Optimizing Agent Navigation in Battlecode

Andrew Hoyle
University of Minnesota
Minneapolis, MN, USA
hoyle020@umn.edu

Jack Lee
University of Minnesota
Minneapolis, MN, USA
lee02802@umn.edu

Katya Borisova
University of Minnesota
Minneapolis, MN, USA
boris040@umn.edu

Abstract

Battlecode is a yearly AI programming competition hosted by MIT. In Battlecode, students compete to write agents working in partially observable compete bound environments. Battlecode limits the amount of compute each agent has by setting a ceiling bound on the amount of Java bytecode instructions the agent can execute. Pathfinding algorithms are computationally expensive and consume a large part of this limit. An educated decision on which pathfinding algorithms to use are invaluable in the competition. In addition, using Java bytecode as a measurement of algorithm performance is better than the current standard of timing execution. From this, it was found that A-Star was the most efficient complete algorithm and Bug was the most efficient if completeness was not required.

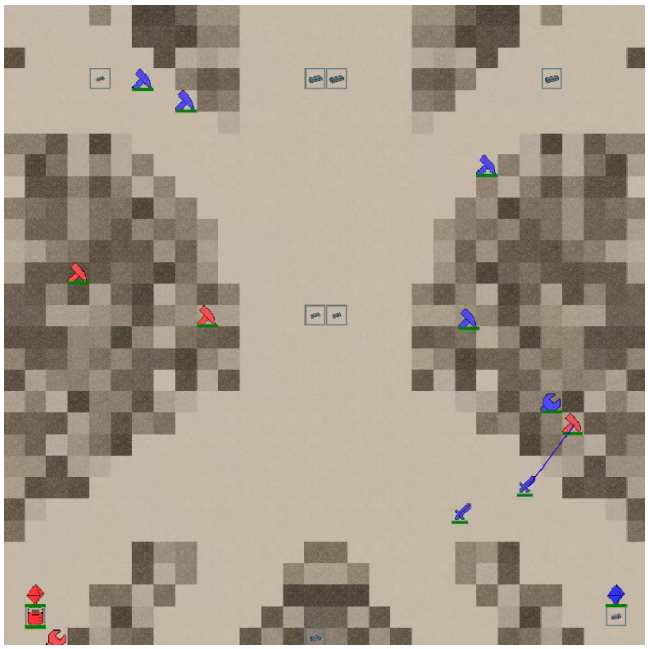


Figure 1. Battlecode Game

1 Introduction

Battlecode is an annual programming contest in which teams compete to write AI to play a real-time strategy game akin to popular games like Starcraft. The environment is a partially observable multi-agent grid-based map, with each cell containing either an agent, an obstacle or nothing. These obstacles can consist of walls which the agent can not move on, rubble which can slow down the movement of an agent, or currents which will push an agent towards an arbitrary direction. Further, visibility can be limited by clouds or weather that occurs on the map. A core win condition that is universally present in every yearly version of Battlecode is dependent on which team has the most resources. Agents must navigate the map to collect these resources, making effective pathfinding vital to highly successful teams.

A major limitation on an agent's ability to achieve accurate pathfinding is a tight bound on computational capabilities. The competition has a manufactured limitation on the number of Java virtual machine (JVM) bytecode instructions each agent can execute on each of its turns. JVM bytecode is the intermediary language that Java compiles to for execution on the virtual machine. Therefore, this project will focus on the comparison, analysis, and optimization of pathfinding algorithms in terms of JVM bytecode as they apply to Battlecode. Much of the bytecode limit is used every turn for path planning purposes. Consequently, this introduces difficulties elsewhere in the agent's programming as it cannot afford to execute nonessential algorithms. This motivates the discovery of the cheapest possible satisfactory pathfinding algorithm.

2 Problem

The bytecode limit make this an interesting problem, as path planning is not the only aspect of the game. Robots must be able to also execute their actions. Actions can include gathering and allocating resources, attacking and defending against enemy robots, and movement. Therefore, part of the game strategy is trying to figure out how to best partition the bytecode usage amongst all the necessary actions that a robot needs to take. Furthermore, no information can be stored from round to round in the game, meaning that after a path planning algorithm is run, the robot cannot access the path it finds the following round. Therefore, if a robot is trying to navigate across the map, the path planning algorithm needs to be re-run every round, which is extremely costly. There does exist very limited communication, fostered by a 2D integer array, accessible by all the other robots. However, there is not nearly enough space for each robot to store its path in the array, not to mention, there are more important things that need to be stored in the array. Furthermore, paths

would likely need to change round-to-round anyways, as other robots on the map might get in the way.

This problem is also particularly useful as it is an uncommon way to evaluate the performance and efficiency/optimally of path planning algorithms. Previous path planning research papers such as Zarembo[13] and Krishnaswamy[16] tend to only focus on traversed nodes and execution time. Execution time is only useful in relative terms between the algorithms due to configuration of the experiment machine and processor architecture. Considering only traversed nodes ignores the cost of the underlying data structure powering the algorithm. This effectively ignores the log term of the priority queue in an algorithm such as A-Star. Bytecode evaluates the algorithm in absolute terms and takes into account the underlying costs of each data structures and their respective operations. It also allows transferability of results between machines because bytecode instruction count is a constant cost post compilation.

3 Congested Video Game Environments

Pathfinding in video games is a crucial component of games. It becomes a bigger issue when team-based pathfinding is necessary for proper coordination. Hang Ma's paper goes into greater detail on how this works in heavily congested and dynamic environments, where aspects like these are vital to determining challenging unit movement. The article further details ways to reduce an environment like this via combinatorial problems to help provide a higher percentile of groups of units pathing to a given destination [18].

Consider a game like Age of Empires, where different units require different forms of micro- and macro-grouping [18]. The difficulty arises when multiple move actions are requested in congested regions, requiring units to navigate around obstacles in order to attack or defend against the enemy. Instead of searching for the direct path, searches can be done via conflicts, so the path consisting of the fewest conflicts will be taken by individual units. The article helps provide solutions that would be worthwhile to examine for utility within Battlecode. Such approaches could help showcase an environment where communication is easily accessible, such as the one from the article, and multi-agent pathfinding can be performed efficiently. But in an environment like Battlecode, where communication is difficult and costly, it might not be so possible.

4 Pathfinding in Game Development

Other formats of pathfinding are further explored in the article by Sara Lutami Pardede, which dives into great detail about different pathing approaches used widely in gaming. The most notable search approach was the Theta* (Theta-star) algorithm. Theta* is useful because it only considers vertices or positions that are within the line of sight of the agent [19]. This makes Theta* incredibly useful in a Battlecode-like environment, where searching must be able to determine approximate directions for units to move, given their respected vision radius. Testing is an important aspect to showcase this, which is what the article does with Theta*. It is tested on the most common formats for video game environments and it showcases promising results for Theta* [19]. These results are acquired by making minor adjustments to Theta* giving reason to apply such thinking to other search algorithms. The difficulties searching in an environment such as Battlecode could be better approached by using the modifications presented in the article.

5 Partially Observable Environments

The environment present in Battlecode is incredibly limited and it is considered partially observable as no single unit will have full vision of the map. As a result, viewing search problems with partially observable environments as belief spaces can prove to be rather useful. This allows the use of beliefs, which are a set of possibilities that the environment can exist in. Unfortunately, states like these can result in agents navigating to dead ends [5]. Similar dead-end states can occur from obstacles surrounding an agent, such as other agents or immovable blockades.

The paper approaches this environment via contingent planning. In such planning, it is accepted that there may be no guarantee of an agent reaching its destination, and every action is a precondition prior to movement. While planning like this can be helpful, situations like these might not be solvable. Consider a situation where the map is completely unknown and dead ends exist within the map. Such a state is not considered solvable, and this is very applicable to the environment supported in Battlecode. For the most part, the map is relatively unknown until units start exploring within the game. From there onward, information must be collected in a format that can be relayed to other units for proper coordination. But real-time information can not be communicated, so every agent must be able to determine their belief states based on what they can sense. Contingency planning can better help agents possibly escape dead end states that could be present in Battlecode.

