Automatic Algorithm Selection in Videogame Pathfinding

by

Devon Sigurdson

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Devon Sigurdson, 2018

Abstract

Videogames often use artificial intelligence to control characters in the game world. In doing so, videogames require one or more agents to navigate from their current location to some desired goal location without collisions. We explore improving algorithm performance in both single and multi-agent pathfinding (MAPF) through the use of algorithm portfolios and machine learning algorithm selection. In the single agent setting we explore selecting from different versions of a parametrized real-time heuristic search algorithm focused on minimizing the distance the agent travels. In the MAPF setting we select from three contemporary MAPF algorithms, Windowed Hierarchical Co-operative A*, Flow Annotated Replanning, and Bounded Multi-agent A* focused on maximizing the number of agents which reach their goal. We use the travel distance and completion time as secondary objectives. Algorithms will often have specific problem instance where their performance either excels or deteriorates, enabling another algorithm to out perform it even if this other algorithm is worse on average. Therefore, in this thesis, we investigate the use of deep learning to automatically select the best algorithm from a portfolio of algorithms for given problem instances. Empirical results show that an off-the-shelf convolutional neural network is able to improve performance over any single algorithm from our portfolio.

Acknowledgements

This research was a collaborative process which included several researchers providing advice and guidance. As such, I use the term we throughout as the work was not done isolation.

The beginning part of this research was developed collaboratively by myself and Professor Vadim Bulitko. In particular, the research for single agent automatic algorithm selection has been previously published (Sigurdson and Bulitko, 2017).

The rest of this thesis was conducted in a collaborative manner with a team of international researchers. I led the development of this research with significant guidance and contributions from Professor Vadim Bulitko at the University of Alberta, Professor William Yeoh at the University of Washington in St. Louis, Professor Carlos Hernandez at Universidad Andres Bello, and Professor Sven Koenig at the University of Southern California. Together we developed Bounded Multi-agent A*. The work has be published as well (Sigurdson, Bulitko, Yeoh, Hernandez, and Koenig, 2018).

Much of this thesis has been adapted directly from these publications. All the development and data analytics was performed by me personally with direction form my co-authors. The actual writing was a collaborative process between my co-authors and I.

I appreciate support from British Petroleum, NSERC and Nvidia.

Table of Contents

1 Introduction **1**
1.1 Motivation and Applications . . . 1
1.2 Contributions . . . 3

2 Problem Formulation **4**
2.1 Pathfinding . . . 4
2.1.1 Multi-agent Pathfinding . . . 4
2.1.2 Single-agent Pathfinding . . . 6
2.2 Automatic Algorithm Selection . . . 6

3 Related Work **8**
3.1 Algorithm Selection Portfolio Formation . . . 8
3.2 Algorithm Selection Granularity . . . 9
3.3 Algorithm Selection Process . . . 10

4 Single-Agent Pathfinding Algorithm Selection **12**
4.1 Introduction . . . 12
4.2 Search Framework . . . 13
4.3 Space of Algorithms . . . 14
4.4 Our Approach . . . 15
4.4.1 Algorithm Selection Granularity . . . 15
4.4.2 Portfolio Formation . . . 15
4.4.3 Automatic Algorithm Selection . . . 17
4.5 Empirical Evaluation . . . 20
4.5.1 Per-genre Algorithm Selection . . . 21
4.5.2 Per-game Algorithm Selection . . . 22
4.5.3 Per-map Algorithm Selection . . . 22
4.5.4 Per-problem Algorithm Section . . . 23
4.6 Discussion . . . 24

5 Automatic Algorithm Selection for Multi-agent Pathfinding **25**
5.1 Introduction . . . 25
5.2 MAPF Algorithms . . . 26
5.2.1 Windowed Hierarchical Cooperative A* . . . 26
5.2.2 Flow Annotation Replanning . . . 27
5.2.3 Bounded Multi-Agent A* . . . 28
5.3 Our Approach . . . 29
5.3.1 Algorithm Selection Granularity . . . 29
5.3.2 Portfolio Formation . . . 29
5.3.3 Automatic Algorithm Selection . . . 30
5.4 Empirical Evaluation . . . 31
5.4.1 MAPF Problem Generation . . . 31
5.4.2 Per-problem Algorithm Selection . . . 32

5.5 Discussion 35

6 Discussion and Future work **36**

7 Conclusion **37**

References **38**

Appendix A Bounded Multi-agent A* **42**
A.1 Preface 42
A.2 Introduction 42
A.3 Problem Formulation 43
A.4 Related Work 45
A.4.1 A* 45
A.4.2 Online MAPF Algorithms 46
A.4.3 Real-time Heuristic Search 48
A.5 Our Approach: BMAA* 49
A.5.1 Procedure NPC-Controller 50
A.5.2 Procedure Search-Phase 51
A.5.3 Procedure Search 52
A.6 Experimental Evaluation 53
A.6.1 Aggregate Completion Rate Results 54
A.6.2 Per-Map Results 56
A.7 Conclusions 59

List of Tables

4.1 Summary of each granularities algorithm selection mapping . . . 17
4.2 Results of per-genre selection 22
4.3 Results of per-game selection 23
4.4 Results of per-map selection 23
4.5 Results of per-problem selection 23

5.1 Algorithm performance on all problems. 33
5.2 Completion rate(%) by problem type. 35

A.1 Completion rates averaged over all MAPF instances for each map. 57
A.2 Completion times (in seconds) averaged over all MAPF instances for each map. 58
A.3 Travel distances averaged over all MAPF instances for each map. 58

List of Figures

1.1 Example of MAPF problem instances. Agents move from their starting location (green) to their corresponding goal(red). . . . 2

4.1 Example for algorithm selection based off of which genre the game belongs to. 18
4.2 Example for algorithm selection based off of which game the map is from. 18
4.3 Example for algorithm selection on a per-map basis. 19
4.4 Example for algorithm selection on a per-problem bases. . . . 20

5.1 Example of Flow Annotations. 28
5.2 Multi-agent algorithm selection example. 30
5.3 Three resized sample MAPF problem types with start (green pixels) and goals (red pixels): outside in (left), cross sides (centre), and switch sides (right). 32
5.4 Problem type definitions and examples. 33
5.5 Completion Rate % for the test problems averaged over 10 splits of the data. 34

A.1 NPCs on a *Dragon Age: Origins* map. 44
A.2 Completion rates averaged over all MAPF instances. 54
A.3 Issue for FAR: One-cell-wide corridors. 55
A.4 Issue for BMAA*: Dead ends. 56
A.5 Unsolvable MAPF instance for the BMAA* versions, where the triangular agent has to move to its red goal location while the dark green square agents are already at their own goal locations in a one-cell-wide corridor. 57

Chapter 1

Introduction

We begin by first providing an overview of motivations for why we believe our research is relevant outside the academic community and how its applications can be used in Section 1.1. We then highlight the contributions that this thesis provides in Section 1.2.

1.1 Motivation and Applications

Videogames are often populated with agents who move around and interact with the environment. To do this, these agents are tasked with finding a path from their current location to a goal located elsewhere. In general, the task of finding a path from one point to another is known as search or pathfinding. Pathfinding agents are typically guided by a heuristic estimate of each state's distance to the goal. The classical A* (Hart, Nilsson, and Raphael, 1968) guarantees the shortest path to the goal but is constrained to solving the entire problem before the agent's first steps can be taken. Learning Real-time A* (LRTA*) (Korf, 1990) pioneered real-time heuristic search by enabling the agent to move before it knows a complete path to the goal. It does so by updating the agent's heuristic beliefs as it searches towards the goal, and limits planning to its neighbours. The limited information centres the search around the agent (Koenig, 2001). This also allows for operating under a real-time constraint as the agent begins navigating the environment independently from the distance to the goal. The limited information and moving prior to knowing how to reach the goal causes the agent to make suboptimal

decisions and frequently revisits states. Several techniques have been developed (Vadim Bulitko and Sampley, 2016; Hernandez and Baier, 2012; Rivera, Baier, and Hernández, 2013) to address this revisitation problem. Recent work showed that no single combination of techniques was always the best choice for all problems and that by selecting among the available techniques leads to improved results over using a single algorithm (Vadim Bulitko, 2016b). This concept is known as the algorithm selection problem (Rice, 1976), where there are several possible algorithms to use to solve a problem and the best choice may vary based on the problem.

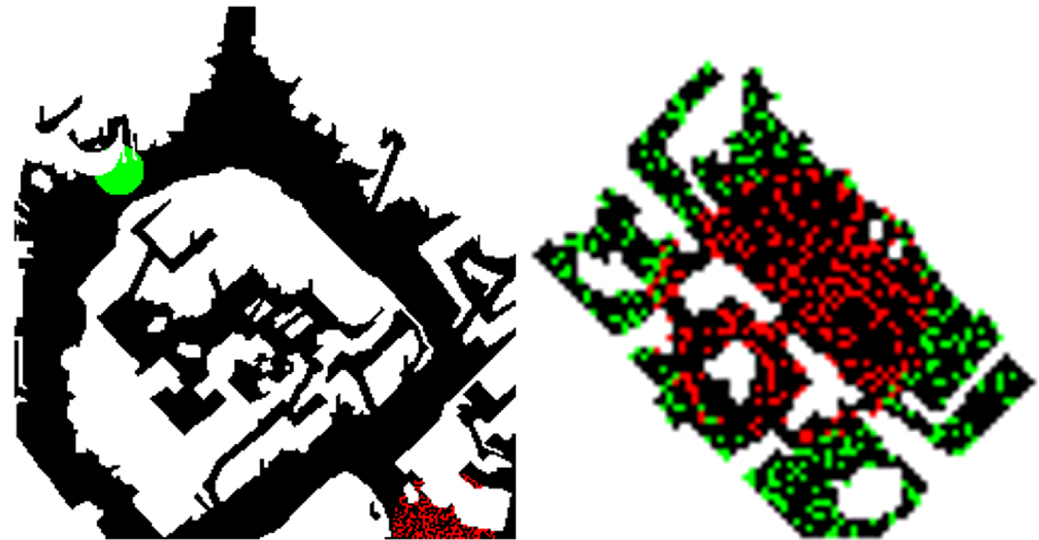

Figure 1.1: Example of MAPF problem instances. Agents move from their starting location (green) to their corresponding goal(red).

Videogames frequently require multiple simultaneous agents navigating a map at the same time, a hypothetical example can be seen in Figure 1.1. Both A* and LRTA* are single agent pathfinding algorithms, but algorithms based on them have been developed for multi-agent pathfinding (MAPF). Given the difficulty of the MAPF problem these algorithms often make trade-offs that sacrifice guaranteeing all agents reach their goal for reasonable operating time. The different sacrifices each method makes should allow per-problem algorithm selection to improve the number of agents that reach their goal.

The algorithm selection problem for pathfinding is likely to be very ap-

plicable to videogames where the maps the agents navigate in can be very diverse. Unreal Engine 4 is a popular videogame engine that has been used to develop over 200 games (Epic Games, 2018), which range from fighting games (e.g. TEKKEN 7 (Bandi Namco, 2017)), shooters (e.g. Gears of War 4 (The Coalition, 2016), Fortnite (Epic Games, 2017)), to role-playing games (e.g. Final Fantasy 7 Remake (Square Enix, 2019a), Kingdom Hearts 3 (Square Enix, 2019b)). It is a common trend seen among many engines, including Unity and Frostbite, to be used for a wide variety of games. This diversity should allow for selecting between multiple pathfinding algorithms to improve results over a applying a single method to all algorithms. Moreover, these game engines are often used by small teams that may not have expertise in pathfinding algorithms and as such could benefit from an automated approach that autonomously selects the best algorithm for their game based on their games characteristics. This motivates our development of a practical approach to automatic algorithm selection for videogame pathfinding.

1.2 Contributions

The thesis makes the following contributions. I demonstrate the ability for automatic algorithm selection to select variants of parametrized Learning Real-time Search in a single agent setting. I then extend our approach to the multi-agent setting which can outperform the best single algorithm in their portfolio. Additionally, I present a new high performance real-time multi-agent algorithm.

Chapter 2

Problem Formulation

Given the nature of algorithm selection there are two problems that need to be defined. The first problem is what the selected algorithms solve, in our case single-agent real-time heuristic search (defined in Section 2.1.2) and the multi-agent pathfinding problem (Section 2.1.1). The second problem is how to actually make these selections (Section 2.2).

