of multiple searches are ideally merged around the point of intersection. In which, with reference to prior discussions regarding the algorithm's polarity, cost overestimation and performance degradation generally, could be resultant of the intersection detection conditions adequacy; that when the two search operations do not connect central to the start node and goal node, the searches non-ideally expand more nodes and incur memory and time expenses before a point of intersection can be detected. Hence, the search time has the potential of being doubled, as revealed previously, where the path generated can also be sub-optimal, relative to cost.

Given this observation, it would be sensible to revise the termination condition of the bidirectional variant of A*, in attempt to yield performatively optimal results, as initially anticipated from its implementation. Conclusively, Bidirectional A* does not fulfil its expectations for choice in the study but it is acknowledged useful, in less object-dense areas of an environment, that Pac-Man, the game, does not entertain.

C. Jack Moorin

Dijkstra – following the iterative testing of the implemented Dijkstra's algorithm within the Pac-Man variant, the average and standard deviation of the nodes traversed by Pac-Man during each test and its total time taken were calculated along with the results for the other algorithms. The Dijkstra algorithm enabled Pac-Man to collect every coin node after traversing an average of '391.236' nodes with a standard deviation of '9.859'. This average value as well as the standard deviation of the values was shared by the BFS algorithm as well as the bi-directional implementations of both. When ranked against the results of the other implemented algorithms these four algorithms rank jointly in first place with the lowest number of total nodes traversed. This can be attributed to the fact that BFS, Dijkstra and their bidirectional implementations always output the shortest path between the start and target nodes, which is why their total nodes traversed are always the same across the large amount of completed tests and were also the lowest nodes traversed resulting from all the implemented algorithms.

While the Dijkstra implementation does manage to output the optimal path between the start and target nodes, it is let down by its performance as the algorithm will always find the shortest path between the starting node and every other map in the graph, even if the targeted node is only a short distance from the start, largely impacting the time taken to compute the algorithm. The average time taken for Pac-Man while using Dijkstra's algorithm to collect every coin node within the graph is '19.257' seconds with a standard deviation of '0.773' seconds. When ranked this places Dijkstra's algorithm as the eighth fastest algorithm to use despite the fact that it always generates the shortest path for Pac-Man to traverse. It is worth noting that the BFS algorithm, which also always finds the shortest path between the nodes and is notably less complex, is ranked first for the lowest amount of time taken to collect all the coin nodes within the graph.

In consideration of the algorithm's performance, even though Dijkstra's algorithm does manage to always find the optimum path between the start and target nodes, in practice its performance is hampered considerably by its own complexity, resulting in it being ranked below other algorithms that do not always generate the optimum path between nodes in time taken to collect all the coin nodes as they are able to make up for this by being less computationally expensive to compute than Dijkstra.

Iterative Deepening Depth-First Search – conversely to Dijkstra's algorithm, IDDFS does not always output the shortest path between nodes and is in fact ranked as the 13th best algorithm for total nodes traversed (one from last) due to the sub-optimal paths it creates, having an average of '404.847' total nodes traversed per test, with a standard deviation of '18.019'. The only algorithm in fact IDDFS managed to outperform in total number of nodes traversed is its non-depth limited Depth-First Search algorithm counterpart, which has the anomalously high number of total nodes traversed of '641.243' with a standard deviation of '89.805'. If this performance of the non-depth limited variant of IDDFS is in fact considered to be anomalously high and DFS removed from consideration as a pathfinding algorithm IDDFS is the least successful of all the implemented algorithms are finding the optimum path between Pac-Man and the targeted node.

IDDFS is also ranked as 13th in the average time taken to traverse the graph collecting all the coin nodes with an

overall average time of '19.655' seconds and a standard deviation of '0.996' seconds. Depth-First Search is again the lowest ranked algorithm due to its anomalously high average time of '31.833' seconds, with a standard deviation of '0.996' seconds. If the performance of DFS is again considered to be anomalous here IDDFS becomes the longest of all the algorithms to run as well as being the least successful in finding the optimal path between the start and target nodes.

In consideration of the algorithm's overall performance IDDFS was both the least successful algorithm in generating an optimal path between the start and target nodes and took the longest of all the algorithms to collect every coin node. While the non-depth limited DFS algorithm performed even worse than IDDFS, its own performance is so much worse than that of the algorithms that it is considered to be anomalous, leaving IDDFS as the worst of all the algorithms.

D. Arpit Sharma

In this Pac-Man variant, the Pac-Man is supposed to collect all the coins present in the environment, in as little time and nodes traversed as possible. The movement of Pac-Man instead of being manually controlled is handled by the algorithm being tested. The test is run for each algorithm for the number of empty cells in the maze with the Pac-Man starting from each of them, iteratively. The time taken and the nodes traversed on each iteration are stored in the respective files of the algorithms being run, named after the algorithm, and are stored in the results folder. This forms the benchmark, made, and then used for the experiment.