6 Bytecode Analysis

The fundamental goal of this project is to present a concrete platform-independent way of comparing search algorithms. It follows then that an ironclad method of measurement is a requirement to proceed. Battlecode naturally suggests the usage of Java Bytecode as the measurement of choice. The number of instructions being run through the Java Virtual Machine (JVM) can be measured independently of the speed of the JVM itself - allowing the instruction execution count of the JVM to be a platform independent measurement device of algorithm efficiency in real world settings [10].

Obtaining this measurement of JVM instruction count is a non-trivial task. Walter Binder and Jarle Hulaas were among the researchers who first explored techniques to accomplish this. They found that a common technique, JVM profilers, could not achieve exact instruction count results, only approximations. To achieve exact bytecode instruction usage, instrumentation must be used [7]. Instrumentation is the technique of inserting new third party instructions into the bytecode to add side effects to code execution. One can measure the total instruction count usage of a method by inserting the incrementation of a global variable after every instruction and comparing prior and posterior values.

Various tools exist for achieving the code instrumentation required for this measurement. Binder and Hulaas went on to describe techniques for instrumentation in their 2007 paper titled "Advanced Bytecode Instrumentation." [8] Their tool never appears to have been released publicly. As such, it can not be applied to the present use case.

A popular framework among past researchers for bytecode instrumentation has been Soot. Soot is a powerful full featured framework for Java bytecode analysis. It supports a wide variety of operations such as call graph construction, bytecode instrumentation and transformation, and includes support for most Java features [21]. Soot can certainly accomplish the task requirements for measuring bytecode usage presented. However, given the wide range of features present in Soot, its usage is complex and is much more full-featured than required for this project.

Such discussion invites the obvious question of "how does Battlecode measure bytecode usage?". The Battlecode's 2023 code pack includes a custom built bytecode counter for the JVM [2]. It is built on top of Java's native instrumentation and reflection interface introduced with Java 1.5. The technique it uses is similar to the one previously described, where after every instruction, an incrementation of a global variable is added. However, instead of adding an incrementation, an addition is added at the end of every basic block equal to that block's number of instructions. A basic block is a deterministic chunk of code with no branching behavior or method calls.

7 Algorithm Introduction

An essential consideration when comparing algorithm performance is what algorithms should be compared and how they are implemented. This is where one of the major benefits of being inspired by a competitive coding competition such as Battlecode comes into play. Competitors are usually eager to brag and share their strategies, enabling us to view a treasure trove of successful implementations of various algorithms and determine which ones are worth exploring in a compute bound environment. No solution is guaranteed to be cost-optimal due to the environment being partially observable. This is because each unit has a different vision radius of the map. Referencing units in Battlecode 2023, Carrier type agents have a vision radius of 20 squared euclidean distances, while Headquarters have a vision radius of 34 squared euclidean distances. Neither of these visions are complete coverage of the map, so agents rely heavily on localized searches to get around and perform actions.

Analyzing the Github repositories of top competitors highlighted a few such pathfinding algorithms. Oxfrd, a top team of Battlecode 2022 used a two phase hybrid approach. Their strategy begins with a highly optimized bytecode hack for Bellman-Ford's algorithm, before switching to Bug as they near their computational limits [3]. Many other top teams, such as Producing Perfection, use a similar dynamic navigation strategy where they adapt their algorithm according to their currently available resources [1]. As the idea of dynamically switching between algorithms does not apply towards a comparison like that which this paper will lay forth, this behavior will not be replicated. This does still indicate that both Bellman-Ford's and Bug Navigation algorithms should be included in the comparison.

8 A-Star Search

No discussion of pathfinding algorithms is complete without mentioning A-Star. A-Star is a prolific algorithm introduced in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael [12]. It is unique in its ability to combine the best facets of a uniform cost search with that of greedy best-first search algorithms. Using a heuristic to complete an informed search while maintaining cost optimality under a well designed heuristic. This makes it well suited for a search that has a large branching factor by ignoring inefficient routes. Implementation of A-Star requires a priority queue or a similar data structure that introduces additional costs to maintain. The additional costs due to a data structure provide a motivating example of why algorithms should be measured in bytecode rather than nodes expanded. [12]

9 Bellman-Ford

The Bellman-Ford algorithm is very similar to Dijkstra's, and searches for the shortest path in a directed graph, where some of the edges may be negative [4]. Algorithms like Dijkstra's, struggle with negative edges because of their greedy nature. As a result, Bellman Ford either returns the shortest path, or states that there is no possible shortest path, since negative edges could occur. The algorithm begins by setting the root node to a distance of zero, and every other node to a distance of infinity. Then, it goes on to 'relax' each node, which means to see if there is a possible way to shorten the route to the node to which the edge points to. If it is possible, the route is replaced with the new one.

Bellman introduced this algorithm as a dynamic programming problem used to find the shortest travel time between two cities, given N cities connected by various edges [6].

The distances between nodes is not proportional with travel time, due to variances in road quality and amount of traffic. Ford on the other hand, introduced this algorithm in order to solve the problem of finding the maximal steady flow of goods from a warehouse or factory to consumers [17]. It is solved by iteratively labeling the nodes. Once the terminal node is labeled, the process ends.

10 Breadth-First Search

Breadth-First Search (BFS) was first introduced by Konrad Zuse and Michael Burke in 1945, and later reinvented by Edward F. Moore in 1959 [11]. BFS visits the nodes preorder. Therefore, after visiting a node, BFS visits all the nodes adjacent to it first before moving on to other nodes [20]. Each neighboring node visited will be added to a queue. All nodes at the current level will be visited before visiting any nodes at the next level. Advantages of this algorithm are that it will never get stuck in loops or infinite search spaces, it is able to find all solutions (if there is more than one) and the cost optimal solution will be found first.

11 Bug Navigation

Bug algorithms serve as a useful starting point for naive pathfinding approaches because of their relatively cheap search cost. This is due to them relying heavily on local searches to determine the direction of the agent's movement. Among these, Bug-0 is the most commonly used approach, which considers all adjacent tiles and identifies the one that brings the agent closest to its destination. If this tile is blocked by another agent or possibly a wall, Bug-0 will inform the agent to move alongside that blockage. In essence, the agent will follow the wall until it finds a tile that will take it closer to its destination. This is very naive, as the agent is only looking at adjacent positions to move on top of and not utilizing all information within its full vision [22].

There are also different variants of bug navigation such as Bug-1, Bug-2 and Dist-Bug. What will be explored and utilized heavily within the project is Bug-2 and Dist-Bug. Both of these variants keep track of a line from the start to the end location [22]. So if an obstacle is encountered and the agent is forced to traverse around it, once it reaches the tile on that line it can move off the obstacle. Dist-Bug improves this further by tracking a start position of when an obstacle was encountered and if a cycle is found, it attempts different directions. This enhanced Bug-based search provides a reliable and cost-effective solution for pathfinding, especially when compared to methods that track multiple positions via a queue like Breadth-First Search.

12 Dijkstra's

Djikstra's algorithm (also known as uniform cost search) is an intuitive algorithm for finding cost optimal paths between two points. As the name suggests, it was proposed by E. W. Djikstra in 1959 [9]. It works by repeatedly expanding the current branch of the path with the lowest cumulative total cost from a priority queue, making it ideal for exploring weighted graphs. In an unweighted graph where all the edges have equal weight, it functions identically to a BFS. For Battlecode's environment, the only way edges could be weighted is due to the presence of rubble, giving it limited application in the scope of this project.

13 Iterative Deepening Depth-First Search

Iterative Deepening Depth-First Search (IDDFS) is a variation of Depth-First Search (DFS) that provides better performance by avoiding unnecessary search and wasteful computations [15]. To begin, DFS is a search algorithm that finds the goal node by expanding the first child node beginning at the root, and the following nodes, until it reaches a node with no children. All expanded nodes are added to a queue to be explored later. When DFS is used on large or infinite graphs, the algorithm may never terminate and find the goal node. Therefore, IDDFS was developed to avoid the non-termination problem, by applying a depth-limited search multiple times, each time incrementing the search depth until the goal node is found. Each time the depth-limited search is redone to a new depth, the nodes enqueued in the previous search are discarded in order to properly start over. IDDFS is guaranteed to be cost optimal and find the shortest path to the goal node because it does not get lost in a potentially infinite search along one path.