2.1 Pathfinding

2.1.1 Multi-agent Pathfinding

A search problem is defined by the pair (G, A). $G = (N, E, c)$ is an undirected weighted graph of nodes (N) connected to each other by edges $E \subset N \times N$. $A = \{a_1, \ldots, a_j\}$ is a set of NPC agents, where each agent $a_i \in A$ is specified by a pair $(n^i_{\text{start}}, n^i_{\text{goal}})$ that indicates its start node n^i_{start} and its goal node n^i_{goal}. The set of edges includes self-loops: $\forall n \in N\, [(n, n) \in E]$ which allows all agents to remain at their current node (i.e., wait). We assume that all edge weights are positive real numbers except self-loop edges which have a weight of 0, as the agent does not travel any distance. Each edge $(n_a, n_b) \in E$ is weighted by the distance or travel cost $c(n_a, n_b) = c(n_b, n_a) >= 0$. Each agent begins in their start state n^i_{start} and changes its current state by traversing edges (i.e., taking actions). The cumulative cost of all edges it traverses prior to reaching their goal state n^i_{goal} is the solution cost or distance travelled. Each agent has access to its own heuristic h where $h(n)$ represents their estimate of the remaining cost to travel from n to n^i_{goal}. We do not assume the heuristic

to be admissible or consistent. The agent is free to update it in any way as long as $h(n^i_{\text{goal}}) = 0$. The initial heuristic $h_0 = h$ is included in the problem description.

In our model, time advances in discrete steps. At time step t, every agent a_i occupies a node $n^i \in N$, also referred to as n^i_{current} when talking about a specific agent's location. When pathfinding, each agent generates a set of moves it plans to execute from its current state. Agents provide these moves to a controller which attempts to have the agent execute its plans one step at a time. Plans are represented as a set of node pairs $P = \{(n, n')\}$. The pairs represent agent's planned actions (i.e., edge traversals) meaning that when the agent is on node n it intends to go to a neighbouring node n' by traversing the edge $(n, n') \in E$. These traversals are instantaneous, meaning that a diagonal move and a cardinal move both take a single time step. This simplification is done to avoid problems that arise from agents being on the edge between nodes on a diagonal traversals. A valid plan must contain a pair with n^i_{current} as the first element. Also, the agent is not allowed to have two different pairs with the same first element (i.e., no ambiguous actions in a single node). Consequently, the set P defines a partial mapping $P : N \to N$.

In the event that the agent's plan is not executable, either because another agent is occupying the node where it wishes to move or because the plan does not have an action planned for the agent's current node, the agent waits in its current node (i.e., traverses the self-loop). Agents can traverse edges with the following restrictions: (i) two agents cannot swap locations in a single time step and (ii) each node can be occupied by at most one agent at any time.

We use the following performance measures for MAPF problems: The *completion rate* is the percentage of agents that are in their goal locations when the runtime limit has been reached (Silver, 2005; Wang and Botea, 2008). The *completion time* for an agent is the amount of time (e.g. wall clock) elapsed when that agent last reached its goal location. Completion time measures the entire application (i.e planning, waiting, executing moves, etc.) but is mainly comprised of the planning time. If an agent leaves its goal and does not return the completion time is undefined. Finally, the *travel*

distance of an agent is the sum of the costs of the edges traversed by that agent. We consider the mean of all agents' travel distance and the mean of all agents' completion time as the performance measures in our MAPF problems. Our primary metric to maximize is the completion rate (because players will notice if NPCs do not reach their goal locations) but report on the other two performance measures and use them for tie breaking.

As common in videogame pathfinding literature, our search graphs in this thesis are based on rectangular grids with each grid cell being a single node in the graph. Each grid cell has up to eight immediate neighbors. It is connected to them via cardinal edges of cost 1 and diagonal edges of cost $\sqrt{2}$.

2.1.2 Single-agent Pathfinding

Single-agent search is a special case of multi-agent search where the number of agents is set to one. In single-agent search we remove the ability to traverse the self-loop. The objective for our single-agent pathfinding problem is to find a real-time heuristic search algorithm that reduces the distance travelled by the agent. In other words, we are looking to reduce the *suboptimality* α of the agent, where α is the ratio between distance travelled by the agent a_i and shortest distance possible from n^i_{start} to n^i_{goal}. Lower ratios are desired, and a suboptimality of 1 means the agent traversed the shortest possible path.

2.2 Automatic Algorithm Selection

An automatic algorithm selection optimization problem (Rice, 1976) is defined by a tuple $\langle \mathcal{I}, \mathcal{P}, Q \rangle$, where $\mathcal{I} = \{i_1, i_2, \ldots\}$ is a set of problem instances, $\mathcal{P} = \{p_1, p_2, \ldots\}$ is a portfolio of algorithms that can be used to solve each instance $i \in \mathcal{I}$ and $Q : \mathcal{P} \times \mathcal{I} \to \mathbb{R}$ is a function that returns the quality of the solution found by an algorithm $p \in \mathcal{P}$ when solving problem instance $i \in \mathcal{I}$.

A solution to this problem is a mapping $\pi : \mathcal{I} \to \mathcal{P}$ that maps each problem instance $i \in \mathcal{I}$ to an algorithm $p \in \mathcal{P}$. The quality of a solution π is the sum of the qualities of the solutions found by the algorithm prescribed by π for each

problem instance:

$$Q(\pi) = \sum_{i \in \mathcal{I}} Q(\pi(i), i). \tag{2.1}$$

A correct selection is one which maximizes performance out of potential algorithms to choose from. The optimal solution is one that maximizes this value:

$$\pi^* = \arg\max_{\pi} Q(\pi). \tag{2.2}$$

For single-agent pathfinding we consider suboptimality as our quality metric Q and portfolio $\mathcal{P}$, where $\mathcal{P}$ is formed by selecting from a large set of parametrized real-time heuristic search algorithms. For MAPF problem instances $\mathcal{I}$ we consider a small portfolio $\mathcal{P}$ of MAPF algorithms, and we use completion rate as our quality metric Q.[1] We break ties in Q in favour of algorithms that have short distances travelled, followed by algorithms that have small completion time. Remaining ties are broken randomly.

Problem instances $\mathcal{I}$ can be grouped so that selections do not take place for the individual problem i but rather for a a group $\mathcal{I}' \in \mathcal{I}$. In this scenario the selection π is done to optimize the selection for the entire group rather than a individual problem instance.

[1] We equivalently optimize the completion rate averaged over all instances in $\mathcal{I}$ instead of the cumulative one in Equation 2.1 above.

Chapter 3

Related Work

Kotthoff (2014) recently provided a comprehensive survey on algorithm selection. In their research they identify three questions regarding algorithm selection that motivate our research:

- which algorithms to include in the algorithm selection portfolio;
- what granularity is selection performed at (i.e. how are problems grouped);
- how to perform the algorithm selection process.

3.1 Algorithm Selection Portfolio Formation

There are several ways to determine which algorithms to include in a portfolio. Portfolios can be hand-crafted by relying on domain expertise, randomly formed from a pool of algorithms or systematically created. A portfolio should be should be sufficiently diverse such that on wide range of problems at least one algorithm achieves good performance. Domain expertise can be used to either select or develop algorithms that are orthogonal from each other in an attempt to create a diverse portfolio (Samulowitz and Memisevic, 2007). Randomly selecting a subset of algorithms can be done to improve performance over using a single algorithm based of prior knowledge (Vadim Bulitko, 2016b) and can be more effective than forming a group from solely the best overall performing algorithms (Lu and Scott, 2004). Lelis et al. (2016) used

a systematic approach in selecting which heuristics to use for A* (Hart et al., 1968). They proposed a greedy subsetting approach to systematically generate their portfolio of heuristics. They formed a subset $\mathcal{P}'$ from a larger set $\mathcal{P}$ by incrementally adding the heuristic p to $\mathcal{P}'$ which increased the performance the most in conjunction with $\mathcal{P}'$. The process stopped when adding additional heuristics provided no benefits. Conceptually, forming a portfolio of heuristics or configurations of an algorithm is the same as portfolio of algorithms.

Vadim Bulitko (2016b) explored algorithm selection for videogames in their selection of different configurations of a parametrized real-time heuristic search algorithm for single-agent pathfinding. They started by generating a large set of possible algorithms to chose from by randomly selecting from potential configurations. They then examined what amount was actually needed by forming their portfolio by randomly drawing a fixed percentage of algorithms from their generated set of algorithms. This simple approach of randomly selecting configurations to form a portfolio was able to still provide improvements over always using the same algorithm for every problem instance.

3.2 Algorithm Selection Granularity

Vadim Bulitko (2016b)'s exploration of algorithm selection examined two granularities in which they would group the problem instances. The highest granularity they considered was grouping problems by what map they were on. Selecting an algorithm for each group yielded improved results over using a single algorithm for every problem. They then studied selecting algorithms on individual single-agent pathfinding problems. This once again was able to have increased performance over using a single algorithm. It was also better than selecting algorithms on a per-map grouping.

Other fields also vary the granularity for which they conduct algorithm selection. For example, SATzilla (Xu, Hutter, Hoos, and Leyton-Brown, 2008) is a satisfiability (SAT) solver to achieve state-of-the-art results using algorithm selection. In designing their algorithm selection technique they considered four grouping of problems: *Random*, *Handmade*, *Industrial*, and *All*.

Their technique varied based on the problem grouping, mainly by including different algorithms in their portfolio for each problem type.

3.3 Algorithm Selection Process

Vadim Bulitko (2016b)'s study on algorithm selection was able to have a clear increase in performance at lower granularity because of the way each problem instance was mapped to an algorithm. They took two approaches to selecting an algorithm for a problem: (1) Exhaustively run each algorithm in your portfolio and use the one with the best results. (2) Always use the algorithm with best prior results.

Vadim Bulitko (2016b)'s algorithm selection technique operated in an off-line manner, meaning in order to select an algorithm on a new set of problems the portfolio must be evaluated on that problem. Some areas of game development have sufficient knowledge of the problems ahead of time, such as which maps will be in the game, to perform an exhaustive off-line search. However, procedural content generation is becoming more common, where maps are generated at run-time and could prevent such approaches from working. Furthermore, in order to see the gains from the more granular, per-problem, selection another method must be used to facilitate the selection that is less expensive than evaluating each algorithm in the portfolio.

I found no directly related work for alleviating this off-line drawback in selecting videogame pathfinding algorithm selection. However, there is work in related fields that could likely be used to address this problem. In particular, the field of satisfiability solvers has been using predictive models in conjunction with algorithm portfolios to achieving state-of-the-art results. In order for SATzilla (Xu et al., 2008) to achieve state-of-the-art results using algorithm selection it had to facilitate the selection fast enough to not diminish the performance gains of the selection. SATzilla used ridge regression to predict the expected performance of its algorithms and selected the algorithm with the best predict performance to solve the problem. The current state-of-the-art in satisfiability solvers is Cost-sensitive Hierarchical Clustering (CSCH),

which again uses a prediction model to predict which solver to use (Malitsky, Sabharwal, Samulowitz, and Sellmann, 2013). In particular, CSHC uses a random forest model, where each tree votes for the best solver based on the SAT problem. An other recent SAT approach converted the ACII representation of the SAT problem to a grayscale image (Loreggia, Malitsky, Samulowitz, and Saraswat, 2016). This image was than used by a convolutional neural network to predict whether each algorithm in the portfolio would be able to solve the SAT problem. The algorithm with highest predicted chance of solving the problem was than used. This approach was able to achieve better results than using the single best solver from the training data in the satisfiability study (Loreggia et al., 2016).

This thesis explores techniques to facilitate algorithm selection in videogame pathfinding that alleviates requirement of evaluating the portfolio seen in Vadim Bulitko (2016b). We explore both systematic and hand-crafted portfolios combined with image classifications to perform algorithm selection. Videogames inherent visual nature should allow image classifiers to learn which features are relevant to pathfinding algorithm performance.

Chapter 4

Single-Agent Pathfinding Algorithm Selection

This chapter begins by providing an overview of the real-time heuristic search framework we use as our pathfinding algorithm in Section 4.2 and the configurations we use to select our algorithms from (Section 4.3). We then propose our portfolio formation technique (Section 4.4.2), and how we actually perform the algorithm selection (Section 4.4.3). Results for our approach are reported in Section 4.5, followed by a general discussion on how they can be improved in Section 4.6.

4.1 Introduction

We started by expanding the work of Vadim Bulitko (2016b). Their work formed its portfolio by randomly selecting a subset of algorithm configurations from a large pool of algorithms and then evaluated their entire portfolio to facilitate selecting the algorithm to use. We first start by replacing their random approach to algorithm portfolio formation with a more systematic approach. Our portfolio is formed via greedy selection, similar to Lelis et al. (2016), which maximizes the portfolio's performance under the perfect selector π^*. We examine the algorithm selection at the same per-map and per-problem granularity as Vadim Bulitko (2016b). Additionally, we examine two higher granularities, per-game and per-genre. Furthermore, we replace their approach of evaluating each potential algorithm to make a selection with

a machine learning approach where an image classifier automatically selects our algorithm.