Nodes traversed – both the implemented Bidirectional BFS algorithm and the Bidirectional Dijkstra algorithm tested to have on average traversed nodes to be 391.236 with an equal standard deviation of fewer than 10 nodes. This means that the Pac-Man treaded this number of nodes in the maze to collect all coins, on average. This also happens to be the lowest number of average nodes traversed in the experiment, and the same results are shared by their unidirectional variants as well. This can be said from the results that these algorithms follow the shortest path, which is one of the main aims of the experiment. Also, since their respective unidirectional variants also gave exactly the same results (both the average and the standard deviation), it means there is no difference in the final path calculated by the unidirectional and bidirectional variants. This reason is supposed to be the fact that the unidirectional subparts of the bidirectional variants explore many nodes before meeting each other in the middle that there is no room for error left, unlike bidirectional A* variants.

Time Taken to collect all coins – but the average time taken for collecting all the coins for the bidirectional BFS variant is 19.140 seconds, which is little more than taken by the unidirectional BFS (18.695 seconds). Theoretically, it was expected to be quicker than the unidirectional one as it possesses much less space and time complexity, but the results are an outcome of the reason that the coins in this Pac-Man variant are adjacent to each other. Therefore, the next closest coin, the Pacman must collect is often, very next to it and sometimes they just have a single empty cell between them. As the Bidirectional Breadth-First Search explores the nodes layer by layer, from both the starting node (Pac-Man's position) and the target position (next closest coin, lying very next to it), it ends up exploring four nodes (two nodes per its each unidirectional unit) plus an intersecting node checking (which in itself is an expensive operation) to calculate the path, as compared to unidirectional BFS, which just explores two nodes. Even when there is an empty cell between the Pac-Man and the coin to be collected (which happens very often), the bidirectional variant must perform four nodes (two from each end), and an intersection check. Although the unidirectional BFS also must explore four nodes (two on both sides), it does not have to perform any intersection checking.

Hence, in these cases, the bidirectional variant expends much more computational power and time as compared to the unidirectional algorithm leading to poor performance in terms of average time taken to collect all coins. The same reason can be extended to explain the poor performance of Bidirectional Dijkstra, at 19.422 seconds when compared to its unidirectional equivalent, which is 0.165 seconds quicker than it on an average. Therefore, the bidirectional variants seem less efficient than their unidirectional variants when the target is too close to the source.

Algorithm completeness – it was seen whether the algorithms get stuck in any of the iterations, meaning that they traverse the part of the generated graph without the goal and are unable to reach the goal. As expected, both the algorithms did not glitch anytime proving that the algorithms are 'complete' algorithms and hence, unfailingly finds the target every time. It makes them suitable for making glitch-free video games, the importance of which has been discussed in earlier sections.

VII. Conclusion

A. Exercising Performance

With respect to the benchmarks used to measure the performance of the implemented algorithms, the total number of nodes traversed by Pac-Man during a test iteration is a good metric to use to compare and contrast the ability of each of the algorithms to output the optimum path between the provided nodes. For example, when analysing the compiled results table, it can be easily deduced that the DFS algorithm is considerably less efficient at finding the optimum path than the other algorithms as the paths it output were long enough to cause Pac-Man to traverse over one and a half times as many nodes within the maze than any of the other algorithms.

The second of the implemented metrics, the length of time in seconds of each test, is used to infer the overall efficiency of the algorithm at generating a path for Pac-Man to take through the maze. This metric unfortunately is not a wholly accurate representation of an algorithms computational efficiency as it also includes the time taken for Pac-Man to traverse the paths output by the algorithm and is therefore also impacted by its ability to find an optimum path. A better implementation of a time-based metric could have been to measure the total time taken during the computation of each algorithm only to remove output path length as a factor. Despite this, the metric does still provide a good idea of each algorithm's computational complexity, as a notable difference can be seen between the length of time taken by less computationally expensive algorithms, such as BFS, and those that were more computationally expensive, such as Dijkstra.

B. Findings

Adam Hubble – in response to the numerical findings of the iterative, experimental procedure conducted, it is inevitable that the Breadth-First Search (BFS) algorithm proves to be performatively optimal compared to all other algorithms trialled in the investigation. Averaging '391.236' nodes traversed, within a compilation time equating to '18.695' seconds. Both of which metrics are the least significant by value; BFS also presents the shared-most consistency in nodes traversed across all iterations of the experimentation and ranks second for compilation time consistency also. Thus, it can be reasoned that BFS not only, always computes the shortest possible path between the position of Pac-Man, the start node, and a specified goal node, but is also the candidate algorithm for enabling Pac-Man to traverse efficiently throughout the maze type of environment configured. This is assumed to be in cause of the algorithms low computational-overhead, that enables compilation times to be curtailed and for Pac-Man to then initiate traversal behaviours faster.