14 Previous Pathfinding Comparisons - Base Environment

A-Star, BFS and Dijkstra's were compared to analyze their efficiency in a two-dimensional grid world [13]. In the experiment, 20% of the nodes in the two-dimensional grid were marked as impassable. The experiment was conducted with a variety of grid sizes, ranging from 64x64 to 1024x1024 nodes. To reduce random error, the experiments were repeated 100 times each. Furthermore, the agents could only move to directly adjacent nodes, meaning no diagonal movement. The action costs were all equal, with the cost between each node being 1. The algorithms were compared by analyzing their performance in terms of execution time (ms) and the number of traversed nodes.

In terms of execution time, BFS performed the worst, followed by Dijkstra's and lastly, A-Star. Since BFS is a brute force uninformed search, its execution time increased exponentially for each increase in grid size. On the other hand, A-Star and Dijkstra's execution time increased linearly with grid size, with A-Star only doing slightly better. Similarly, BFS traversed the most nodes out of the three algorithms, although Dijkstra's only fared slightly better. However, A-Star traversed significantly less nodes than both BFS and Dijkstra's, traversing about 10% of the nodes that the other two

algorithms traversed. Therefore, A-Star was the most optimal algorithm to use for grid based searches with impassable nodes because of its heuristic.

15 Previous Pathfinding Comparisons - Game-Like Environment

A-Star and Dijkstra's were compared in a game-like environment to determine which was the most efficient (able to find the shortest path to the goal node in the least amount of time) [16]. The experiments were composed of randomly generated sets of nodes in three different possible navigable environments. Each algorithm was tested in each environment 50 times. The grids were composed of passable and impassable nodes.

In the first environment, Dijkstra's examined six times as many nodes as A-Star on average. However, both algorithms performed very similarly in terms of the number of average nodes in a path, the average length of a path and the average path execution time. Similarly, Dijkstra's did worse than A-Star in the second and third environments, this time with Dijkstra's examining about 5 times as many nodes as A-Star on average, with the rest of the metrics being very similar.

Each algorithm was given an 'efficiency score' per environment, with the 'efficiency score', S, equal to k/(V*L), where k was a constant of 10,000 to increase understandability, V was the average number of visited nodes and L was the average path length. Therefore, the efficiency scores for A-Star were 3.037, 18.291 and 17.802 for the three environments, and Dijkstra's efficiency scores were 0.512, 3.881 and 3.83. Evidently, Dijkstra's was less efficient than A-Star in all three environments, mostly because of the number of nodes the algorithm visited.

16 Previous Pathfinding Comparisons - Robot Environment

Robot navigation, using a grid-based environment to represent the navigation space in the real world, is applicable to the grid-based environment in Battlecode. In robot navigation, robots must plan the shortest possible path to the goal destination, while avoiding obstacles that are present in the environment [11]. BFS, Dijkstra's and A-Star were compared in this environment, with the robot able to move in any direction. The algorithms were tested in three different grid maps with varying configurations. A-Star used the Manhattan distance from the goal node to the search node as a heuristic.

It was found that A-Star was twice as fast as BFS and Dijkstra's in the first two maps. BFS and Dijkstra's took almost the same amount of time to complete the search. In the third map, A-Star was about 10 times as fast as BFS and Dijkstra's, taking 0.6 seconds to run. BFS and Dijkstra's took about 5 seconds. Map three also proved to be more challenging for BFS and Dijkstra's to search over because it contained more obstacles than in the first two maps, and the obstacles were configured in a more complex way.

17 Approach

Rather than using the Battlecode environment, instead simplifications were made with a custom environment. Therefore classes were created to best model the software architecture of the Battlecode setting. This environment is publicly viewable at the cited GitHub repository [14].

17.1 `Coords` Class

This represents a 2D point on the grid. This class is vital to referencing locations within the simulation. Further it keeps track of the path cost used in algorithms such as Dijkstra's and A-Star. The functions attached to this class consist of custom hashing and equality checking. These two functions allow the `Coords` class to be stored in data structures such as hash tables and priority queues.

17.2 `GridLocation` Class

The grid represents 2D array of `GridLocation` objects. This class is important to maintaining state of a position on a grid. This allows the use of getters and setters in Java to alter positions and get positions. The class keeps track of a type for each position. Labeled as either an "AGENT", "OBSTACLE", "EMPTY", "RUBBLE", or "GOAL" along with a respective cost for that type.

17.3 `Grid` Class

The `Grid` class is where the generation of the grid and control of it exists at. The object keeps track of notable instance variables such as `world` which represents the 2D grid along with positions for where an agent and goal is placed. The `world` is generated based off a seed where specific attributes of the world can be specified through the constructor. Attributes such the dimensions of the grid and the rate of congestion can be customized in this constructor. In terms of world generation, it is done purely at random, where locations with rubble have a random score between 1 to 100 and locations with obstacles are marked as impassable.

17.4 `Agent` Class

The `Agent` class represents the agent within the world. It keeps track of it's current location and it used within the `Sim` class. Where it moves towards the goal utilizing the different algorithms.

17.5 `Sim` Class

The `Sim` class houses all the functions required for testing the different algorithms. Further it initializes the map, sets a starting and goal position for the agent. This class is explored in greater detail in the *Experiment Setup* section.

17.6 Algorithms

In order to best build all the algorithms, the strategy design pattern was utilized. It helped enforced function requirements in each algorithm and reduced the amount of code that needed to be written for testing. The algorithms used in this experiment were A*, Bug, BFS and Dijkstra's. DFS and IDDFS were also implemented; however, they were not used after it became evident that they find a new path every round, which leads to a lot of backtracking. From the tests run, DFS and IDDFS never made it to the goal, as they would keep moving back and forth between two positions. An attempt was made to implement Bellman-Ford; however, due to the complexity of the algorithm, it was not completed.

17.6.1 A Note on the Handling of Partially Observable Environments.

It can be noted that none of the algorithms properly handle partially observable environments. There is a important motivator behind this. Partially observable algorithms are less standardized and wide spread. Thus, to aid in the generalizability of the results, a generic solution that could be applied to any pathfinding algorithm was used. The algorithms were given a intermediary goal in every movement step. That intermediary goal was the closest position to the true goal within the agent's vision radius. Calculating this goal was done outside of the context of instrumentation to avoid the constant addition to all algorithms cost.

17.7 Instrumentation

Hooking into Battlecode's pre-existing framework for measuring bytecode was used for instrumentation. Battlecode measures how much bytecode a player's agent has used. This is for the purpose of halting a player's agent if it uses too many computational resources. It does this by setting up a custom class loader and loading the players code via Java reflection. Then, a monitor class with a static "bytecodesLeft" variable is loaded in addition to the player's agent. If this variable hits zero, the player's turn ends. Whenever any other class is loaded with the custom Battlecode class loader, subtractions are added at the end of every basic block from this static variable.

Battlecode's instrumentor was hijacked for measurement purposes by setting the bytecodesLeft variable to an arbitrarily large integer. Preventing the monitor class from ever forcing the halt of the testing system. The algorithm under test was then loaded in the custom Battlecode loader. After running the algorithm, the bytecodesLeft integer has the amount of bytecode used subtracted from it. Measuring the difference between the arbitrarily large integer and the remaining bytecodesLeft yields the amount of bytecode that was used to run the algorithm.

18 Experiment Setup

The different searches were ran in a new experiment environment to tackle the issues of observability. There were limitations modifying the Battlecode environment to create more generalizable results. Because of this, a new environment was created where the vision radius of agents can be modified to see how it changes the performance of the searches.

The performance of the optimized algorithms was evaluated by running controlled experiments in which an agent is required to move to a certain position on the board in a single-agent environment. The final results of each experiment were the mean number of Java bytecode instructions it took for an agent to reach the goal state from the initial state over numerous runs of the pathfinding algorithm.