4.2 Search Framework

We specifically select which parametrized real-time heuristic search algorithm (Algorithm 1) (Vadim Bulitko, 2016b) to use. We use the same parametrized algorithm as (Vadim Bulitko, 2016c) which fixes the lookahead at 1 (i.e., allow the agent to consider only the immediate neighbours of its current node during the planning stage) and takes the following input parameters: $w, w_c, b, \texttt{lop}, \texttt{da}, \texttt{expendable}$.

The agent will deploy depression avoidance (Hernandez and Baier, 2012) techniques if the parameter $\texttt{da}$ = true, as shown in line 5. If used then line 5 will have the agent's set of local neighbouring nodes $L(n_{\text{current}})$ temporarily set to include nodes which have minimal amounts of learning($|h(n) - h_0(n)|$) (Vadim Bulitko, 2016a). This is done to prevent frequent node revisitation by discouraging agents from revisiting the same nodes right away.

Line 7 consists of the learning rule which utilizes weighted heuristics (Vadim Bulitko, 2016c; Rivera et al., 2013; Rivera, Baier, and Hernández, 2015) and lateral learning (Vadim Bulitko and Sampley, 2016). These parameters are controlled with $w, w_c, \texttt{lop}$, and b. The learning operator is represented by $\texttt{lop}$ which consists of $\texttt{min}, \texttt{max}, \texttt{median}$, and $\texttt{mean}$. w weighs the heuristic update, which can increase the speed at which the heuristic value converges to h^*, but higher weights can lead to inadmissible heuristics that are larger than h^*. w_c is another weighting control that weights the cost of traversing from the current node to the neighbouring nodes. The lateral learning portion of line 7 is defined by the agent's neighbourhood L_b^f as the b fraction of the neighbourhood $L(n_{\text{current}})$ with minimum f values:

$$L_b^f(n) = (n^1, ..., n^{\lfloor b|L(n_{\text{current}})|\rfloor}) \tag{4.1}$$

where $(n^1, ..., n^{\lfloor b|L(n_{\text{current}})|\rfloor}, ..., n^{|L(n_{\text{current}})|})$ is the immediate neighbourhood sorted in the ascending order by their f values. A b value of 1 represent

the full neighbourhood, while a value of 0 represents the single neighbour with lowest f value.

When the control parameter `expendable` is active and a node is deemed expendable (Guni Sharon, Sturtevant, and Felner, 2013) then it is pruned from the graph, as shown in line 9. In order for a node to be expendable all its immediate neighbours must be reached from each other within the immediate neighbourhood (denoted by $\varepsilon(n_{\text{current}})$), as well as learning must have occurred in line 7. The agent moves to its new node in line 10.

Algorithm 1: Parametrized Real-time Heuristic Search

input : search problem $(N, E, c, n_{\text{start}}, n_{\text{goal}}, h)$,
control parameters $(w, w_c, b,$ `lop`, `da`, `expendable`$)$

1 $t \leftarrow 0$
2 $n_{\text{current}} \leftarrow n_{\text{start}}$
3 **while** $n_{current} \neq n_{goal}$ **do**
4   **if** `da` **then**
5         $L(n_{\text{current}}) \leftarrow L_{\text{min learning}}(n_{\text{current}})$
6     $h_{\text{prev}} \leftarrow h(n_{\text{current}})$
7     $h(n_{\text{current}}) \leftarrow$
      $\max\left\{h(n_{\text{current}}), w \cdot lop_{n \in L_b^f(n_{\text{current}})}(w_c \cdot c(n_{\text{current}}, n) + h(n))\right\}$
8   **if** `expendable` *&* $h(n_{current}) - h_{prev} > 0$ *&* $\varepsilon(n_{current})$ **then**
9   remove n_{current} from the search graph;
10     $n_{\text{current}} \leftarrow \arg\min_{n \in L(n_{\text{current}})}(c(n_{\text{current}}, n) + h(n))$
11     $t \leftarrow t + 1$

4.3 Space of Algorithms

We use the same control parameters as Vadim Bulitko (2016c) and reproduce them here for the reader's convenience: $w \in [1, 10]$, `da` $\in \{true, false\}$, `expendable` $\in \{true, false\}$, `lop` $\in \{\min, \text{avg}, \text{median}, \max\}$, $b \in [0, 1]$ which defines a six-dimensional space of real-time heuristic search algorithms. While the space of algorithms that we select from are variations of the parameterized search algorithm, seen in Algorithm 1, conceptually these can be thought of as separate algorithms and some of the combinations were originally developed as standalone algorithms.

4.4 Our Approach

We use the three questions Kotthoff (2014) proposed for algorithm selection to guide our approach. First step is to identify at what granularity the selection will take place, second is to determine the algorithms to include in the portfolio for each granularity, and lastly how will the actual selection be made for each granularity.

4.4.1 Algorithm Selection Granularity

Before defining which algorithms to include in our portfolio and how to select them we determined at which granularity to select them. We will always perform our selection once at the start of a problem and use that choice for the remainder of the problem. We chose to examine four granularities, including two used by Vadim Bulitko (2016b):

- *Per-genre:* The algorithm selection is determined by the videogame genre the input is from.
- *Per-game:* The algorithm selection is determined by which videogame the input is from.
- *Per-map:* The algorithm selection is determined by examining the map the agent will be traversing.
- *Per-problem:* The algorithm selection is determined by examining both the map and specific problem the agent will be solving.

Per-genre and per-game rely on predefined labels that are assigned based upon specific videogame characteristics.

4.4.2 Portfolio Formation

We create potential algorithms A by drawing parametrizations uniform randomly from the algorithm space defined in Section 4.3. Each of these algorithms is then ran on a set of problems $\mathcal{I}$. We use the results of how these

algorithms performed to create the portfolio of algorithms that our selection procedure will pick from.

The large performance gains seen in state-of-the-art algorithms come from specialized algorithms that excel at a very specific set of problems, often at the cost of general performance (Vadim Bulitko, 2016b). Using a randomly formed algorithm portfolio was suitable for Vadim Bulitko (2016b) because they always selected the best algorithm from the portfolio and so as long as one algorithm in the portfolio performed well it did not matter if the rest were poor performers. Since our algorithm selection is done in an on-line fashion, where we are not guaranteed to select the optimal algorithm for a given problem, we must attempt to ensure only high performing and diverse algorithms are able to be selected.

To reduce the potential harm from selecting an suboptimal algorithm we reduce the portfolio to a subset of high performing and complementary algorithms. Similar to Lelis et al. (2016), we form our subset of algorithms using a greedy portfolio formation shown in Algorithm 2.

Algorithm 2 forms a portfolio by greedily adding the algorithm which maximizes the portfolios performance under the perfect selector π^*. The algorithm takes a set of algorithms A with a target portfolio size n indicating the maximum number of members in the portfolio. If none of the remaining algorithms increase the portfolio's performance under π^* than the portfolio does not increase. The subset of algorithms is represented by the set $\mathcal{P}$ and initialized to a empty set in line 1. Line 3 finds algorithm a from the set of algorithms A which has the lowest suboptimality in combination with portfolio $\mathcal{P}$ defined by f. The algorithm is then added to the set in line 4.

Algorithm 2: Greedy Portfolio Formation

input : A, n
output: $\mathcal{P}$
1 $\mathcal{P} \leftarrow \emptyset$
2 **for** $i \in \{1, \ldots, n\}$ **do**
3 $\quad a \leftarrow \arg\min_{a \in A} f(\mathcal{P}, a)$
4 $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{a\}$

4.4.3 Automatic Algorithm Selection

We can now create the mapping π of our problem instance $i \in \mathcal{I}$ to our algorithm $p \in \mathcal{P}$. For each of the granularities our process varies slightly, but uses the same general idea to create π. Our automatic algorithm selector π is a traditional image classifier, where given some input image the classifier predicts the corresponding label. For us the input is a visual representation of pathfinding problem instance and the label is used to determine the algorithm to use for the input.

Table 4.1: Summary of each granularities algorithm selection mapping

Approach	Input	Output	Additional Mapping	Section
per-genre	map	genre	$\{(genre, p \in \mathcal{P})\}$	4.5.1
per-game	map	game	$\{(game, p \in \mathcal{P})\}$	4.5.2
per-map	map	$p \in \mathcal{P}$	N/A	4.5.3
per-problem	map + start & goal	$p \in \mathcal{P}$	N/A	4.5.4

Per-genre: For our per-genre selection we use a black and white image of the map that the agent will be traversing, where white represents impassable obstacles and black represents open nodes. We determine the best algorithm to use for each videogame genre in $\mathcal{I}_{\text{training}}$. Each genre in $\mathcal{I}_{\text{training}}$ is then mapped to the corresponding algorithm that performed best for that genre. We then train a convolutional neural network (CNN) image classifier to predict which genre each map in $\mathcal{I}_{\text{training}}$ belongs to. When using this approach to determine which algorithm to use for a problem, an image representing the map the agent will be traversing is the input for the network to predict the genre the map belongs. The algorithm mapped to that genre is then ran on all problems on that map. We can see an example of the per-genre selection processes in Figure 4.1.

Per-game: Our per-game selection is similar to per-genre selection. We use the same black and white image of the map the agent will be traversing. We determine the best algorithm to use for each videogame game in $\mathcal{I}_{\text{training}}$. Each game in $\mathcal{I}_{\text{training}}$ is then mapped to the corresponding algorithm that performed

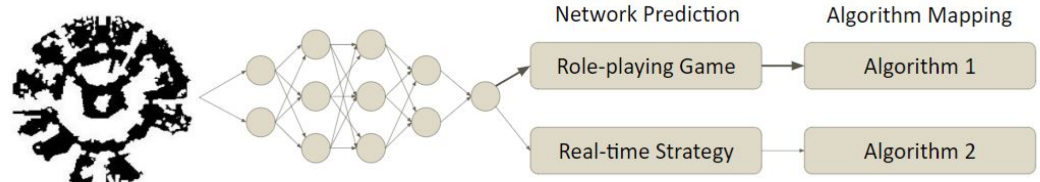

Figure 4.1: Example for algorithm selection based off of which genre the game belongs to.

best for that game. We then train a CNN to predict which game each map in $\mathcal{I}_{\text{training}}$ belongs to. When using this approach to determine which algorithm to use the image representing the map the agent will traverse is inputted into the network which predicts the game the map belongs to. The algorithm mapped to that game is then ran on all subsequent problems on that map. We can see an example of the per-game selection processes in Figure 4.2.

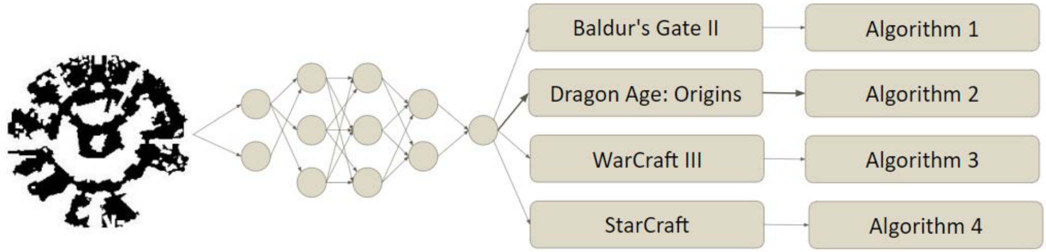

Figure 4.2: Example for algorithm selection based off of which game the map is from.

Per-map: We use the same black and white image of the map the agent will be traversing. We then use Algorithm 2 to form our portfolio $\mathcal{P}$. In the previous two granularities there was only one algorithm assigned to each label and as such did not need to use Algorithm 2. For each map in $\mathcal{I}_{\text{training}}$ we assign the label representing the best performing algorithm on that map from $\mathcal{P}$. We then train a CNN to predict which algorithm to use for each map in $\mathcal{I}_{\text{training}}$. When using this approach to determine which algorithm to use the image representing the map the agent will traverse is inputted into the network which predicts the algorithm that will perform best on that map. The predicted algorithm is then ran on all subsequent problems on that map. We can see an example of the per-map selection processes in Figure 4.3.

Removing predefined labels allows for grouping maps based on the algorithms that perform best on them and not some predefined label. For a game designer to use this approach they would require intimate knowledge of each algorithm in their portfolio's performance and how it relates to the maps they have been used on. The trained network instead will learn the features related to each algorithm's performance.