Competing with BFS, the A* algorithm when applying the Manhattan distance heuristic, is also proven to be a successful candidate for the path optimisation domain but is rendered sub-optimal, for the investigation conducted. This is presumed to be in effect of the computational expense arisen by the calculations and appliance of heuristics in its search, which within smaller and simpler environments, are consequential to its performative capability. Thereby, one can assume a different outcome for environments with increasingly complex structural arrangements, and with more obstacle-dense areas; providing what is already known about the optimality of the A* algorithm in the field.

Moreover, it is imperative to note that all bidirectional variants of the algorithms investigated are less performatively optimal, when compared to their unidirectional counterparts. This is evidenced by each algorithm computing paths that are either non-optimal or are optimal but are instead compiled within longer periods. Each algorithms performance is assumed to be affected by a collective of factors, that concern the start and goal nodes being too close to each other, thus rendering a bidirectional search expensive and impractical. As well, as the performance implications concerned with detecting intersecting nodes, in each search programs path.

In continued discussion of performative degradation, evidently Depth-First Search (DFS) and its variant: Iterative-Deepening Depth-First Search (IDDFS), prove to be the worst algorithms performatively; as is given by the average number of nodes traversed and compilation times amassed, that are significantly greater than all other solutions trailed. We can declare such performance to be relative to the deepening focuses of either algorithm, that favours distance in their search and renders them incomplete; as the production operates with a grid and not explicitly a graph tree, the algorithm consequently compiles a path using an elongated pattern within its search. Thus, Pac-Man traverses more nodes, for which is why its application is not recommended.

Jack Moorin – the results compiled by the Pac-Man variants iterative testing show that the most successful of all the implemented algorithms was clearly the Breadth-First Search algorithm. BFS not only always output the optimum path between Pac-Man and the target node but was also the fastest of all the algorithms to run likely due to it having very little computational complexity. It is likely due to this low computational complexity that it was the most successful of the algorithms as other methods which always output the shortest path between nodes, such as Dijkstra, were outperformed by methods which do not but made up for the time lost navigating unnecessarily long paths by being notably less complex and faster to run. It is worth noting however, that BFS is only able to find the optimum path between nodes within an unweighted graph as it search process is carried out on the basis of exploring the nodes neighbouring those that have already been searched without any considerations made for weight.

The results also clearly show that the least successful of all the implemented algorithms was the Depth-First Search algorithm which output the longest paths between nodes and also took the longest amount of time for Pac-Man to collect every coin node. Given that the DFS algorithm has the same computational complexity as the BFS algorithm, the fastest of the implemented algorithms, the amount of time taken by Pac-Man to complete each test can largely be attributed to the length of the paths output as opposed to the algorithm being incredibly complex.

Arpit Sharma – this experiment concludes that the bidirectional variants of the Breadth-First Search algorithm and Dijkstra algorithm are as promising and reliable as their unidirectional equivalents in an unweighted grid, in terms of successfully finding the shortest path, that relates to the algorithm's completeness. But the bidirectional algorithms seem to lag behind their unidirectional variants in terms of time, when there is just a 'single edge distance' between the 'starting node' and the 'target node', or if they just have a single node between them in the graph tree.

As the size of the map is very small and simple, it can be afforded to explore all possible nodes using a simple algorithm like BFS algorithm or Dijkstra algorithm. This is supposed to be the reason behind the best performance of the BFS algorithm in the experiment. But, if the complexity of the map would increase, the performance is expected to worsen and that of more complex algorithms, which make use of a variety of heuristics like A* is supposed to go up.

The performance of the DFS algorithm and IDDFS algorithm is worse among all other algorithms tested. It is believed to be because of the fact they are not optimal algorithms. As the complexity of the map would increase, their chances of getting trapped inside an infinite loop is expected to increase, leading to glitches, if used for video games.

C. Future Work

Following the completion of the study, plans and other ideas can be formulated regarding the completion of future work, which continues to expand upon the research completed as part of this study. As discussed in the literature review, the Shortest-Path Problem (SSP) is an 'important problem' [11] with many applications, such that further work is warranted into the development and testing of novel algorithms and approaches, in attempt to solve the problem more effectively. As demonstrated within the study, numerous methods already exist that can be used to compile a path between two given nodes in a graph, however, not all of these methods are assured to consistently compute the optimal path between the nodes; specifically those

identified to often face an overwhelming decrease in efficiency, when faced with increasing space and time complexities.