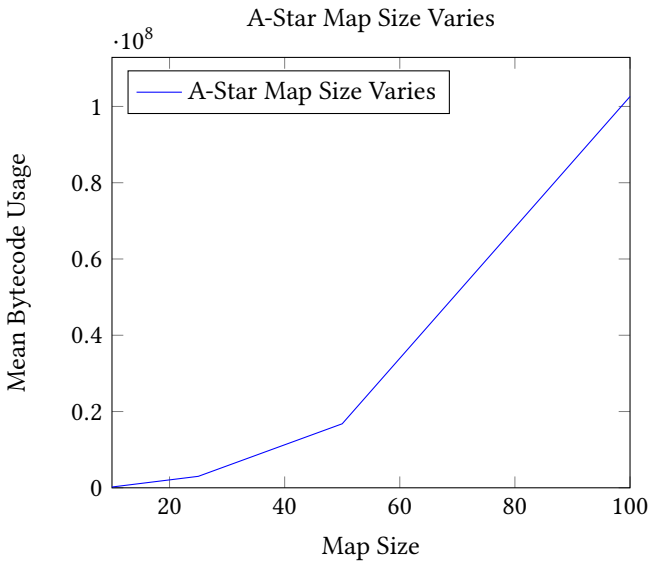
The statistically validity of these runs was ensured by running until certain criteria was met. To be specific, a 95% confidence interval was calculated using the running results of the experiment. Then, simulation was only halted after this confidence interval's half width was less than 10% of the samples mean.

While calculation of the agent's next move was done within the instrumented environment, other details of the simulator were excluded from the instrumentation. This includes the confidence interval calculation as well as the updating of the agent's position on the grid. Pseudo-code showing the simulation code can be seen following.

19 Findings

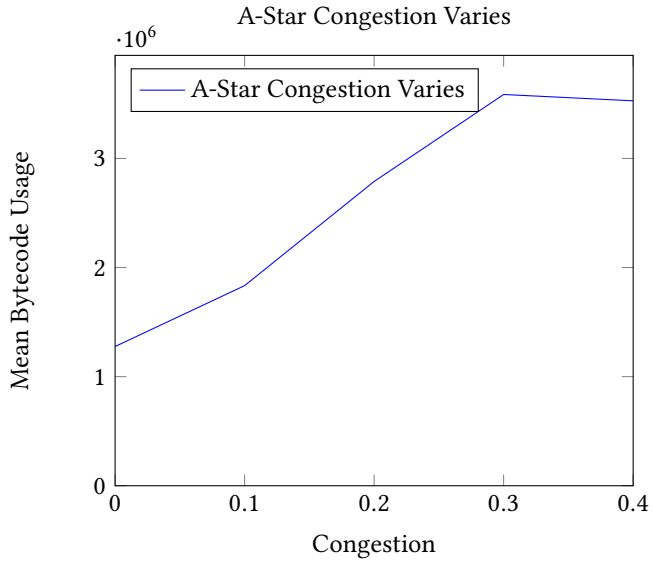
For the following graphs comparing Map Sizes, the congestion is set to 20%. For the Congestion graphs, the map size is 25x25. For the Vision Radius graphs, the map size is 25x25 and the congestion is 0%.

19.1 A-Star

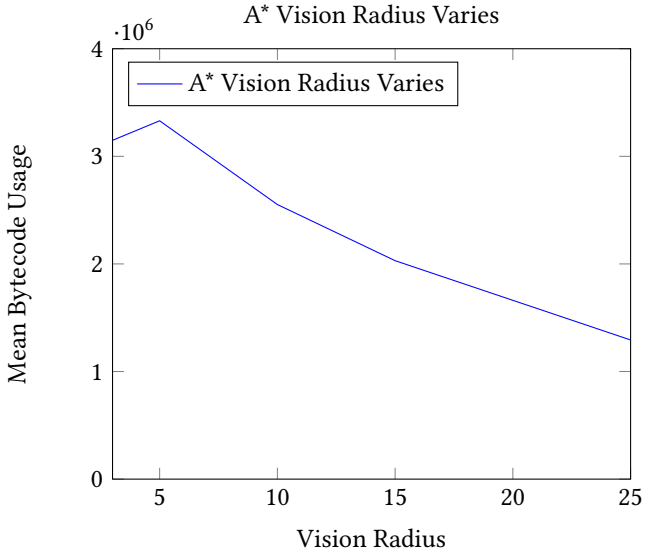


As map size increases, A-Star's mean bytecode usage increases almost linearly. The rate at which it increases is the highest for maps of size 50x50 and greater. Reaching a map

size of 100x100, A-Star averages a cost of 10^8.

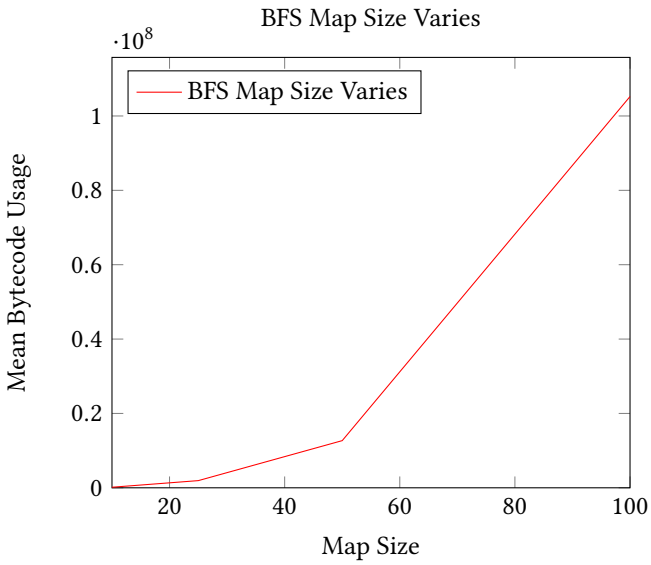


With congested maps, A-Star's mean bytecode usage increases linearly, but when the map is about 30% congested, the mean bytecode usage plateaus and even begins to decrease. With 40% of the map congested, A-Star averages a cost of $3 * 10^6$.

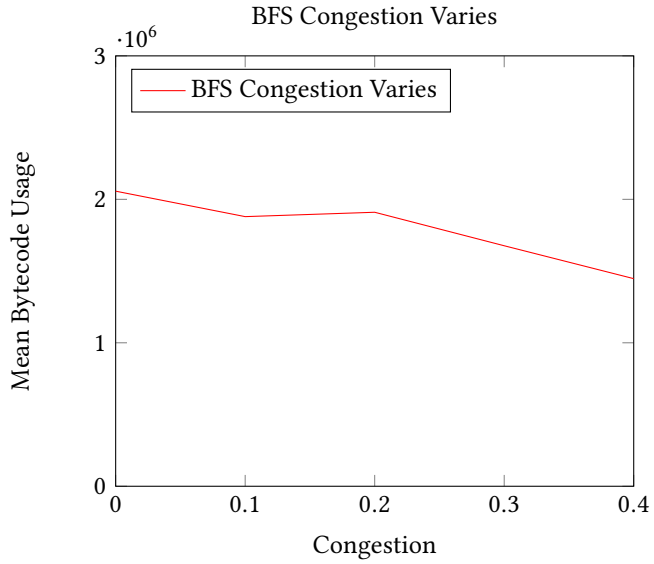


A-Star's mean bytecode usage surprisingly decreases at a pretty steady rate as vision radius increases. A-Star's bytecode usage at a vision radius of 5 averages $3 * 10^6$, and drops to about $1 * 10^6$ mean bytecode with a vision radius of 25. The decrease in bytecode usage with an increase in vision radius is due to A-Star's use of a heuristic. As vision radius increases, A-Star is able to utilize it's hueristic more effectively, and find better, cheaper (rubble-wise) paths to the goal. With a small vision radius, A-Star is not able to view most of the map, and may struggle to figure out which path is the best in the long-term. As a result, it may change paths over the course of the search.