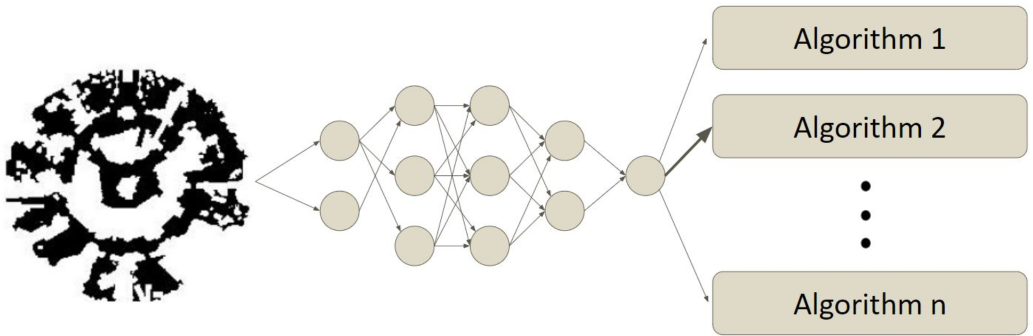

Figure 4.3: Example for algorithm selection on a per-map basis.

Per-problem: Our per-problem selection differs greatly from the previous selections. We use the same black and white image of the map the agent will be traversing as the base for our input. We additionally add a 3×3 green pixel at the start and a 3×3 red pixel at the goal and add a single-pixel-wide line connecting the start and goal. We also reduced the problems to a single map $\mathcal{I}'_{\text{training}} \in \mathcal{I}_{\text{training}}$. We did so to simplify the problem classification to only having the configurations of start and goals locations change and not the map itself. We use Algorithm 2 to form our portfolio $\mathcal{P}$. For each problem in $\mathcal{I}'_{\text{training}}$ we assign the label representing the best performing algorithm from $\mathcal{P}$ on that problem. We then train a CNN to predict which algorithm to use for each problem in $\mathcal{I}'_{\text{training}}$. When using this approach to determine which algorithm to use the image representing the problem the agent will be solving is inputted into the network, which then predicts the algorithm that will perform best on that problem. The predicted algorithm is then ran on that problem. We can see an example of the per-problem selection process in Figure 4.4.

Per-problem selection has the most potential for improvement as it is our

most granular selection. It provides the network details about the exact problem the algorithm is solving. It would be difficult to imagine a way for a game designer to use this approach without some automated selection process as there would likely be far too many problems for them to specify an algorithm manually on a per-problem bases.

We provide a summary table of each granularities approach in Table 4.1. Per-genre and per-game require additional mapping as they use a prior approach to determine the algorithm to use aided by automatic classification of the input. Per-map and per-problem do not require any additional mapping as they select the algorithm to use directly. The input to the network is the same for the per-map, per-genre, and per-game, with per-problem adding additional information to input.

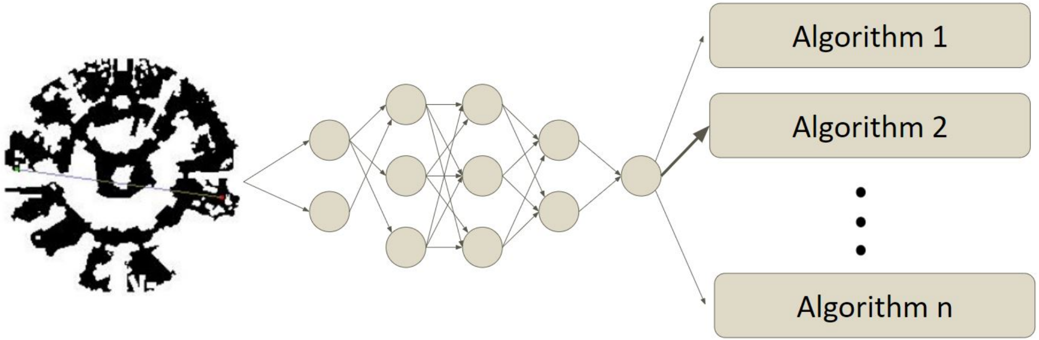

Figure 4.4: Example for algorithm selection on a per-problem bases.

4.5 Empirical Evaluation

For all of our evaluations we use AlexNet, a readily available deep neural network, as our image classifier[1] included in MATLAB 2017a Neural Network Toolbox. We chose AlexNet as it commonly available in many deep learning frameworks, such as PyTorch, Cognitive Neural Tool Kit, TensorFlow, MATLAB and other deep learning frameworks (Facebook, 2017; Google, 2017; Microsoft, 2017; Vedaldi and Lenc, 2015). We want our approach to be accessible to game developers who likely would not have an extensive background

[1] pre-trained on the ImagenNet dataset

in artificial intelligence. The input for our classifications is a 227×227 colour pixel image which represents the problem. For all selections the image is bicubically resized to fit this input.

The search problems are on the 342 maps in the MovingAI benchmarks (Sturtevant, 2012). For per-genre, per-game, and per-map selections each map has 50 problems created with randomly connected start and goals. This provides a benchmark of 17100 (342×50) search problems. We create 1000 configurations to be ran on the problems as defined in Section 4.4.2. Our problem based selection was limited to a single map, Lak506d, from *Dragon Age: Origins* and we used the 1169 problems that are provided for that map from the MovingAI benchmark. The map was chosen for its relatively similar size to the network as well as containing a large number of problems in the MovingAI benchmark. We provide a visualization of this map in Figure 4.4.

For each of the different granularities we evaluate our approach using 20 random partitions of the problem set $\mathcal{I}$ in to $\mathcal{I}_{\text{training}}$ and $\mathcal{I}_{\text{testing}}$. For per-genre, per-game, and per-map the split is done for entire maps. Where $\mathcal{I}_{\text{training}}$ consists of 256 (75%) random maps of the possible 342 maps and $\mathcal{I}_{\text{testing}}$ has the remaining 86 (25%) maps. For per-problem selection the 1169 problem instances $\mathcal{I}$ are split randomly with 877 (75%) being used for training and 292 (25%) being used for testing. We compare our network based selection π to the perfect classifier π^* and the best single pathfinding algorithm for each granularity.

4.5.1 Per-genre Algorithm Selection

Moving AI contains role-playing games (RPG) and real-time strategy (RTS) games. The best performing algorithm on RPG and RTS maps are not guaranteed to be different, but were in our case. There are 231 role-playing game maps in Moving AI from *Baldur's Gate II* (BioWare, 1998) and *Dragon Age: Origins* (BioWare, 2009). The remaining 111 maps are from the real-time strategy games from *StarCraft* (Blizzard Activision, 1998) and *WarCraft III* (Blizzard Entertainment, 2002).

Our genre classification achieved an average accuracy of $96 \pm 2.2\%$. The

average suboptimality achieved by our network π and the perfect selector π^* was 12.33 ± 1.0 while the suboptimality achieved by using the algorithm that performed best across $\mathcal{I}_{\text{training}}$, which we refer to as *prior*, was 12.58 ± 1.1 in comparison. Using the network resulted in better performance in 14 out of 20 of the splits of $\mathcal{I}$. It is worth noting here that π^* is a network which predicts the genre correctly every time and not a network that predicts the best algorithm.

Table 4.2: Results of per-genre selection

Algorithm Selection Type	Accuracy	Suboptimality
Prior	N/A	12.58 ± 1.1
π	$96 \pm 2.2\%$	12.33 ± 1.0
π^*	$100 \pm 0.0\%$	12.33 ± 1.0

4.5.2 Per-game Algorithm Selection

We next consider the task of selecting the best algorithm from our 1000 possible algorithms for each of the four Moving AI games in our 256 map training set. The best single algorithm is the algorithm which did best across all 256 maps. π^* is again a classifier with 100% accuracy meaning it always picks the correct game the map is from but not necessarily the best algorithm for that map. In each of our 20 splits of $\mathcal{I}$ there was always at least one map from each game in both $\mathcal{I}_{\text{training}}$ and $\mathcal{I}_{\text{testing}}$

Our ability to classify which game a map belonged to obtained a lower accuracy than the genre classification with a accuracy of $90 \pm 3.1\%$. The average suboptimality for π^* is 12.52 ± 1.3, our network selection resulted in an average suboptimality of 12.54 ± 1.3, and using the best single algorithm achieved a suboptimality of 12.72 ± 1.2. Selecting an algorithm dynamically resulted in better performance 13 out of the 20 trials.

4.5.3 Per-map Algorithm Selection

π^* in this scenario represents both a classifier with perfect accuracy and the lower bound for suboptimality achievable when using our map-based selec-

Table 4.3: Results of per-game selection

Algorithm Selection Type	Accuracy	Suboptimality
Prior	N/A	12.72 ± 1.2
π	$90 \pm 3.1\%$	12.54 ± 1.3
π^*	$100 \pm 0.0\%$	12.52 ± 1.3

tions. The per-map selection with a portfolio size of 2 achieved an accuracy of $55 \pm 5.3\%$ for selecting the best algorithm to use for a given map out of the 2 algorithms. The average result of the network selected algorithms were a suboptimality of 12.20 ± 1.2 in comparison to the best single algorithm achieving a result of 12.27 ± 1.3. π^* achieved a suboptimality of 11.27 ± 1.1. π achieved better results than the best single algorithm on $\mathcal{I}_{\text{training}}$ on 12 out of the 20 trials.

Table 4.4: Results of per-map selection

Algorithm Selection Type	Accuracy	Suboptimality
Prior	N/A	12.27 ± 1.3
π	$55 \pm 5.3\%$	12.20 ± 1.2
π^*	$100 \pm 0.0\%$	11.27 ± 1.1

4.5.4 Per-problem Algorithm Section

Our testing was limited to lak506d, a map from *Dragon Age: Origins*, which was slightly smaller (207×196) than our network input. There are 1169 problems on the map. Using a portfolio size of 2 again the suboptimality achieved by the network was 14.20 ± 0.7 in contrast to the best single algorithm achieving 14.77 ± 0.5. The accuracy of network was $70 \pm 2.3\%$. π^* had a suboptimality of 11.35 ± 0.5. π achieved a better average performance than using the best single algorithm from $\mathcal{I}'_{\text{training}}$ on 18 out of the 20 trials.

Table 4.5: Results of per-problem selection

Algorithm Selection Type	Accuracy	Suboptimality
Prior	N/A	14.77 ± 0.5
π	$70 \pm 2.3\%$	14.2 ± 0.7
π^*	$100 \pm 0.0\%$	11.35 ± 0.5

4.6 Discussion

Our initial hypothesis that an off-the-shelf neural network could be used to select which algorithm to use from a portfolio of algorithm without running them can be done such that using the selections would increase performance over using the best algorithm based off prior knowledge was verified in our initial testing. We saw that an off-the-shelf network was able to pick up the differences between maps to successfully classify which genre and which game a map was from. Its ability to select which algorithm to use directly without these predefined labels was less successful. The accuracy for the predefined labels were above 90% while per-map and per-problem achieved 55% and 70% respectively.

The results for using the algorithm based off prior information, π and π^* were in line with previous work (Vadim Bulitko, 2016b), in that more granular selections yield greater room for performance increases. The room for improvement between using the single best algorithm and π^* in the per-genre, per-game and per-map were relatively small with per-map only having a difference of 12.27 and 11.27, respectively. Per-problem however, offers a larger potential for improvement, seen in Table 4.5. Here the best single algorithm achieves a suboptimality of 14.77 ± 0.5 and represents the best possible outcome for a per-map selection on that map, as it would be the correct choice of a per-map algorithm selection on that map. Performing the selection on a per-problem granularity could result in a substantial improvement in suboptimality to 11.35 ± 0.5[2].

This motivated us to pursue improving the ability for a network to select algorithms on a per-problem basis. We did not want to use custom networks . We wanted our approach to allow for easy adoption by game developers and not one that requires an advance understanding of image classification or neural networks.

[2]Large suboptimality values are impractical. Path lengths can be decreased via increasing the lookahead from an agent's immediate neighbours. We restricted the lookahead as there is a dominant strategy for decreasing suboptimality with a lookahead of infinity which results in A*'s path.

Chapter 5

Automatic Algorithm Selection for Multi-agent Pathfinding

In this chapter we provide an overview of the three MAPF algorithms used in our portfolio (Section 5.2) and why we selected them (Section 5.3.2). We explain how we perform our selection in Section 5.3.3. We then explain our testing environment in Section 5.4.1 and our results in Section 5.4.2.

5.1 Introduction

In developing our portfolio, which contained two contemporary MAPF algorithms *Flow Annotated Replanning* and *Windowed Hierarchical Cooperative A**, we created a complementary algorithm *Bounded Multi-Agent A**(BMAA*). BMAA* is a real-time multi-agent pathfinding algorithm. We provide a brief description below and reproduce Sigurdson et al. (2018) in Appendix A for the reader convenience.