The development of an algorithm that could consistently compile an optimal path whilst not suffering from the same performance issues of other algorithms, such as Dijkstra, would be a notable achievement and part of a further branch of study in continuation of this project. Such that this study aims to proof some of the most popular and contemporary attempts led into pathfinding, by researchers in the fields of CI and AI alike.

D. Challenging Encounters

During the completion of this study, one of the leading challenges confronted was the successful collaboration of three geographically distanced individuals, each of whom was attempting to contribute to the common work of the study submitted. Due to the intervention of the COVID-19 pandemic, the members of the study were unable to physically meet in-person, to collaborate and express ideas thoughtfully, and instead had to make use of online messaging tools to ground a means of communication. This medium of collaboration has undoubtedly had impact on the quality of communication between group members, that often led to instances of miscommunication and resultant time wasted on group members completing collaborative tasks, that were formerly tackled by a group member in advance of notice being reciprocated.

Another difficulty encountered was resultant of the geographical distance that separates each study member, such that each member would attempt to work on the same document for the project report, or the code solution comprising the implementation of the Pac-Man variant, simultaneously. This once more, contributed to time being wasted, due to the miscommunications of group members, that often distorted the works progression, relative to acknowledging the components of the work that were agreed complete and outstanding. The impacting factor that this contest had on the project, however, was the amount of time that was wasted, from continually merging the works of each study member into a collaborative document.

For the foreseeable future, it would be recommended to utilize cloud-based software's that entertain collaborative efforts, to mitigate the extreme of time being wasted and to resultingly hasten a projects progression. With reference to each study member's investment into the investigation led, refer to *Appendix A* and *Appendix B* accordingly.

Despite all the difficulties listed, collectively, we believe that the quality of work submitted exceeds the expectations supposed for an investigation of a similar scope. Thus, we remark this work as a model for future works in the field of pathfinding.

VIII. Acknowledgement

For the relevant staff at De Montfort University, we express our gratitude for the guiding support and extending allowances for this project's completion, namely, Prof. Aladdin Ayesh and Dr. Ioannis Kypraios. Additionally, we extend our appreciation for the facilities and resources available to us, during these unprecedented times of the COVID-19 pandemic; the members of the nation's health service, the NHS, are undoubtedly courageous. May God bless the souls of those who have lost their lives during this global catastrophe.

Adam Hubble – I would like this opportunity to exercise my appreciation for Shirley Lagaran Gocotano, who is emotionally closest to me currently. Without your continued support and investment into my interests and endeavours, would this project reveal a similar quality?

Jack Moorin – I would personally like to extend my thanks to the other group members of this project, Adam Hubble and Arpit Sharma for their work on this project. Without the shared input and discussion from these individuals, the quality of the study would have been notably worse than what the completed state showcases.

Arpit Sharma – I would like to thank both of my groupmates, Adam Hubble, and Jack Moorin for showing team spirit and hard work throughout the experiment, which led to the completion of this study.

References

[1] IEEE (2021) What is Computational Intelligence?. [Online] IEEE Computational Intelligence Society. Available from: https://cis.ieee.org/about/what-is-ci [Accessed: 24/05/21]

[2] Duch, W., 2007. What is Computational Intelligence and where is it going?. In Challenges for computational intelligence (pp. 1-13). Springer, Berlin, Heidelberg.

[3] IBM (2020) Artificial Intelligence (AI). [Online] IBM Cloud Education. Available from: https://www.ibm.com/uk-en/cloud/learn/what-is-artificial-intelligence [Accessed: 24/05/21]

[4] IFAC Proceedings Volumes (2021) A brief introduction to artificial intelligence. [Online] Institut Universitaire de France. Available from: https://www.sciencedirect.com/science/article/pii/S1474667015330226 [Accessed: 24/05/21]

[5] IEEE (2011) Artificial Intelligence. [Online] IEEE Xplore. Available from: https://ieeexplore.ieee.org/document/5941871 [Accessed: 24/05/21]

[6] Brunette, E.S., Flemmer, R.C. and Flemmer, C.L., 2009, February. A review of artificial intelligence. In 2009 4th International Conference on Autonomous Robots and Agents (pp. 385-392). Ieee.

[7] Gallagher, M. and Ryan, A., 2003, December. Learning to play Pac-Man: An evolutionary, rule-based approach. In The 2003 Congress on Evolutionary Computation, 2003. CEC'03. (Vol. 4, pp. 2462-2469). IEEE.