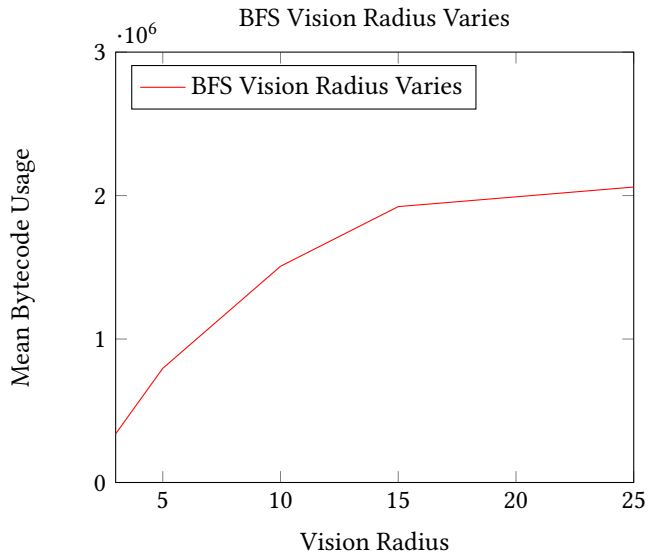
19.2 Breadth-First Search



Similar to A-Star, BFS's mean bytecode cost grows slowly until it hits a map size of 50x50. At that point, it grows at a much higher rate, reaching a cost of 10^8 for a 100x100 map.

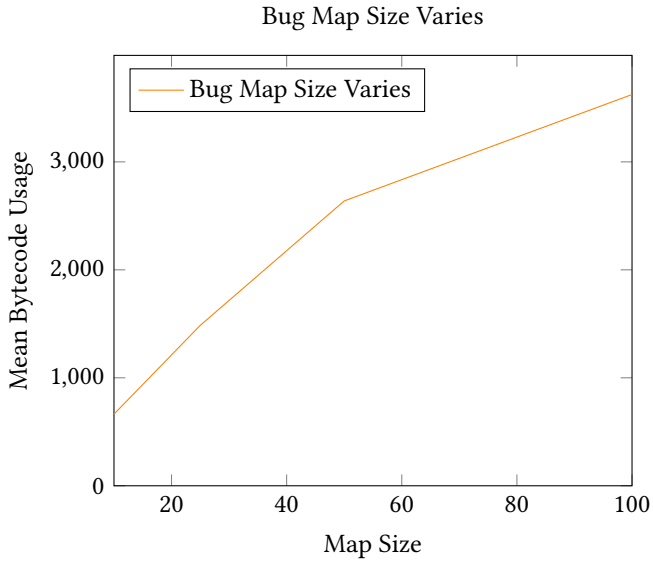


Contrastingly, BFS's mean bytecode cost decreases with congestion almost linearly. With no congestion, it averages $2 * 10^6$, but with 40% of the map congested, it averages a cost of $1.5 * 10^6$. This is due to there being less possible paths available to explore.

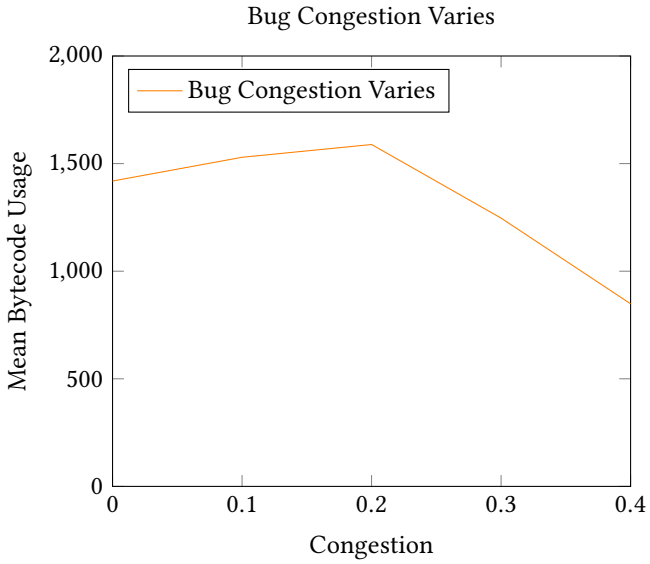


Understandably, BFS's average bytecode usage increases with vision radius, but enters almost a plateau at a vision radius of about 15. This makes sense, as BFS has to consider more paths and explore more map locations.

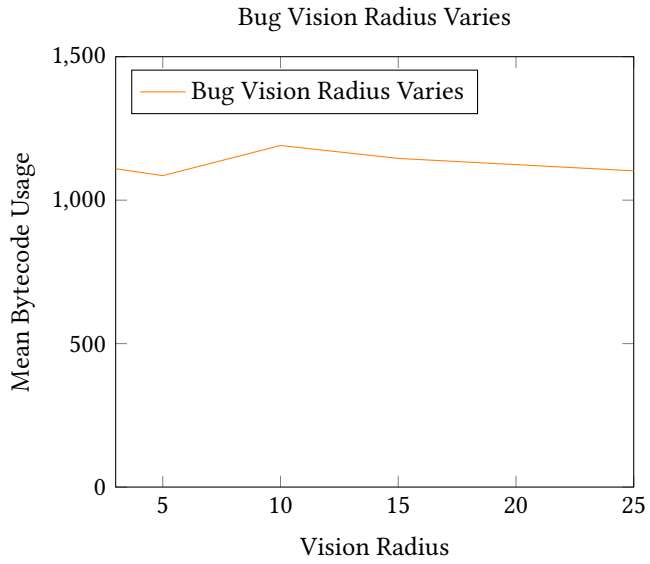
19.3 Bug Navigation



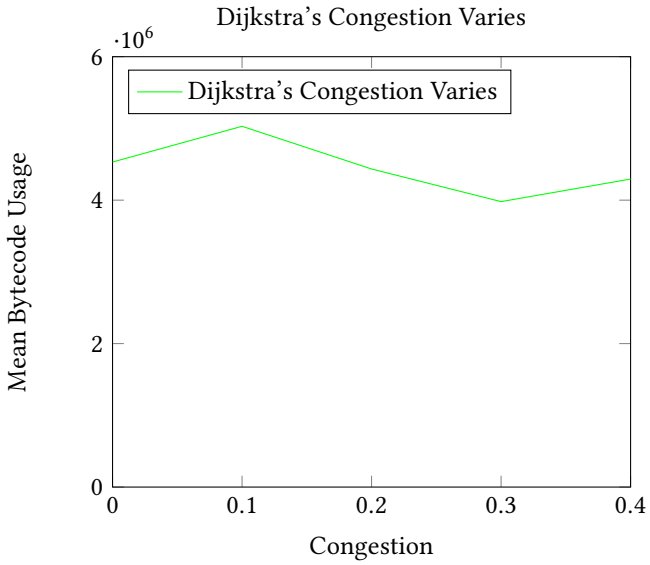
As map size increases past 50x50, Bug's rate of average bytecode usage actually decreases. With a map of size 100x100, Bug's mean bytecode usage is about 3,500.



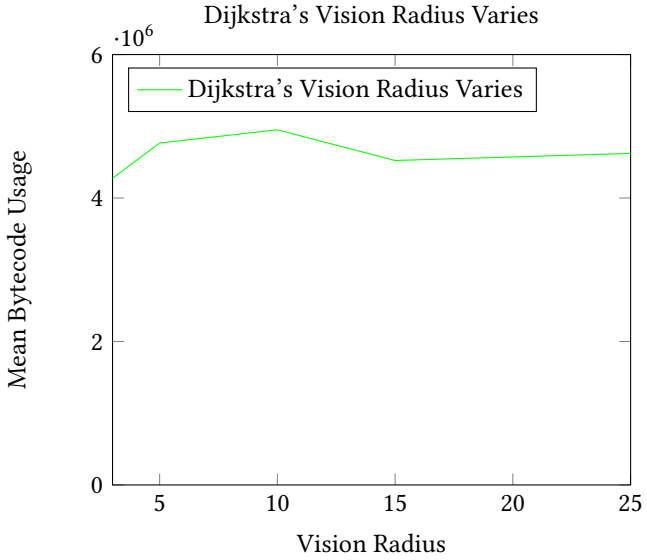
For maps with 20% or more congestion, Bug's mean bytecode usage rapidly decreases. With 0% of the map congested, Bug averages 1,500 bytecode usage; however, with 40% of the map congested, Bug averages less than 1,000. This drop in bytecode usage is likely due to a bias towards maps with goals closer to the agent. When choosing maps, any map that Bug did not finish searching after 1,000 iterations was discarded in case of infinite loops.