We forgo the portfolio formation technique of Chapter 4 in favour of a hand-crafted portfolio. We did so based on our experience using these MAPF algorithms where each algorithm had clear scenarios where they performed well, in contrast to selecting configurations. We also synthetically generate MAPF problems based off common videogame scenarios. This was done to create a more diverse set of examples for how each algorithm might perform in these scenarios.

5.2 MAPF Algorithms

While researchers have proposed a number of algorithms to solve MAPF problems (de Wilde, ter Mors, and Witteveen, 2013; G. Sharon, Stern, Felner, and Sturtevant, 2015; C. Wang and Botea, 2011), in this chapter, we focus on using methods that are better suited for environments where agents must take actions within a very small amount of time (e.g., videogames). These algorithms are not guaranteed to have every agent reach their goal. There are algorithms, such as Conflict Based Search(CBS) G. Sharon et al., 2015, which has completion guarantees but require finding a complete solution before moving. This can leave the agents frozen in place has the solutions is being planned. The nature of videogames can have an agent's goal change in a moments notice rendering the paths obsolete causing the planning to restart. This makes CBS, and other similar algorithms, impractical for use in videogames and thus not considered in our experiments.

The more agents there are on a map the more difficult it is to have all agents reach their goal. We therefore focus on increasing the completion rate to have more agents reach their goal. Game designers can increase the likelihood of all agents reaching their goal by reducing the number of agents. Our portfolio is comprised of three scalable A*-based algorithms: *Windowed Hierarchical Cooperative A** (WHCA*) (Silver, 2005), *Flow Annotation Replanning* (FAR) (Wang and Botea, 2008), and *Bounded Multi-Agent A** (BMAA*) (Sigurdson et al., 2018). We choose these algorithms because they use different strategies to solve MAPF problems and, as a result, can excel on different MAPF problems.

5.2.1 Windowed Hierarchical Cooperative A*

Silver (2005) proposed a family of A*-based algorithms for solving MAPF problems: In *Cooperative A**, each agent runs an A* search in a three dimensional graph (x-coordinate, y-coordinate, and time) to reach its goal and shares its plan with other agents through reservation tables. Therefore, the agents are able to avoid collisions since each agent knows where all the other

agents will be and when they will be there. To improve scalability, *Hierarchical Cooperative A** uses hierarchical search, where each agent uses the length of the shortest path found in an abstracted state space as a guiding heuristic. Finally, to further improve scalability, Windowed Hierarchical Cooperative A* limits the search depth of each agent to within a window. Once a partial path within the window is found, the agent follows it and searches for the next partial path by shifting the window along the current path.

5.2.2 Flow Annotation Replanning

Like WHCA*, *Flow Annotation Replanning* (FAR) (Wang and Botea, 2008) also takes account of other agents plans. However, instead of searching for a new path when the current one is blocked, agents in FAR simply wait at their current nodes until they can reserve their next set of moves in a reservation table. FAR also detects deadlocks, where agents would wait on each other indefinitely, and forces them to move away from their current nodes and to replan their paths.

To reduce the likelihood of agents blocking each other, FAR annotates at each node with the direction that agents at that node should follow. These annotations combined creates "highways," where agents can quickly move from one end of the map to another without ever stopping to wait for other agents. Figure 5.1 shows an example map annotated by FAR which uses the following strategy: It first creates an edge-less annotated graph G' that has the same set of nodes as the original graph G. Then, edges are added to G' in alternating directions, that is, even-numbered rows are assigned west-bound edges and odd-numbered rows are assigned east-bound edges. Similarly, even-numbered columns are assigned north-bound edges and odd-numbered columns are assigned south-bound edges. Additional edges are added in special cases. For example, nodes on corridors that are only one-node wide retain their bi-directional connectivity. Self-loops are always retained as agents always have the ability to wait, however self-loops are not considered in the search.

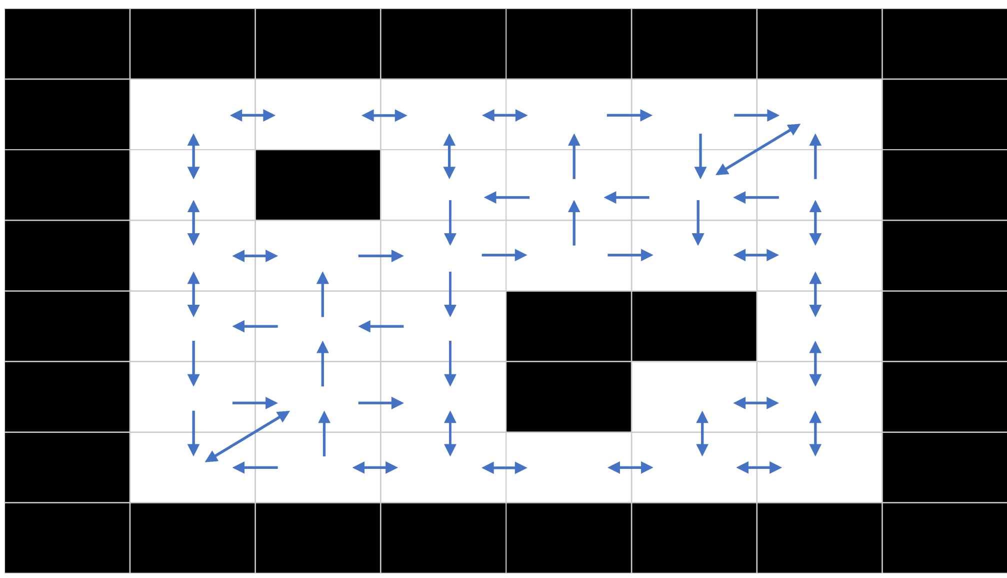

Figure 5.1: Example of Flow Annotations.

5.2.3 Bounded Multi-Agent A*

*Bounded Multi-Agent A** (BMAA*) (Sigurdson et al., 2018) is based on a *Real-Time Adaptive A** (RTAA*) (Koenig and Likhachev, 2006), a well-known *single-agent* real-time heuristic search algorithm. RTAA* runs the following procedures iteratively until the agent reaches its goal: (1) perform a bounded-depth A* search from the agent's current position; (2) update the heuristic values of all nodes in the closed list of that A* search to make them more informed; and (3) move the agent along the partial path returned by that A* search. Sigurdson et al. (2018) extended RTAA* to a *multi-agent* setting, where other agents are treated as (moving) obstacles during the search. Additionally, each agent is able to request other agents that are currently located on its goal cell to vacate. The vacating agent will move to any available neighboring node and resume its regular search procedures from its new location. See Appendix 3 for a more in depth explanation of A* and Appendix A.5 for a more in depth explanation of BMAA*.

5.3 Our Approach

We once again are motivated by the three questions raise by Kotthoff (2014) that must be addressed to solve our algorithm selection problem defined in Section 2. Theses questions are key to being successful in algorithm selection: (1) Which granularity to perform the selection? (2) What algorithms to include in the portfolio $\mathcal{P}$ of algorithms? (3) How will the mapping π decide which algorithm $p \in \mathcal{P}$ to use for the problem $i \in \mathcal{I}$ be performed?

5.3.1 Algorithm Selection Granularity

For multi-agent pathfinding we only considered selecting an algorithm at the per-problem granularity. We decided to focus only on the per-problem granularity as it has the most potential for improvement out of the four previously examined granularities.

5.3.2 Portfolio Formation

Ideally, the the portfolio of algorithms should be sufficiently diverse so that for each possible problem instance, there exists at least one algorithm in the portfolio that does well on that problem instance. Larger portfolios, however, may slow down the learning process as well as result in lower performance due to selection errors. Therefore, in this chapter, we consider a relatively small but diverse set of algorithms: Windowed Hierarchical Cooperative A* (WHCA*), Flow Annotation Replanning (FAR), and Bounded Multi-Agent A* (BMAA*) as described in Section 5.2. We choose BMAA* because we anticipate that it will do well in problem instances where well-informed heuristics are available. In problem instances where heuristics are more substantially misleading, it performs poorly as it runs only bounded-depth searches to find partial paths for the agents, which may be in the wrong direction. Conversely, FAR computes complete paths, but has a limited gridlock breaking procedure that can lead to failure in particularly congested problems. Finally, we choose WHCA* as it can solve particularly tricky problems as it communicates agents plans and takes them into account through a windowed cooperative search. Doing

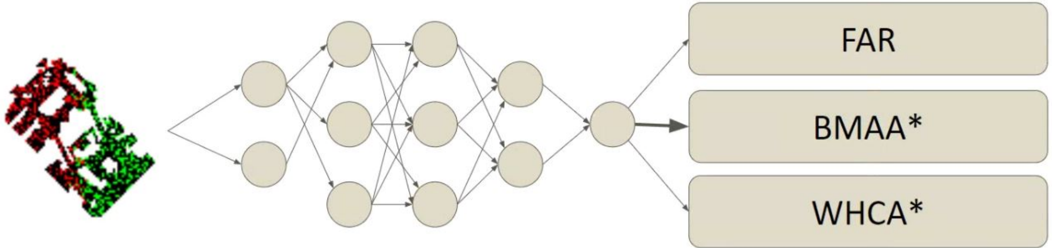

Figure 5.2: Multi-agent algorithm selection example.

so, however, is computationally expensive and is not always necessary. We forwent the previous systematic approach in favour of a hand-craft approach because our experience using multi-agent pathfinding suggested that each of three aforementioned algorithms should have specific problems where they excel. We likely could have used the systematic approach to tune each of the algorithms in our portfolio, but instead used the authors' recommend configurations which were consistent with our preliminary experiments.

5.3.3 Automatic Algorithm Selection

Similar to the case of single-agent pathfinding, we treat the MAPF algorithm-selection problem as an image classification problem. Thus, we classify each image (a representation of a MAPF problem instance) with respect to a fixed set of possible labels (algorithms from the portfolio).

In order to provide MAPF problem instances as inputs to the CNN, we represent each problem instance as an image, where blocked and unblocked node are represented by white and black pixels, respectively. A partial agent specification is also provided as part of the input, represented by a single pixel. A start location is indicated with a green pixel and a goal location with a red pixel (Figure 5.2). The mapping between a specific agent's start and goal is anonymized as there is no information in the images as to between which goal belongs to which agent. We resize the image to that of the network input.

5.4 Empirical Evaluation

We again use AlexNet (Krizhevsky, Sutskever, and Hinton, 2012)to solve the image classification problem. We choose a deep learning approach with the anticipation that it will be able to learn to recognize important features of the problem (e.g., topologies of the map, distribution of agents, etc.) and exploit them automatically (Sigurdson and Bulitko, 2017). We once again choose AlexNet as it is a common, readily-available CNN, which requires no additional engineering.

We use 20 video-game maps from *Baulder's Gate II* available in the MovingAI benchmark (Sturtevant, 2012). This included the 10 largest maps from the game used to originally evaluate FAR (Wang and Botea, 2008) and 10 additional small and mid-sized maps. We included more maps to get a more diverse set of maps ranging from 564 traversable nodes to 51586 nodes in size. We fixed the number of agents to 300 for each MAPF problem. For each map we created 20 MAPF problems for each of the problem types defined in Section 5.4.1. If an agent is not at their goal at the 30 second time limit then its goal achievement time is artificially set to 30 seconds.

We split the dataset randomly into 70% for training and the other 30% for testing. All of the performance metrics are reported for the test set. We perform this splitting over 10 trials and report the results averaged over the 10 trials.

5.4.1 MAPF Problem Generation

Rather than using only randomly selected start and goals, we generated different types of MAPF problems. We ensure that start and goals are in the same connected component for all the problems generated. These problem types are intended to represent some common videogame scenarios. For example, agents *swapping sides* is commonly seen in many strategy games where agents are trying to reach an opposing team's base. *Tight to wide* happens when there is a common spawn location and some general area that the agents are trying to reach. These are not intended to represent all scenarios that occur in

Figure 5.3: Three resized sample MAPF problem types with start (green pixels) and goals (red pixels): outside in (left), cross sides (centre), and switch sides (right).

games but rather provide a more diverse set of MAPF problem types instead of relying on only the random problem type. The definition and an example of each problem type can be seen in Figure 5.4.

5.4.2 Per-problem Algorithm Selection

We compare the results of our network π against the results of each algorithm in our portfolio (i.e., WHCA*, FAR, and BMAA*), a *perfect selector* π^* and the *worst selector* possible, which always selects the worst algorithm for the problem. We set the parameters for the algorithms in our portfolio to the following: the size of reservation tables used by FAR is 3; the window size of WHCA* is 8, the lookahead value of BMAA* is 32 and BMAA* has flow annotations turned off. We chose these values as they performed well in our initial testing. All the algorithms were implemented in C#.