[8] Doostmohammadian, M., Pourazarm, S. and Khan, U.A., 2014, April. Distributed algorithm for shortest path problem via randomized strategy. In Proceedings of the 11th IEEE International Conference on Networking, Sensing and Control (pp. 463-467). IEEE.

[9] Tao, J.Y., Hua, Z.P., Bo, Y.Q. and Hui, F.Z., Study on Simplified Algorithm for the Shortest Path. In 2009 International Conference on Information Engineering and Computer Science.

[10] Muniandy, M.A., Mee, L.K. and Ooi, L.K., 2014, October. Efficient route planning for travelling salesman problem. In 2014 IEEE Conference on Open Systems (ICOS) (pp. 24-29). IEEE.

[11] Ortega-Arranz, H., Llanos, D.R. and Gonzalez-Escribano, A., 2014. The shortest-path problem: Analysis and comparison of methods. Synthesis Lectures on Theoretical Computer Science, 1(1), pp.1-87.

[12] Sanders, P., Schultes, D. and Vetter, C., 2008, September. Mobile route planning. In European Symposium on Algorithms (pp. 732-743). Springer, Berlin, Heidelberg.

[13] Barceló, J., Codina, E., Casas, J., Ferrer, J.L. and García, D., 2005. Microscopic traffic simulation: A tool for the design, analysis and evaluation of intelligent transport systems. Journal of intelligent and robotic systems, 41(2), pp.173-203.

[14] Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D. and Werneck, R.F., 2016. Route planning in transportation networks. In Algorithm engineering (pp. 19-80). Springer, Cham.

[15] Rétvári, G., Bíró, J.J. and Cinkler, T., 2007. On shortest path representation. IEEE/ACM Transactions on Networking, 15(6), pp.1293-1306.

[16] Böhm, C., Kny, E., Emde, B., Abedjan, Z. and Naumann, F., 2011, March. Sprint: ranking search results by paths. In Proceedings of the 14th International Conference on Extending Database Technology (pp. 546-549).

[17] Cui, X. and Shi, H., 2011. Direction oriented pathfinding in video games. International Journal of Artificial Intelligence & Applications, 2(4), p.1.

[18] Pan, T. and Pun-Cheng, S.C., 2020. A Discussion on the Evolution of the Pathfinding Algorithms.

[19] Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. Numerische mathematik, 1(1), pp.269-271.

[20] Tao, J.Y., Hua, Z.P., Bo, Y.Q. and Hui, F.Z., Study on Simplified Algorithm for the Shortest Path. In 2009 International Conference on Information Engineering and Computer Science.

[21] Evans, J., 2017. Optimization algorithms for networks and graphs. CRC Press.

[22] Cui, X. and Shi, H., 2011. A*-based pathfinding in modern computer games. International Journal of Computer Science and Network Security, 11(1), pp.125-130.

[23] Sabri, A.N., Radzi, N.H.M. and Samah, A.A., 2018, April. A study on Bee algorithm and A∗ algorithm for pathfinding in games. In 2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE) (pp. 224-229). IEEE.

[24] Zikky, M., 2016. Review of A*(A star) navigation mesh pathfinding as the alternative of artificial intelligent for ghosts agent on the Pacman game. EMITTER International Journal of Engineering Technology, 4(1), pp.141-149.

[25] Luo, W., 2018, January. PAC-MAN Game Based on SAPF Algorithm. In 2018 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS) (pp. 686-689). IEEE.

[26] Ortega Vargas, Á.J., Serrano, J.E., Castellanos Acuña, L. and Martinez-Santos, J.C., 2020. Path Planning for Non-Playable Characters in Arcade Video Games using the Wavefront Algorithm.

[27] Hui, Y.C., Prakash, E.C. and Chaudhari, N.S., 2004, November. Game ai: artificial intelligence for 3d path finding. In 2004 IEEE Region 10 Conference TENCON 2004. (pp. 306-309). IEEE.

[28] Valenzano, R. and Xie, F., 2016, February. On the completeness of best-first search variants that use random exploration. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 30, No. 1).

[29] van Opheusden, B. and Ma, W.J., 2019. Tasks for aligning human and machine planning. Current Opinion in Behavioral Sciences, 29, pp.127-133.

[30] Zhou, M. and Gao, N. (2019) Research on Optimal Path based on Dijkstra Algorithms. 10.2991/icmeit-19.2019.141.

[31] Panda, M. and Mishra, A., 2018. A Survey of Shortest-Path Algorithms. International Journal of Applied Engineering Research, 13(9), pp.6817-6820.

[32] Martell, V. and Sandberg, A., 2016. Performance Evaluation of A* Algorithms.