Just like Dijkstra's, Bug's average bytecode usage is very consistent in regards to changes in vision radius. The bytecode usage sits at around 1,100. This is due to Bug only caring about the direction to the goal, rather than the actual goal itself.

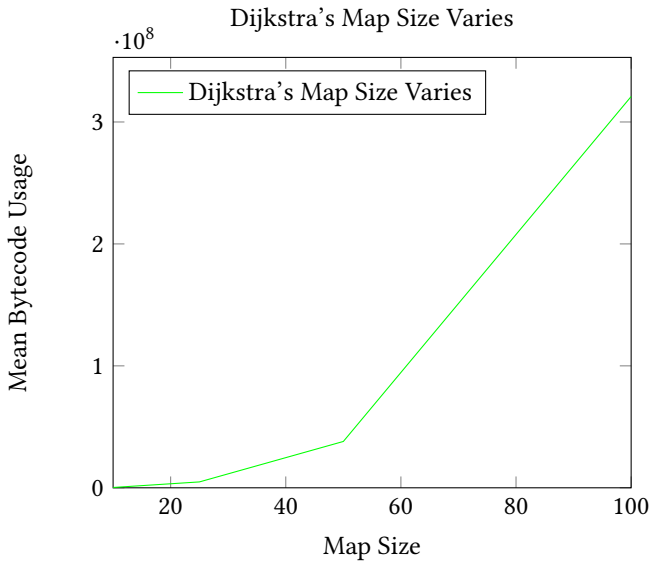


However, Dijkstra's mean bytecode usage is very non linear for variances in congestion. Ultimately, the bytecode usage did not change from 0% congestion to 40% congestion. For both congestion amounts, the average bytecode usage was about $4.5 * 10^6$.



For variances in vision radius, Dijkstra's seems to stay at a pretty steady bytecode usage of about $5 * 10^6$. This is likely due to the fact that Dijkstra's is a uniform-cost search, so the path costs do not have a significant change with regards to vision radius.

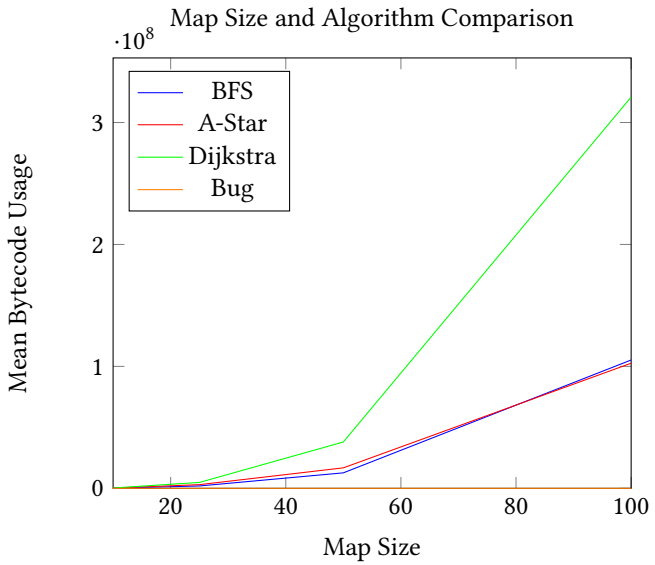
19.4 Dijkstra's



Dijkstra's average bytecode usage grows slowly before reaching a map size of 50x50. For a 100x100 map, Dijksta's used $3 * 10^8$ bytecode.

20 Analysis

20.1 Map Size

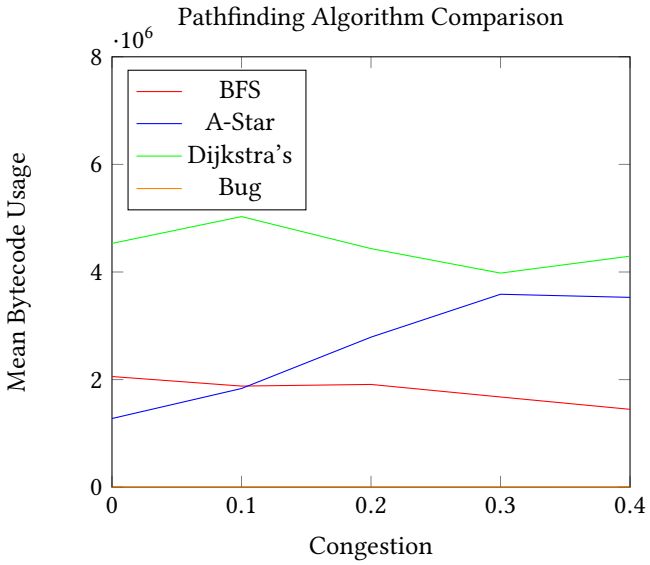


The graph above provides a comparison of the resulting algorithms bytecode performances for different map sizes, each being 20% congested. It is shown that the at small map sizes of 20x20, the algorithms perform neck to neck in terms of bytecode cost. This can be deduced by the fact the finding a path to the goal can computationally be done quickly. As the map size increases, the searches algorithms drastically change. At map size of 40x40, Dijkstra performances significantly worse due the cost of the priority queue in the builtin `java.util` library. While A-Star similarly uses a priority queue, the heuristic allows it to consider less positions compared to Dijkstra's. This trend continues as Dijkstra's performs worse on greater map sizes while Breadth-First Search and A-Star perform about equally.

The reason A-Star performs similarly to Breadth-First Search while considering less positions could be due to the priority queue. This is because A-Star needs to keep track of the next best positions to visit and this can only be done with the use of a priority queue. Another note to make for Breadth-First Search is that it does not consider the rubble on each tile compared to Dijkstra's and A-Star. This could be vital information where in an actual game, ignoring such terrain features could result in a quick loss. While these notes are most likely the reason for the performance of these two algorithms being so similar, it is not possible to confirm. One of most notable performances is that of the Bug search algorithm, it cost is extremely low on different datapoints on the graph. This is because Bug only considers adjacent locations, ignoring rubble and not considering a path compared to the other search algorithms.

Based off the graph one would be inclined to believe Bug is the best suited algorithm for traversing a grid cheaply in terms of bytecode. But this could be further from the truth, Bug is not a complete algorithm nor does it guarantee a efficient path from one position to another in the grid. Further there could be unintentional biases present within the experiment where bug is performing well on maps where the agent's position is relatively close to the goal position.

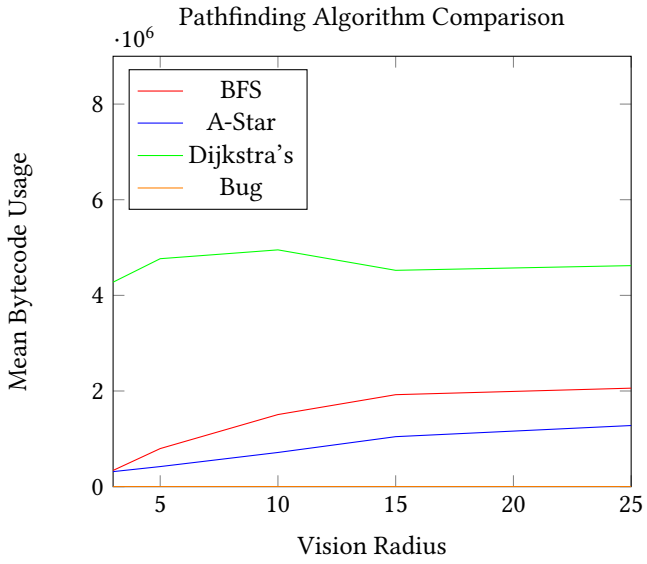
20.2 Congestion



This graph references different bytecode results from each algorithm based off varying levels of congestion on a 25x25 map. Out all the complete algorithms present here, A-Star performs the best at low congestion rates. This makes sense since its tries to find the most optimal path via euclidean based heuristic. Allowing it search quickly around obstacles once encountered. Dijkstra's suffers from having search a good amount of the map as it only considers the accumulating path cost of the rubble. Breadth-First Search expands outward in a flood-fill matter and can circumvent obstacles since it ignores the cost of traversing over rubble. Bug does not consider any sort of path, only a individual move so its cost should always remain low.