Figure 5.5 shows completion rate averaged over 10 random splits of training and test data. We provide more details in Table 5.1. A perfect selector would result in a 80.8% completion rate (Table 5.1). Our approach achieves a completion rate of 76.6%, which is between the best single algorithm (FAR), with the completion rate of 66.1%. FAR is followed by BMAA* with a completion rate of 65.7% and WHCA* with a completion rate of 54.6%. If one were to consistently select the worst performing algorithm, the completion rate would drop to 44.3%.

Problem Type	Definition	Example
Random	Agents are assigned random start and goals.	
Cross sides	Agents start on one side (e.g. bottom) and traverse to the opposing side.	
Swap sides	Half of the agents start on one side (i.e. left) while the other half start on the opposite side with goals on the opposing side.	
Inside out	Agents start near the center and are assigned goals near the outer edges of the map.	
Outside in	Agents start near the edge and are assigned goals near the center.	
Tight to tight	Agents start close together and are assigned goals close together elsewhere on the map.	
Tight to wide	Agents start close together and are assigned goals that are in the same general area of the map.	

Figure 5.4: Problem type definitions and examples.

Table 5.1: Algorithm performance on all problems.

	Completion Rate(%)	Distance	Completion Time (s)
π^*	$\mathbf{80.8 \pm 0.8}$	283.1 ± 7.5	15.5 ± 0.2
π	76.6 ± 1.2	261.0 ± 8.8	16.2 ± 0.4
BMAA*	65.7 ± 0.7	465.7 ± 5.9	$\mathbf{14.4 \pm 0.2}$
FAR	66.1 ± 1.0	405.7 ± 10.1	15.9 ± 0.2
WHCA*	54.6 ± 1.1	$\mathbf{88.3 \pm 1.7}$	21.7 ± 0.2
Worst	44.3 ± 0.7	328.6 ± 11.0	19.7 ± 0.2

On the distance travelled metric, which breaks completion rate ties, π significantly improves upon BMAA* and FAR with a 44% (from 465.7 to

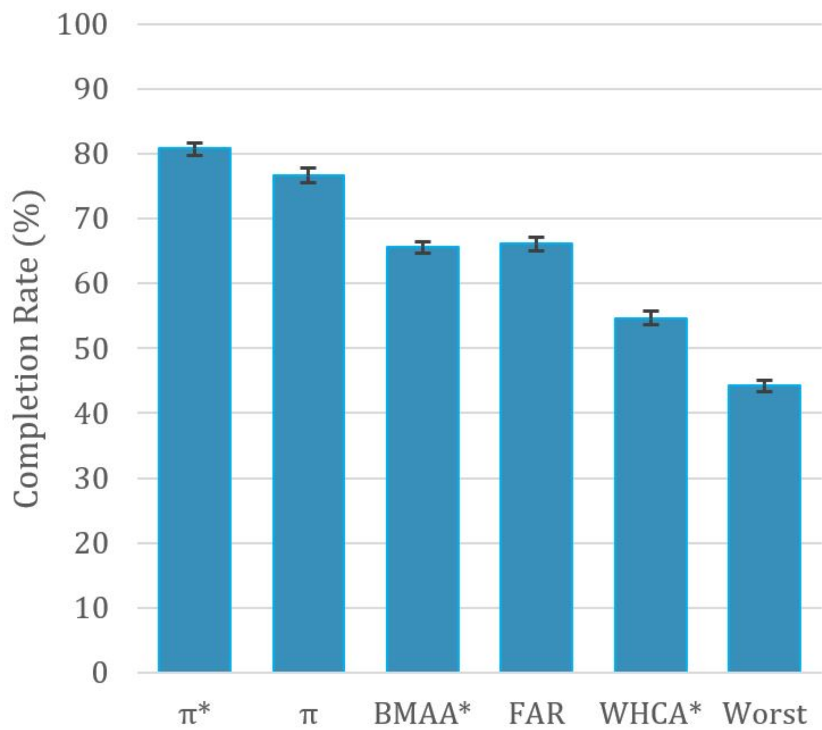

Figure 5.5: Completion Rate % for the test problems averaged over 10 splits of the data.

260.9) and 36% (405.7 to 260.9) reduction, respectively. However, the best single algorithm on this metric is WHCA* with 88.3. Our distance metric is recorded for all agents and not just agents who reach their goal. In other words, an algorithm that never moves the agents would have the smallest distance despite not completing the task. As distance was only for breaking ties in our model, this was not of great concern.

On the goal achievement time metric, BMAA* is better than π with a 11.1% (from 16.2s to 14.4s) improvement. On this metric, BMAA* is followed up by WHCA* with a goal achievement time of 15.9s and FAR with 21.7s. The reason why both WHCA* and FAR are slower despite the agents travelling smaller distances than BMAA* is that many of the agents end up waiting for other agents due to their use of reservation tables. WHCA* also performs a more expensive three-dimensional search.

Over the 10 splits of the training and test data, BMAA*, FAR, and WHCA* was the best choice for an average of 283.5 (33.8%), 258.9 (31.0%),

Table 5.2: Completion rate(%) by problem type.

Problem Type	BMAA*	FAR	WHCA*
Random	**79.4**	75.3	68.9
Cross sides	69.4	**83.7**	56.4
Swap sides	**64.4**	48.5	34.5
Inside out	**76.7**	75.3	59.9
Outside in	**73.3**	72.3	60.7
Tight to tight	33.8	**37.8**	36.7
Tight to wide	59.2	**71.0**	54.9

and 296.6 (35.3) problems, respectively. The neural network predicted BMAA*, FAR, and WHCA* to be the best choice for an average of 269.0, 249.4, and 321.6 problems, respectively. The networks selected the correct algorithm 72.9% of the time.

WHCA* is often the best choice despite its low completion rate, 54.6 ± 1.1 (%) across all problem types. WHCA* distance travelled makes it the best choice on problems where all three algorithms achieve 100% completion rate. This leads to its frequent choosing despite its completion rate.

5.5 Discussion

We demonstrated that using an off-the-shelf deep neural network to automatically select MAPF algorithms from a portfolio can improve the performance over the individual algorithms in that portfolio. This approach is promising since it does not require designing new MAPF algorithms. Furthermore, with deep learning, human designers do not even have to hand-craft a set of features to describe MAPF problem instances. As a result, this process is accessible to a broad range of game developers. Similar ideas were recently explored for SAT/CSP solver selection (Loreggia et al., 2016). However, our approach simplifies the process one step further by using an off-the-shelf deep neural network (AlexNet) in place of a custom-built CNN.

Chapter 6

Discussion and Future work

Our current approach uses actual videogame maps but not the actual problems that were ran on those maps. Additionally, videogame maps can be arbitrary sizes and may not be suitable to simple resizing. Future work will extend the evaluation to more realistic problems by data mining from actual gameplay logs, as well as address a more diverse set of maps which may not be suitable to resizing. We will investigate assigning algorithms to agents as the pathfinding problems progress through out the course of a game. We will also examine a per-agent basis (at least a per-agent-cluster basis) assignment, as one algorithm is unlikely to be the best choice for all agents within a MAPF problem.

Another area we will explore is understanding the features the network is learning. Understanding the reasoning behind why the network makes a certain decision can help guide future development of algorithms as one learns in which scenarios an algorithm fails.

Chapter 7

Conclusion

We demonstrated that an off-the-shelf deep neural network can be used to perform algorithm selection for videogame pathfinding. We demonstrated that this can be done for selecting parameters for an algorithm as well as selecting between different algorithms. The results demonstrated a significant improvement in the more difficult multi-agent pathfinding environment. Furthermore, we proposed a systematic approach to creating a portfolio of algorithms as well as demonstrated the robustness of our approach to different performance metrics (suboptimality and completion rate).

This approach is promising as it does not require developing custom networks to yield improved results. Our overall goal of being able to use machine learning to select which algorithm to use so that using a portfolio of algorithm is better than any of the individual algorithms was successful. Videogame engines should be able to harness the benefits of algorithm selection without using expensive sampling approaches or requiring the developer to have intimate knowledge of the intricacies of the available algorithms. Games within a single engine are diverse indicating that in practice there should be room for performance gains via algorithm selection.

In conclusion, this thesis we presented a systematic approach to greedy algorithm portfolio formation, creating a mapping π successful enough to improve results over contemporary search algorithms. Additionally we developed BMAA*, a new real-time heuristic search pathfinding algorithm.

References

Bandi Namco. (2017). Tekken 7. Retrieved June 28, 2018, from https://tk7.tekken.com/

BioWare. (1998). Baldur's Gate. November 30, 1998. Interplay. Retrieved from http://www.bioware.com/bgate/

BioWare. (2009). Dragon Age: Origins.

Björnsson, Y., Bulitko, V., & Sturtevant, N. (2009). TBA*: Time-bounded A*. In *Proceedings of the international joint conference on artificial intelligence* (pp. 431–436).

Blizzard Activision. (1998). StarCraft. Released: March 31, 1998. Interplay. Retrieved from https://starcraft.com/en-us/

Blizzard Entertainment. (2002). Warcraft III: Reign of chaos. July 3, 2002. Blizzard Entertainment. Retrieved from http://www.blizzard.com/war3

Bulitko, V. [Vadim]. (2016a). Evolving real-time heuristic search algorithms. In *Proceedings of the synthesis and simulation of living systems*.

Bulitko, V. [Vadim]. (2016b). Per-map algorithm selection in real-time heuristic search. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment* (pp. 143–148).

Bulitko, V. [Vadim]. (2016c). Searching for real-time search algorithms. In *Proceedings of the international symposium on combinatorial search*.

Bulitko, V. [Vadim], & Lee, G. (2006). Learning in real time search: A unifying framework. *Journal of Artificial Intelligence Research*, *25*, 119–157.

Bulitko, V. [Vadim], & Sampley, A. (2016). Weighted lateral learning in real-time heuristic search. In *Proceedings of the symposium on combinatorial search*.

Cohen, L., Uras, T., Kumar, T., Xu, H., Ayanian, N., & Koenig, S. (2016). Improved solvers for bounded-suboptimal multi-agent path finding. In *Proceedings of the international joint conference on artificial intelligence* (pp. 3067–3074).

Cserna, B., Bogochow, M., Chambers, S., Tremblay, M., Katt, S., & Ruml, W. (2016). Anytime versus real-time heuristic search for on-line planning. In *Ninth annual symposium on combinatorial search*.

de Wilde, B., ter Mors, A., & Witteveen, C. (2013). Push and Rotate: Cooperative multi-agent path planning. In *Proceedings of the international conference on autonomous agents and multiagent systems* (pp. 87–94).

Epic Games. (2017). Fortnite. Retrieved June 28, 2018, from https://www.epicgames.com/fortnite/

Epic Games. (2018). Unreal 4 games wiki. Retrieved June 22, 2018, from https://wiki.unrealengine.com/Category:Games

Facebook. (2017). Pytorch. GitHub. Retrieved June 28, 2018, from https://github.com/pytorch/vision/blob/master/torchvision/models/alexnet.py

Google. (2017). Tensorflow. GitHub. Retrieved June 28, 2018, from https://github.com/tensorflow/models/blob/master/research/slim/nets/alexnet.py

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Hernandez, C., & Baier, J. (2012). Avoiding and escaping depressions in real-time heuristic search. In *Journal of artificial intelligence research* (pp. 523–570). JAIR'43.

Hernández, C., & Baier, J. (2012). Avoiding and escaping depressions in real-time heuristic search. *Journal of Artificial Intelligence Research*, *43*, 523–570.

Hernandez, C., Botea, A., Baier, J., & Bulitko, V. (2017). Online bridged pruning for real-time search with arbitrary lookaheads. In *Proceedings of the international joint conference on artificial intelligence* (pp. 510–516).

Ishida, T. (1997). *Real-time search for learning autonomous agents*. Springer Science & Business Media.

Koenig, S. (2001). Agent-centered search. *AI Magazine*, *22*(4), 109.

Koenig, S., & Likhachev, M. (2006). Real-time adaptive A*. In *Proceedings of joint conference on autonomous agents and multiagent systems* (pp. 281–288). doi:10.1145/1160633.1160682

Koenig, S., & Ma, H. (2017). Ai buzzwords explained: Multi-agent path finding(mapf). *AI Matters*.

Koenig, S., & Sun, X. (2009). Comparing real-time and incremental heuristic search for real-time situated agents. *Journal of Autonomous Agents and Multi-Agent Systems*, *18*(3), 313–341.

Korf, R. (1990). Real-time heuristic search. *Artificial Intelligence*, *42*(2–3), 189–211.

Kotthoff, L. (2014). Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 48–60.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of advances in neural information processing systems* (pp. 1097–1105).