[33] Kapi, A.Y., Sunar, M.S. and Zamri, M.N., 2020. A review on informed search algorithms for video games pathfinding. International Journal, 9(3).

[34] Sabri, A.N., Radzi, N.H.M. and Samah, A.A., 2018, April. A study on Bee algorithm and A∗ algorithm for pathfinding in games. In 2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE) (pp. 224-229). IEEE.

[35] He, Z., Shi, M. and Li, C., 2016, June. Research and application of path-finding algorithm based on unity 3D. In 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS) (pp. 1-4). IEEE.

[36] Appaji, N.S.D., 2020. Comparison of Searching Algorithms in AI Against Human Agent in the Snake Game.

[37] Carina Krafft (2021) Implementation and comparison of pathfinding algorithms in dynamic 3D space. [Online] University of Applied Sciences Hamburg. Available from: https://users.informatik.haw-hamburg.de/~schumann/BachelorArbeitCarinaKrafft.pdf [Accessed: 24/05/21]

[38] Barnouti, N.H., Al-Dabbagh, S.S.M. and Naser, M.A.S., 2016. Pathfinding in strategy games and maze solving using A* search algorithm. Journal of Computer and Communications, 4(11), p.15.

[39] Rafiq, A., Kadir, T.A.A. and Ihsan, S.N., 2020, February. Pathfinding Algorithms in Game Development. In IOP Conference Series: Materials Science and Engineering (Vol. 769, No. 1, p. 012021). IOP Publishing.

[40] Nosrati, M., Karimi, R. and Hasanvand, H.A., 2012. Investigation of the*(star) search algorithms: Characteristics, methods and approaches. World Applied Programming, 2(4), pp.251-256.

[41] Python Tutorial (2021) OOP Python Tutorial. [Online] Python. Available from: https://www.python-course.eu/python3_magic_methods.php [Accessed: 24/05/21]

[42] Zhang, D.H. and Chen, Y.M., 2014, July. Study on the bidirectional A∗ algorithm based on avoiding risk. In 2014 International Conference on Audio, Language and Image Processing (pp. 115-119). IEEE.

[43] Saian, P.O.N., 2016, October. Optimized A-Star algorithm in hexagon-based environment using parallel bidirectional search. In 2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE) (pp. 1-5). IEEE.

[44] Whangbo, T.K., 2007, June. Efficient modified bidirectional A* algorithm for optimal route-finding. In International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (pp. 344-353). Springer, Berlin, Heidelberg.

[45] Monzonís Laparra, D., 2019. Pathfinding algorithms in graphs and applications.

[46] Python (2001), Welcome to Python. [Online] Python Software Foundation. Available from: https://www.python.org/ [Accessed 18/04/21]

[47] Visual Studio (2021), Visual Studio IDE. [Online] Microsoft. Available from: https://visualstudio.microsoft.com/ [Accessed 18/04/21]

[48] Pygame (2021), *News*. [Online] pygame. Available from: https://www.pygame.org/news [Accessed 18/04/21]

[49] Piskel (2021), *Piskel App*. [Online] piskelapp. Available from: https://www.piskelapp.com/ [Accessed 18/04/21]

[50] Leiserson, C.E. and Schardl, T.B., 2010, June. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures (pp. 303-314).

[51] Chen, J.C., 2003. Dijkstra's shortest path algorithm. Journal of Formalized Mathematics, 15(9), pp.237-247.

[52] Moore, E.F., 1959. The shortest path through a maze. In Proc. Int. Symp. Switching Theory, 1959 (pp. 285-292).

[53] Coppin, B., 2004. Artificial intelligence illuminated. Jones & Bartlett Learning.

[54] Bader, D.A. and Madduri, K., 2006, August. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In 2006 International Conference on Parallel Processing (ICPP'06) (pp. 523-530). IEEE.

[55] Rahayuda, I.G.S. and Santiari, N.P.L., 2021, February. Dijkstra and Bidirectional Dijkstra on Determining Evacuation Routes. In Journal of Physics: Conference Series (Vol. 1803, No. 1, p. 012018). IOP Publishing.

[56] Akanmu, T.A., Olabiyisi, S.O., Omidiora, E.O., Oyeleye, C.A., Mabayoje, M.A. and Babatunde, A.O., 2010. Comparative study of complexities of breadth-first search and depth-first search algorithms using software complexity measures. In Proceedings of the World Congress on Engineering (Vol. 1).

[57] Lim, K.L., Seng, K.P., Yeong, L.S., Ang, L.M. and Ch'ng, S.I., 2015. Uninformed pathfinding: A new approach. Expert systems with applications, 42(5), pp.2722-2730.