As the congestion on the map increases, so does the performances of A-Star and Breath-First Search. On a map with a 10% congestion rate, Breadth-First Search surpasses A-Star in terms of bytcode performance. This makes sense, while the heuristic guarantees a optimal path for A-Star, it could make it consider more positions if it follows paths that lead into dead ends. The path found by Breadth-First Search is not optimal but it does find a path at a lower bytecode cost than A-Star. As the congestion rate continues to grow, this trends becomes more noticeable for Breadth-First Search.

20.3 Vision Radius



At a congestion rate of 0% and a map size of 25x25, the effect that varying vision radius has on these algorithms is incredibly noticeable. At a small vision radius, the bytecode usage varies heavily. All the algorithms minus Bug, have a low bytecode cost due to the fact that they consider less locations within their vision. As the vision radius grows though, it is clear the more bytecode is being utilized except for Dijkstra's and Bug. As stated earlier, Bug considers only adjacent positions at its given position so its cost should always be low. As for Dijkstra's, the reason for the stagnating bytecode cost could be because it always examines the same amount of locations each move. At vision radius of 25, A-Star performs the most optimally out of all these algorithms due to the fact that it finds the best path at the lowest bytecode cost.

21 Future Work and Conclusion

Although Bug seems to be the most optimal search algorithm based on its low bytecode usage, it is not. Bug is incomplete, which does not make it very reliable. In a situation like Battlecode, reliable search algorithms are necessary. A-Star and BFS are the next best search algorithms in terms of bytecode usage. They performed very similarly in terms of mean bytecode usage on map size, congestion and vision radius. However, BFS had less bytecode usage as congesion increased, and A-Star had slightly less bytecode usage as vision radius increased. Out of the two of them, A-Star takes into account rubble, and BFS does not. This means that A-Star will try to avoid paths that require traveling through costly rubble, versus BFS will not take into account rubble cost. As a result, BFS is not guaranteed to find the most cost optimal solution. Even though A-Star is more costly in terms of bytecode than Bug and BFS (in congested environments), A-Star is still the best choice for a search algorithm in a grid based environments with an emphasis on bytecode usage. This aligns with what was discovered after examining previous research. All the sources that compared search algorithms found A-Star to be the best. Interestingly however, the sources also found that Dijkstra's did equal to or better than BFS. But, in these experiments, BFS always did significantly better than Dijkstra's.

Regarding future work, it would be worthwhile to compare Bellman Ford with the other algorithms. It would be especailly important to compare Bellman-Ford with A-Star, as A-Star was found to be the best algorithm. Furthermore, examining top Battlecode teams' repos, it was found that many teams dynamically switch between algorithms, specifically Bellman-Ford and Bug. Therefore, it would also be interesting to look into how dynamically switching between algorithms could improve bytecode usage and pathfinding. Lastly, the code could be made more extendable for enhanced reusability in other domains. As of now, the code is very specific to Battlecode; however, it definitely has potential to be generalized for other path planning research and applications, unrelated to Battlecode. With possible use cases existing outside AI all together. Testing on many embedded systems is extremely difficult. Creating a demand for a general purpose code performance measurement tool with a focus on transferability of results. Enabling performance testing outside of the target embedded environment.

22 Work Division

Andy wrote the Introduction, Problem, Congested Video Game Environments, Pathing in Game Development, Partially Observably Environments, Bug, Approach, Analysis and Challenges sections. In terms of code, he wrote the Grid and GridLocation classes to set up the environment. He also wrote the A-Star, Dijkstra's, BFS and Bellman-Ford algorithms.

Jack wrote the Abstract, Bytecode Analysis, Algorithm Introduction, Dijkstra's, A-Star, A Note on Partial Observability and Experiment Setup sections. He also included algorithms 1 and 2. Code-wise, he wrote the instrumentation code in the Instrumentor class. He also completed the Sim class and implemented the Bug algorithm. He also did the testing and experiments, as well as creating the graphs in the Findings and Analysis sections.

Katya wrote the BFS, IDDFS, Bellman-Ford, Previous Pathfinding Comparisons, Approach, Findings, Conclusion/Future Work and Work Division sections. For the coding aspect, she implemented the Agent and Algo classes, as well as the DFS and IDDFS algorithms.

References

[1] [n. d.]. awesomelemonade/Battlecode2022. https://github.com/awesomelemonade/Battlecode2022.

[2] [n. d.]. battlecode/battlecode23. https://github.com/battlecode/battlecode23/tree/master/engine/src/main/battlecode/instrumenter/bytecode.

[3] [n. d.]. MarkHaoxiang/CU-Battlecode-2022. https://github.com/MarkHaoxiang/CU-Battlecode-2022/tree/master/src.
[4] Samah W. G. AbuSalim. 2020. Comparative Analysis between Dijkstra and Bellman-Ford Algorithms in Shortest Path Optimization.
[5] Alexandre Albore. 2014. Acting in partially observable environments when achievement of the goal cannot be guaranteed. https://www.academia.edu/1403878/Acting_in_Partially_Observable_Environments_When_Achievement_of_the_Goal_Cannot_be_Guaranteed
[6] Richard Bellman. [n. d.]. ON A ROUTING PROBLEM. ([n. d.]).
[7] Walter Binder and Jarle Hulaas. 2006. Exact and Portable Profiling for the JVM Using Bytecode Instruction Counting. *Electronic Notes in Theoretical Computer Science* 164, 3 (2006), 45–64. https://doi.org/10.1016/j.entcs.2006.07.011 Proceedings of the 4th International Workshop on Quantitative Aspects of Programming Languages (QAPL 2006).
[8] Walter Binder, Jarle Hulaas, and Philippe Moret. [n. d.]. Advanced Java Bytecode Instrumentation. ([n. d.]).
[9] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (dec 1959), 269–271. https://doi.org/10.1007/BF01386390
[10] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. 2003. Dynamic Metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Anaheim, California, USA) *(OOPSLA '03)*. Association for Computing Machinery, New York, NY, USA, 149–168. https://doi.org/10.1145/949305.949320
[11] Faza Fahleraz. [n. d.]. A comparison of BFS, Dijkstra's and A* Algorithm for Grid-Based Path-Finding in Mobile Robots. ([n. d.]).
[12] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136
[13] Sergejs Kodors Imants Zarembo. 2013. Pathfinding Algorithm Efficiency Analysis in 2D Grid. In *Proceedings of the 9th International Scientific and Practical Conference. Volume II.*
[14] Katya Borisova Jack Lee and Andrew Hoyle. [n. d.]. JackLee9355/MEAT. https://github.com/JackLee9355/MEAT.
[15] Navneet kaur and Deepak Garg. [n. d.]. Analysis Of The Depth First Search Algorithms. ([n. d.]).
[16] Nikhil Krishnaswamy. 2009. A Comparison of Efficiency in Pathfinding Algorithms in Game Development. Bachelor's Thesis.
[17] Jr. L. R. Ford. [n. d.]. NETWORK FLOW THEORY. ([n. d.]).
[18] Hang Ma, Jingxing Yang, Liron Cohen, T. K. Satish Kumar, and Sven Koenig. 2017. Feasibility Study: Moving Non-Homogeneous Teams in Congested Video Game Environments. arXiv:1710.01447 [cs.AI]
[19] Sara Pardede, Fadel Athallah, Yahya Huda, and Fikri Zain. 2022. A Review of Pathfinding in Game Development. *[CEPAT] Journal of Computer Engineering: Progress, Application and Technology* 1, 01 (2022), 47–56. https://doi.org/10.25124/cepat.v1i01.4863
[20] Robbi Rahim. [n. d.]. Breadth First Search Approach for Shortest Path Solution in Cartesian Area.
[21] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.
[22] Muhammad Zohaib, Syed Mustafa Pasha, Nadeem Javaid, and Jamshed Iqbal. 2013. Intelligent Bug Algorithm (IBA): A Novel Strategy to Navigate Mobile Robots Autonomously. *CoRR* abs/1312.4552 (2013). arXiv:1312.4552 http://arxiv.org/abs/1312.4552