Lawrence, R., & Bulitko, V. [V.]. (2013). Database-driven real-time heuristic search in video-game pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, *5*(3), 227–241. doi:10.1109/TCIAIG.2012.2230632

Lelis, L., Franco, S., Abisrror, M., Barely, M., Zilles, S., & Holte, R. (2016). Heuristic subset selection in classical planning. (pp. 3185–3191).

Loreggia, A., Malitsky, Y., Samulowitz, H., & Saraswat, V. (2016). Deep learning for algorithm portfolios. In *Proceedings of the aaai conference on artificial intelligence* (pp. 1280–1286). Phoenix, Arizona.

Lu, H., & Scott, P. (2004). Groups of diverse problem solvers can outperform groups of high-ability problem solvers. *Proceedings of the National Academy of Sciences*, *101*(46), 16385–16389. doi:10.1073/pnas.0403723101. eprint: http://www.pnas.org/content/101/46/16385.full.pdf

Malitsky, Y., Sabharwal, A., Samulowitz, H., & Sellmann, M. (2013). Algorithm portfolios based on cost-sensitive hierarchical clustering. In *Ijcai*.

Microsoft. (2017). Cntk. GitHub. Retrieved June 28, 2018, from https://github.com/Microsoft/CNTK/tree/master/Examples/Image/Classification/AlexNet

Rice, J. (1976). The algorithm selection problem. In *Proceedings of advances in computers,vol. 15* (pp. 65–118).

Rivera, N., Baier, J., & Hernández, C. (2013). Weighted real-time heuristic search. In *International conference on autonomous agents and multi-agent systems, AAMAS* (pp. 579–586). Retrieved from http://dl.acm.org/citation.cfm?id=2485012

Rivera, N., Baier, J., & Hernández, C. (2015). Incorporating weights into real-time heuristic search. *Artificial Intelligence*, *225*, 1–23. doi:10.1016/j.artint.2015.03.008

Samulowitz, H., & Memisevic, R. (2007). Learning to solve qbf. In *Proceedings of the 22nd national conference on artificial intelligence* (pp. 255–260). Vancouver, British Columbia, Canada.

Sharon, G. [G.], Stern, R., Felner, A., & Sturtevant, N. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, *219*, 40–66.

Sharon, G. [Guni], Sturtevant, N. R., & Felner, A. (2013). Online detection of dead states in real-time agent-centered search. In *Proceedings of the symposium on combinatorial search* (pp. 167–174). Retrieved from http://www.aaai.org/ocs/index.php/SOCS/SOCS13/paper/view/7226

Sigurdson, D., & Bulitko, V. [Vadim]. (2017). Deep learning for real-time heuristic search algorithm selection. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment* (pp. 108–114).

Sigurdson, D., Bulitko, V., Yeoh, W., Hernandez, C., & Koenig, S. (2018). Real-time multi-agent pathfinding. In *Proceedings of ieee conference on computational intelligence and games* (In press).

Silver, D. (2005). Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment* (pp. 117–122).

Square Enix. (2019a). Final fantasy vii remake. Retrieved June 28, 2018, from https://www.ffvii-remake.square-enix.com/

Square Enix. (2019b). Kingdom hearts iii. Retrieved June 28, 2018, from https://www.kingdomhearts.com/

Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, *4*(2), 144–148. Retrieved from http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

The Coalition. (2016). Gears of war 4. Retrieved June 28, 2018, from https://gearsofwar.com/en-au/games/gears-of-war-4

Vedaldi, A., & Lenc, K. (2015). MatConvNet – convolutional neural networks for MATLAB. In *Proceeding of the ACM international conference on multimedia* (pp. 689–692).

Wang, C., & Botea, A. (2011). MAPP: A scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research*, *42*, 55–90.

Wang, & Botea. (2008). Fast and memory-efficient multi-agent pathfinding. In *Proceedings of the international conference on automated planning and scheduling* (pp. 380–387).

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2008). Satzilla: Portfolio-based algorithm selection for sat. In *Journal of artificial intelligence research*.

Appendix A

Bounded Multi-agent A*

Reproduced from the proceedings of the IEEE conference on Computational Intelligence and Games 2018.

A.1 Preface

In our research developing an accessible approach to automatic algorithm selection we decided to create an algorithm that would pair well with two well known MAPF methods: Flow Annotated Replanning, and Windowed Hierarchical A*. The algorithm we created ended up preforming favourably with these two algorithms and as such motivated us to push the research to a publishable state. We reproduce it here instead of the body of thesis to keep the body focused on algorithm selection and not a specific algorithm.

A.2 Introduction

Pathfinding is a core task in many video games, for example, to allow *non-player characters* (NPCs) to move to given goal locations on a known stationary map. A* (Hart et al., 1968) is a classic algorithm for single-agent pathfinding. The artificial intelligence algorithms in video games, however, often need to find collision-free paths for several agents to their given goal locations. Figure A.1 illustrates *multi-agent pathfinding* (MAPF) (Koenig and Ma, 2017) on a map from the *Dragon Age: Origins* video game, where NPCs (green dots) have to move to their given goal locations (red dots).

The constraints on MAPF algorithms depend on the application. For example, real-time strategy games, such as *StarCraft* (Blizzard Activision, 1998), require the NPCs to move simultaneously in real time, which limits the amount of time available to compute the next moves for all NPCs before they need to start moving. Video games can generate maps procedurally to create new game levels on the fly, which makes it impossible to preprocess the maps. Players can often re-task NPCs at will or the map can change, rendering their previously calculated paths obsolete on a moment's notice. Finally, game settings can limit the amount of coordination allowed among characters in the game (such as sharing their paths or heuristic values), and some characters might not even be under the complete control of the system (because they are on an opposing team).

These constraints motivated our development of *Bounded Multi-Agent A** (BMAA*) — a MAPF algorithm that operates in real time, loses only a small amount of search in case players re-task NPCs or the map changes and neither requires explicit inter-agent coordination, complete control of all NPCs nor preprocessing of maps. BMAA* works as follows: Every agent treats the other agents as (moving) obstacles, runs an individual real-time heuristic search that searches the map around its current location within a given lookahead to select the next move and updates heuristic values assigned to locations to avoid getting stuck. We show how BMAA* can be enhanced by, first, adding flow annotations from the MAPF algorithm FAR (Wang and Botea, 2008) (that impose move directions similar to one-way streets) and, second, allowing agents to push other agents temporarily off their goal locations, when necessary, if agents are allowed to send each other move requests. In our experiments, BMAA* has higher completion rates and smaller completion times than FAR, thus demonstrating the promise of real-time heuristic search for MAPF.

A.3 Problem Formulation

A MAPF problem is defined by a pair (G, A). $G = (N, E, c)$ is an undirected weighted graph of nodes N connected via edges $E \subseteq N \times N$. The costs $c(e)$

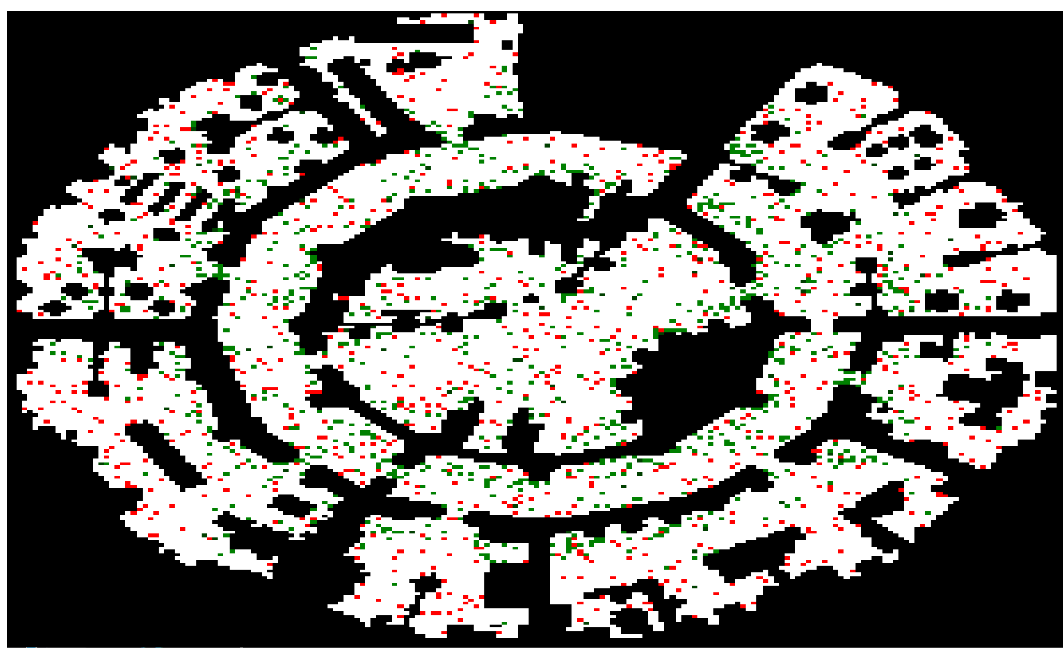

Figure A.1: NPCs on a *Dragon Age: Origins* map.

of all edges $e \in E$ are strictly positive with the following exceptions: There exists an edge for every node that connects the node to itself, allowing the agent to always wait in its current node. The costs of these edges are zero. $A = \{a^1, \ldots, a^n\}$ is a set of agents. Each agent $a^i \in A$ is specified by the pair $(n^i_{\text{start}}, n^i_{\text{goal}})$ of its start node n^i_{start} and goal node n^i_{goal}. We use graphs that correspond to rectangular 8-neighbor grids, as is common for video games. The nodes correspond to the cells not blocked by stationary obstacles. The nodes of two neighboring cells are connected with an edge. The costs of these edges are one for cardinal neighbors and $\sqrt{2}$ for diagonal neighbors.

Time advances in discrete steps. Every agent always occupies exactly one node at every time step. We use $n^i_{\text{curr}} \in N$ to refer to the current node of agent a^i. The agent determines a prefix P of a path from its current node to its goal node and sends it to a central NPC controller. $P(n)$ is the successor node of node n on the path. The central NPC controller then moves the agent from node n^i_{curr} to node $P(n^i_{\text{curr}})$ with the following exceptions: The agent waits in its current node if $P(n^i_{\text{curr}})$ is undefined or the agent would collide with another agent. Two agents collide iff they swap nodes or move to the same node from one time step to the next one.

We use the following performance measures: The *completion rate* is the percentage of agents that are in their goal locations when the runtime limit has been reached (Silver, 2005; Wang and Botea, 2008). The *completion time* for an agent is the time step when that agent last reached its goal location. If an agent leaves its goal and does not return the completion time is undefined. Finally, the *travel distance* of an agent is the sum of the costs of the edges traversed by that agent. We consider the mean of all agents' travel distance and the mean of all agents' completion time as the performance measures in our MAPF problems. These performance measures cannot be optimized simultaneously. Their desired trade-off can be game specific. We choose to maximize the completion rate (because players will notice if NPCs do not reach their goal locations) but report on the other two performance measures as well.

A.4 Related Work

We now review search algorithms that are related to BMAA*, focusing on pathfinding with heuristic search algorithms, which use heuristic values to focus their search.

A.4.1 A*

A* (Hart et al., 1968) provides the foundation for our BMAA* and many other MAPF algorithms, even though it was developed for single-agent pathfinding. An A* search for an agent explores the search space starting at its current node. The exploration is informed by heuristic values and driven toward nodes with a low estimate of the estimated cost of moving from the current node via them to the goal node. Algorithm 3 shows the pseudo-code for a version of A* that finds a cost-minimal path for agent a^i from its current node n^i_{curr} to its goal node n^i_{goal} under mild assumptions about the graph and the heuristic values.[1]

[1] In our pseudo-code, *First* returns a node with the smallest f-value in the open list (breaking ties in favor of a node with the largest g-value, with any remaining ties broken by first-in first-out); *Pop* removes a node with the smallest f-value from the open list (breaking ties in favor of a node with the largest g-value) and returns it; *Add* adds an element to a

It maintains two lists of nodes, namely the closed and open lists. The closed list is initially empty (line 2), and the open list contains the current node (line 3). The closed list is an unordered set of nodes that A* has already expanded. The open list is an ordered set of nodes that A* considers for expansion. A* always expands a node in the open list with the lowest f-value next, where the f-value of node n is $f(n) = g(n) + h(n)$. Its g-value $g(n)$ is the cost of the lowest-cost path from the current node to node n discovered so far, and its h-value $h(n)$ (or, synonymously, heuristic value) is the heuristic estimate of the cost of a lowest-cost path from node n to the goal node. (The g-values are initially zero for the start node and infinity for all other nodes.) A* removes node n from the open list and adds it to the closed list (lines 9 and 10). It then expands the node by iterating over all of its neighbors n'. It updates the g-value of node n' if node n' has not yet been expanded (i.e., it is not yet in the closed list) and the g-value of node n' can be decreased due to the fact that the cost of the path from the current node via node n to node n' is smaller than the g-value of node n' (because the search has then discovered a lower-cost path from the current node to node n') (line 17). In this case, it also updates the parent of node n' to node n (line 17) and adds it to the open list if it is not already in it (line 19). A* continues its search until either the open list is empty (line 5) or the node about to be expanded is the goal node (line 6). In the former case, no path exists from the current node to the goal node. In the latter case, the path P that is obtained by repeatedly following the parents from the node about to be expanded to the current node is a cost-minimal path from the current node to the goal node in reverse (line 7).