[58] X Cui & H Shi (2011) 'Direction oriented pathfinding in video games' *International Journal of Artificial Intelligence & Applications*, NA, 2011

[59] N Lipovetzky, M Ramirez & H Geffner (2015) 'Classical Planning with Simulators: Results on the Atari Video Games' *ICAPS Conference*

[60] Cui, X. and Shi, H. (2011) 'A*-based Pathfinding in Modern Computer Games' IJCSNS International Journal of Computer Science and Network Security, VOL 11, NA, January 2011

[61] S D H Permana, K B Y Bintoro, B Arifitama & A Syahputra (2018) 'Comparative Analysis of Pathfinding Algorithms A*, Dijkstra and BFS on Maze Runner Game', *International Journal of Information System & Technology*, Universitas Trilogi

[62] E. W. Dijkstra (1959) 'A Note on Two Problems in Connexion with Grpahs' *Numerische Mathematik 1*, NA, NA, NA, pg 269 – 271

[63] Y Deng, Y Chen, Y Zhang & S Mahadevan (2012) 'Fuzzy Dijkstra algorithm for shortest path problem under uncertain enviroment' *Applied Soft Computing*, Southwest University, Shanghai Jiao Tong University, Vanderbilt University, NA, NA

[64] P Sanders & K Mehlhorn (2008) 'Chapter 10. Shortest Path' *Algorithms and Data Structures: The Basic Toolbox*, NA, NA

[65] M Seo, H Iida, J W.H.M. Uiterwijk (2001) 'The PN*-search algorithm: Application to tsume-shogi' *Artificial Intelligence Vol 129*, Shizuoka University, NA, NA, NA, pg 253-27

APPENDICES

Appendix A:

	Participant	Time	Date	Activity
Example	AS	1	10/03/2021	Pac-Man game repository setup
	JM	8	10/03/2021	Setting up original Pac-Man for group use
	AH	8	10/03/2021	Setting up original Pac-Man for group use
	AS	8	10/03/2021	Setting up original Pac-Man for group use
	AH	2	11/03/2021	Group management file setup
	AH	1	12/03/2021	Group management file setup
	AH	1	16/03/2021	Visualisation tool scouting
	AH	1	07/04/2021	Group management file maintenance
	AH	1	19/04/2021	Group management file maintenance
	AH	8	14/05/2021	Scoring method implementation
	AH	2	14/05/2021	Debug mode method implementation
	JM	8	14/05/2021	Setting up Pac-Man
	AS	8	13/05/2021	Setting up Pac-Man
	AH	5	15/05/2021	A* library solution implementaton
	AH	2	15/05/2021	A* solution implementation
	AH	4	15/05/2021	Conference paper format and styling
	JM	8	15/05/2021	Iterative testing benchmark setup
	JM	4	15/05/2021	Designed Pac-Man and ghost icons
	JM	8	16/05/2021	Joint implementation of Dijkstra with AS
	AS	8	16/05/2021	Joint implementation of Dijkstra with JM
	AH	8	16/05/2021	A* solution implementation
	AH	8	16/05/2021	A* solution implementation
	AS	3	17/05/2021	Shortest from all algorithm implementation
	JM	8	17/05/2021	Implementation of IDDFS algorithm
	AH	4	17/05/2021	A* solution implementation
	AH	2	17/05/2021	Refactor A* library solution implementation
	AH	8	17/05/2021	Refactor A* solution implementation
	AS	8	17/05/2021	Bidirectional BFS solution implementation
	AS	8	18/05/2021	Bidirectional BFS solution implementation
	AH	8	18/05/2021	Implemented A* heuristic catalogue
	AS	9	18/05/2021	Bidirectional Dijkstra solution implementation
	AH	8	19/05/2021	Bidirectional A* solution implementation
	AH	2	19/05/2021	Iterative testing benchmark refactor
	AS	8	19/05/2021	Bidirectional A* solution implementation with AH
	JM	1	19/05/2021	Bidirectional A* solution implementation with AH
	JM	8	19/05/2021	Conference paper common contribution
	AH	12	20/05/2021	Conference paper common contribution
	AS	10	20/05/2021	Conference paper common contribution
	JM	8	20/05/2021	Conference paper common contribution
	AH	12	21/05/2021	Conference paper common contribution
	AS	12	21/05/2021	Conference paper common contribution
	JM	12	21/05/2021	Conference paper common contribution
	AS	5	22/05/2021	Travelling Salesperson Problem (TSP) attempt
	AH	15	22/05/2021	Conference paper common contribution
	AS	10	22/05/2021	Conference paper common contribution
	JM	12	22/05/2021	Conference paper common contribution
	AH	15	23/05/2021	Conference paper common contribution
	AS	15	23/05/2021	Conference paper common contribution
	JM	8	23/05/2021	Conference paper common contribution
	AH	8	24/05/2021	Presentation creation and finalisation
	JM	8	24/05/2021	Presentation creation and finalisation
	AS	8	24/05/2021	Presentation creation and finalisation