A Experimental Algorithms

A.1 SimulateUntilConfident

```
Function simulateUntilConfident
    instrumentor ← new Instrumentor();
    results ← new List();
    while confidenceIntervalPointEstimate(results)
     > 0.1 OR results.size() < 50 do
        grid ← instrumentor.newInstrumentedGrid(mt.nextLong(),
         congestion, rows, columns);
        agentPos ← instrumentor.instrumentedCoordsToCoords(instrumentor.
         getInstrumentedAgentPos(grid));
        goalPos ← instrumentor.instrumentedCoordsToCoords(instrumentor.
         getInstrumentedGoalPos(grid));
        agent ← new Agent(10000, agentPos,
         goalPos); i ← 0;
        totalWork ← 0;
        broken ← false;
        while agentPos ≠ goalPos do
            work ← instrumentor.moveAgent(agent,
             grid, algo);
            if i > 10000 then
                // Infinite loop detected
                broken ← true;
                break;
            end
            totalWork ← totalWork + work;
            i ← i + 1;
        end
        if broken = false then
            results.add(totalWork);
        end
    end
    return confidenceInterval(results);
end
```

Algorithm 1: Simulate Until Confident

A.2 UpdateIntermediateGoalPosition

```
Function UpdateIntermediateGoalPos int agentX,
 int agentY, int visRadius
    radius ← visRadius; bestPosition ← null;
     closestDistance ← Double.MAX_VALUE;
    if visRadius = −1 then
        intermediateGoalPos ← goalPos; return;
    end
    agentPos ← new Coords(agentX, agentY);
    for i ← agentX − visRadius to
     i ≤ agentX + visRadius do
        for j ← agentY − visRadius to
         j ≤ agentY + visRadius do
            if i < 0 or i ≥ rows or j < 0 or
             j ≥ columns then
                Continue;
            end
            position ← new Coords(i, j);
            if getLocation(i, j).getType() =
             LocationType.OBSTACLE then
                Continue;
            end
            distanceToAgent ← euclidean(agentPos,
             position);
            distanceToGoal ← euclidean(goalPos,
             position);
            if distanceToAgent ≤ radius and
             distanceToGoal < closestDistance then
                closestDistance ← distanceToGoal;
                bestPosition ← position;
            end
        end
    end
    intermediateGoalPos ← bestPosition;
end
```

Algorithm 2: UpdateIntermediateGoalPos

B Pathfinding Algorithms

B.1 A-Star

```
Function AStarMove Grid grid, Coords initial,
 Coords goal, int visRadius
    costSoFar ← new dou-
     ble[grid.world.length][grid.world[0].length]; for
     double[] row : costSoFar do
        Arrays.fill(row, -1.0);
    end
    frontier ← new PriorityQueue<Coords>();
    parents ← new HashMap<Coords, Coords>();
    frontier.add(initial);
    parents.put(initial, null);
    costSoFar[initial.x][initial.y] ← 0;
    while frontier is not empty do
        curr ← frontier.poll();
        // Goal was found if curr.equals(goal) then
            path ← new ArrayList<Coords>();
            while curr is not null do
                path.add(curr);
                curr ← parents.get(curr);
            end
            return path.get(path.size() - 2);
        end
        for dx = -1; dx <= 1; dx++ do
            for dy = -1; dy <= 1; dy++ do
                if dx == 0 and dy == 0 then
                    continue;
                end
                xf ← curr.x + dx, yf ← curr.y + dy;
                if inBounds(grid, initial, visRadius, xf,
                 yf) then
                    gScore ←
                     costSoFar[curr.x][curr.y] +
                     grid.getLocation(xf, yf).getCost();
                    if costSoFar[xf][yf] == -1.0 or
                     gScore < costSoFar[xf][yf] then
                        costSoFar[xf][yf] ← gScore;
                        next ← new Coords(xf, yf);
                        next.pathCost ← gScore +
                         heuristic(goal, next);
                        frontier.add(next);
                        parents.put(next, curr);
                    end
                end
            end
        end
    end
    return new Coords(initial.x, initial.y);
end
```

Algorithm 3: A-Star Search

B.2 BFS

```
Function BFSMove Grid grid, Coords initial, Coords goal, int visRadius
    visited ← false; frontier ← new LinkedList<Coords>; parents ← new HashMap<Coords, Coords>;
    visited[initial.x][initial.y] ← true;
    frontier.add(initial);
    parents.put(initial, null);
    while frontier is not empty do
        curr ← frontier.remove();
        if curr equals goal then
            path ← new ArrayList<Coords>;
            temp ← curr;
            while temp is not null do
                path.add(temp);
                temp ← parents.get(temp);
            end
            return path.get(path.size() - 2);
        end
        for dx from -1 to 1 do
            for dy from -1 to 1 do
                if dx = 0 and dy = 0 then
                    continue;
                end
                xf ← curr.x + dx;
                yf ← curr.y + dy;
                if inBounds(grid, visited, initial, visRadius, xf, yf) then
                    Coords next ← new Coords(xf, yf);
                    frontier.add(next);
                    parents.put(next, curr);
                    visited[next.x][next.y] ← true;
                end
            end
        end
    end
    return new Coords(initial.x, initial.y);
end
```

Algorithm 4: Breadth-First Search

B.3 Bug

```
Function BugMove Grid grid, Coords initial, Coords goal, int visRadius
    newX ← x + dx;
    newY ← y + dy;
    if not inBounds(grid, newX, newY) then
        return new Coords(x, y);
    end
    if grid.getLocation(newX, newY).getType() ≠ LocationType.OBSTACLE then
        return new Coords(newX, newY);
    end
    newDX ← −dy;
    newDY ← dx;
    return if depth < 16 then
        bugMove(grid, x, y, newDX, newDY, depth+1)
    end
    return new Coords(x, y);
end
```

Algorithm 5: Bug

B.4 Dijkstra's

```
Function DijkstrasMove Grid grid, Coords initial,
  Coords goal, int visRadius
    costSoFar ← new dou-
      ble[grid.world.length][grid.world[0].length];
    for row : costSoFar do
        Arrays.fill(row, -1.0);
    end
    frontier ← new PriorityQueue<Coords>();
    parents ← new HashMap<Coords, Coords>();
    frontier.add(initial);
    parents.put(initial, null);
    costSoFar[initial.x][initial.y] ← 0;
    while frontier is not empty do
        Coords curr ← frontier.poll();
        if curr equals goal then
            path ← new ArrayList<Coords>;
            temp ← curr;
            while temp is not null do
                path.add(temp);
                temp ← parents.get(temp);
            end
            return path.get(path.size() - 2);
        end
        for dx from -1 to 1 do
            for dy from -1 to 1 do
                if dx = 0 and dy = 0 then
                    continue;
                end
                xf ← curr.x + dx;
                yf ← curr.y + dy;
                if inBounds(grid, initial, visRadius, xf,
                  yf) then
                    gScore ←
                      costSoFar[curr.x][curr.y] +
                      grid.getLocation(xf, yf).getCost();
                    if costSoFar[xf][yf] = -1.0 or gScore
                      < costSoFar[xf][yf] then
                        costSoFar[xf][yf] ← gScore;
                        next ← new Coords(xf, yf);
                        next.pathCost ← gScore +
                          heuristic(goal, next);
                        frontier.add(next);
                        parents.put(next, curr);
                    end
                end
            end
        end
    end
    return new Coords(initial.x, initial.y);
end
```

Algorithm 6: Dijkstra's