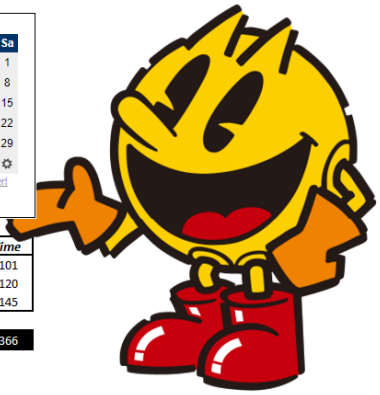
< > May 2021

Su	Mo	Tu	We	Th	Fr	Sa
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2			

Vote on a built-in Date Picker!

Name	Initials	Time
Jack Moorin	JM	101
Arpit Sharma	AS	120
Adam Hubble	AH	145

Total	366



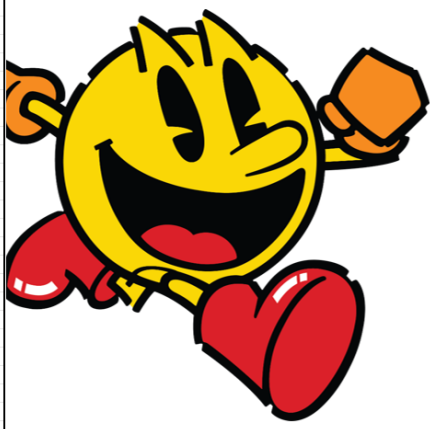


Figure 9: 'ProjectManagementACI' spreadsheet file (.xlsx), capturing the 'Activity Log' worksheet used to record the working commitments and time investments of each member of the study.

Appendix B:

	Participant(s)			Date	Period	Note	Platform
Example	AH	AS	JM	03/03/21	1	Consolidate project ideas and choice of algorithms	Blackboard
	AH	AS	JM	09/03/21	1	Discussed and visualised choice of algorithms and benchmarking techniques	Blackboard
	AH	AS	JM	16/03/21	1	Discussed algorihtmic visualisation tools and reviewed choice of algorithms	Blackboard
	AH	AS	JM	24/03/21	1	Discussed algorithmic implementation progression and support framework for intended algorithms	Blackboard
	AH	AS	JM	31/03/21	1	Discussed the contents of the independant literature reviews	Blackboard
	AH	AS	JM	07/04/21	1	Discussed algorithmic implementation progression	MS Teams
	AH	AS	JM	14/04/21	1	Discussed algorithmic implementation progression	MS Teams
	AH	AS	JM	21/04/21	1	Discussed algorithmic implementation progression	MS Teams
	AH	AS	JM	28/04/21	1	Discussed algorithmic implementation progression	MS Teams
	AH	AS	JM	05/05/21	1	Discussed algorithmic implementation progression	MS Teams
	AH	AS	JM	12/05/21	1	Discussed algorithmic implementation progression	MS Teams
	AH	AS	JM	15/05/21	8	Finalising algorithmic implementations of the Pac-Man application	MS Teams
	AH	AS	JM	16/05/21	8	Finalising algorithmic implementations of the Pac-Man application	MS Teams
	AH	AS	JM	17/05/21	8	Finalising algorithmic implementations of the Pac-Man application	MS Teams
	AH	AS	JM	18/05/21	8	Finalising algorithmic implementations of the Pac-Man application	MS Teams
	AH	AS	JM	19/05/21	8	Finalising algorithmic implementations of the Pac-Man application	MS Teams
	AH	AS	JM	20/05/21	8	Finalising algorithmic implementations of the Pac-Man application	MS Teams
	AH	AS	JM	21/05/21	8	Progressing with the collaborative passages of the conference paper	MS Teams
	AH	AS	JM	22/05/21	12	Progressing with the collaborative passages of the conference paper	MS Teams
	AH	AS	JM	23/05/21	15	Progressing with the collaborative passages of the conference paper	MS Teams
	AH	AS	JM	24/05/21	4	Progressing with the projects presentation preparation	MS Teams
	AH	AS	JM	25/05/21	4	Project submission overview	

Total	102

< > March 2021

Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8			

Vote on a built-in Date Picker!

Figure 10: 'ProjectManagementACI' spreadsheet file (.xlsx), capturing the 'Meeting Backlog' worksheet used to record the collaborative efforts and assemblies of the study members, with relevance to each's nature.