A.4.2 Online MAPF Algorithms

We focus on online MAPF algorithms, where there entire problem is not required to be solved before agents begin moving. since we are interested in MAPF algorithms that operate in a short amount of time, lose only a small amount of search in case players re-task NPCs or the map changes and neither require explicit inter-agent coordination, complete control of all NPCs

list; and *GetNeighbors* returns all neighboring nodes of a node in the graph.

Algorithm 3: A*.

```
1  P ← ()
2  closed ← ∅
3  open ← {n^i_curr}
4  g(n^i_curr) ← 0
5  while open ≠ ∅ do
6      if open.First() = n^i_goal then
7          calculate P
8          break
9      n ← open.Pop()
10     closed.Add(n)
11     for n′ ∈ n.GetNeighbors() do
12         if n′ ∉ closed then
13             if n′ ∉ open then
14                 g(n′) ← ∞
15             if g(n′) > g(n) + c(n, n′) then
16                 g(n′) ← g(n) + c(n, n′)
17                 n′.parent ← n
18                 if n′ ∉ open then
19                     open.Add(n′)
```

nor preprocessing of maps. We describe only the most suitable online MAPF algorithms below.

*Windowed Hierarchical Cooperative A** (WHCA*) (Silver, 2005) finds collision-free paths for all agents for their next window of moves. It shares the paths of all agents up to the given move limit through a reservation table, which adds a time dimension to the search space and thus results in expensive searches. Beyond the move limit, WHCA* simply assumes that every agent follows the cost-minimal path to its goal node and thus ignores collisions among agents. The move limit needs to be sufficiently large to avoid conflicts among agents, resulting in searches that might exceed the amount of time available to compute the next moves for all NPCs before they need to start moving. Furthermore, WHCA* requires all NPCs to be under its complete control.

Flow Annotated Replanning (FAR) (Wang and Botea, 2008) combines the reservation table from WHCA* with flow annotations that make its searches less expensive since no time dimension has to be added to the search space.

Each agent has to reserve its next moves before it executes them. Agents do not incorporate these reservations into their search but simply wait until other agents that block them have moved away, similar to waiting at traffic lights. FAR attempts to break deadlocks (where several agents wait on each other indefinitely) by pushing some agents temporarily off their goal nodes. However, agents can still get stuck in some cases. The flow annotations of FAR (Wang and Botea, 2008) change the edges of the original graph G in order to reduce the number of collisions among agents. They effectively make the undirected original graph directed by imposing move directions on the edges, similar to one-way streets, which reduces the potential for head-to-head collisions among agents. This annotation is done on a grid in a way so that any node remains reachable from all nodes from which it could be reached on the original graph, as follows: The new graph initially has no edges. The first row of nodes is connected via westbound edges, the second row is connected via eastbound edges, and so on. Similarly, the first column of nodes is connected via northbound edges, the second column is connected via southbound edges, and so on. Sink nodes (with only in-bound edges) and source nodes (with only out-bound edges) are handled by adding diagonal edges adjacent to them. If sink and source nodes are in close proximity of each other, the diagonal edges can end up pointing at each other and result in a loss of connectivity, in which case additional undirected edges are added around them. Corridor edges (that is, edges on paths whose interior nodes have degree two) of the original graph remain undirected, which is important in case the corridor is the only connection between two components of the original graph. A standard implementation of A* is then used to search for a path to the goal in this restricted graph.

A.4.3 Real-time Heuristic Search

Video games often require NPCs to start moving in a short amount of time, which may not be possible with any of the search algorithms reviewed above since they need to compute a complete path before an agent can execute the first move. *Real-time heuristic search* (RTHS) algorithms, on the other

hand, perform a constant amount of search per move regardless of the size of the map or the distance between the start and goal nodes. They have been studied for single-agent pathfinding (Vadim Bulitko and Lee, 2006; Cserna et al., 2016; Ishida, 1997; Koenig and Sun, 2009), starting with the seminal work by Korf (Korf, 1990). They need to compute only the prefix of a path before the agent can execute the first move — and repeat the operation until the agent reaches the goal node. To avoid cycling forever without reaching the goal node due to the incompleteness of the searches, the algorithms update the heuristic values over time by making them locally consistent (Korf, 1990), incrementally building the open and closed lists (Björnsson, Bulitko, and Sturtevant, 2009) or ignoring parts of the map (Hernandez, Botea, Baier, and Bulitko, 2017). There are two benefits to using RTHS algorithms in video games. First, an NPC can start moving in a short amount of time. Second, only a small amount of search is lost in case a player re-tasks NPCs or the map changes.

A well-known RTHS algorithm *Real-Time Adaptive A** (RTAA*) (Koenig and Likhachev, 2006) performs an A* search, limited to a given number of node expansions. RTAA* then uses the f-value of the node A* was about to expand to update the heuristic values of all expanded nodes (that is, all nodes in the closed list *closed*) as shown in Procedure *Update-Heuristic-Values* in Algorithm 6. The agent then moves along the path from its current node to the node A* was about to expand, limited to a given number of moves — and RTAA* repeats the operation.

A.5 Our Approach: BMAA*

Our *Bounded Multi-Agent A** (BMAA*) is a MAPF algorithm where every agent runs its own copy of RTAA*. BMAA* satisfies our requirements: It operates in real-time, loses only a small amount of search in case players re-task NPCs or the map changes. Additionally, it does not requires explicit inter-agent coordination, complete control of all NPCs or preprocessing of maps. The design of BMAA* is modular to allow for extensions by adding or

changing modules. For example, BMAA* can be enhanced by, first, adding flow annotations from FAR and, second, allowing agents to push other agents temporarily off their goal nodes, when necessary, if agents are allowed to send each other move requests.

We parameterize BMAA* as follows in the spirit of recent research in the context of *Parameterized Learning Real-Time A** (Vadim Bulitko, 2016a): First, *expansions* is the limit on the number of node expansions of the A* search of RTAA*. Second, *vision* is the distance within which agents can see other agents. Third, *moves* is the number of moves that each agent makes along its path before RTAA* determines a new path for the agent. Fourth, *push* is a Boolean flag that determines whether agents can push other agents temporarily off their goal nodes. Finally, *flow* is a Boolean flag that determines whether RTAA* uses the flow annotations from FAR.

A.5.1 Procedure NPC-Controller

Algorithm 4 shows the pseudo-code for the central NPC controller. The time step *time* is initialized to zero at the start of BMAA*, and the central NPC controller is then invoked at every time step with A, the set of agents currently under the control of the system. In the search phase, the central NPC controller lets every agent under the control of the system use the Procedure *Search-Phase* shown in Algorithm 5 to find a prefix of a path from its current node to its goal node (line 2, Algorithm 4). In the subsequent execution phase, the central NPC controller iterates through all agents under the control of the system: First, it retrieves the node that the agent should move to next, which is the successor node of the current node of the agent on its path (line 5, Algorithm 4). Second, if the desired node is blocked by an agent that has reached its own goal node already and agents can push other agents temporarily off their goal nodes ($push = true$), it can push the blocking agent to any neighboring node (line 7, Algorithm 4). The blocking agent returns to its own goal node in subsequent time steps since all agents always execute RTAA* even if they are in their goal nodes. Finally, it moves the agent to the desired node if that node is (no longer) blocked (line 9, Algorithm 4) and increments the

```
Algorithm 4: BMAA*'s NPC Controller.
input  : A
1  forall a^i ∈ A do
2      a^i.Search-Phase()
3  forall a^i ∈ A do
4      if a^i.P(n^i_curr) is defined then
5          n ← a^i.P(n^i_curr)
6          if push ∧ n is blocked by agenta^j then
7              a^j.PushAgent()
8          if nis not blocked by an agent then
9              a^i.MoveTo(n)
10 time ← time + 1
```

current time step (line 10, Algorithm 4).

```
Algorithm 5: Search-Phase
1  if Search.P(n^i_curr) is undefined or time ≥ limit then
2      Search()
3      if Search.open ≠ ∅ then
4          n ← Search.open.First()
5          f ← g(n) + h(n)
6          Update-Heuristic-Values(Search.closed, f)
7          limit ← time + moves
```

```
Algorithm 6: Update-Heuristic-Values
input  : closed, f
1  for n ∈ closed do
2      h(n) ← f − g(n)
```

A.5.2 Procedure Search-Phase

Algorithm 5 shows the pseudo-code for the search phase. It finds a new prefix of a path from the current node of the agent to its goal node when it has reached the end of the current path, the current node is no longer on the path (for example, because the agent has been pushed away from its goal node), or the agent has already executed *moves* moves along the path. (The "expiration" time step *limit* for the path keeps track of the last condition on line 1 and is

Algorithm 7: Search

```
1  P ← ()
2  exp ← 0
3  closed ← ∅
4  open ← {n^i_curr}
5  g(n^i_curr) ← 0
6  while open ≠ ∅ do
7      if open.First() = n^i_goal ∨ exp ≥ expansions then
8          calculate P
9          break
10     n ← open.Pop()
11     closed.Add(n)
12     for n′ ∈ n.GetNeighbors(flow) do
13         d ← distance(n^i_curr, n′)
14         if n′ is blocked by an agent ∧d ≤ vision then
15             if n′ ≠ n^i_goal then
16                 continue
17         if n′ ∉ closed then
18             if n′ ∉ open then
19                 g(n′) ← ∞
20             if g(n′) > g(n) + c(n, n′) then
21                 g(n′) ← g(n) + c(n, n′)
22                 n′.parent ← n
23                 if n′ ∉ open then
24                     open.Add(n′)
25     exp ← exp + 1
```

set on line 7.) If so, then it uses Procedure *Search* in Algorithm 7 to execute an RTAA* search (line 2) and uses Procedure *Update-Heuristic-Values* to update the heuristic values afterward (lines 4-6).

A.5.3 Procedure Search

Algorithm 7 shows the pseudo-code for an A* search, as discussed before, but with the following changes: First, each agent maintains its own heuristic values across all of its searches. Second, the search also terminates after it has expanded *expansions* nodes. Thus, the path P obtained on line 8 by repeatedly following the parents from the node about to be expanded to the current node is now only the prefix of a path from the current node of the

agent to its goal node. Finally, *GetNeighbors* returns a node's neighbours that are not blocked by stationary obstacles. However, other agents within the straight-line distance *vision* within which agents can see other agents are treated as obstacles as long as they do not block its goal node. Thus, the corresponding nodes are immediately discarded (lines 14-16). If RTAA* uses the flow annotations from FAR (*flow* = *true*), then *GetNeighbors* returns only those neighboring nodes of a node of the graph which are reachable from the node via the flow annotations from FAR. The flow annotations are not computed in advance but generated the first time the node is processed and then cached so that they can be re-used later.

A.6 Experimental Evaluation

We experimentally evaluate four versions of BMAA* both against FAR and against A*-Replan, which is equivalent to FAR with no flow annotations. BMAA* cannot push other agents temporarily off their goal locations (*push* = *false*) and uses no flow annotations (*flow* = *false*), BMAA*-c can push other agents temporarily off their goal locations, BMAA*-f uses flow annotations, and BMAA*-f-c combines both features. All BMAA* versions use the parameters *lookahead* = *32*, *moves* = *32* and *vision* = $\sqrt{2}$. We choose these parameters on the basis of preliminary experiments. Increasing *lookahead* often decreases the travel distance at the cost of increasing the search time per move. Increasing *vision* often reduces the completion rate since it makes agents react to far-away agents. FAR and A*-Replan use a reservation size of three, as suggested by the creators of FAR, meaning that agents must successfully reserve their next three moves before they execute them. All MAPF algorithms use the octile heuristic values as heuristic values (or, in case of BMAA*, to initialize them), are coded in C# and are run on a single Intel Broadwell 2.1Ghz CPU core with 3GB of RAM and a runtime limit of 30 seconds per MAPF instance, which is sufficiently large to allow for full A* searches.

We evaluate them on ten maps from the MovingAI benchmark set (Sturtevant, 2012). We use three maps from *D*ragon Age: Origins (DAO), three maps

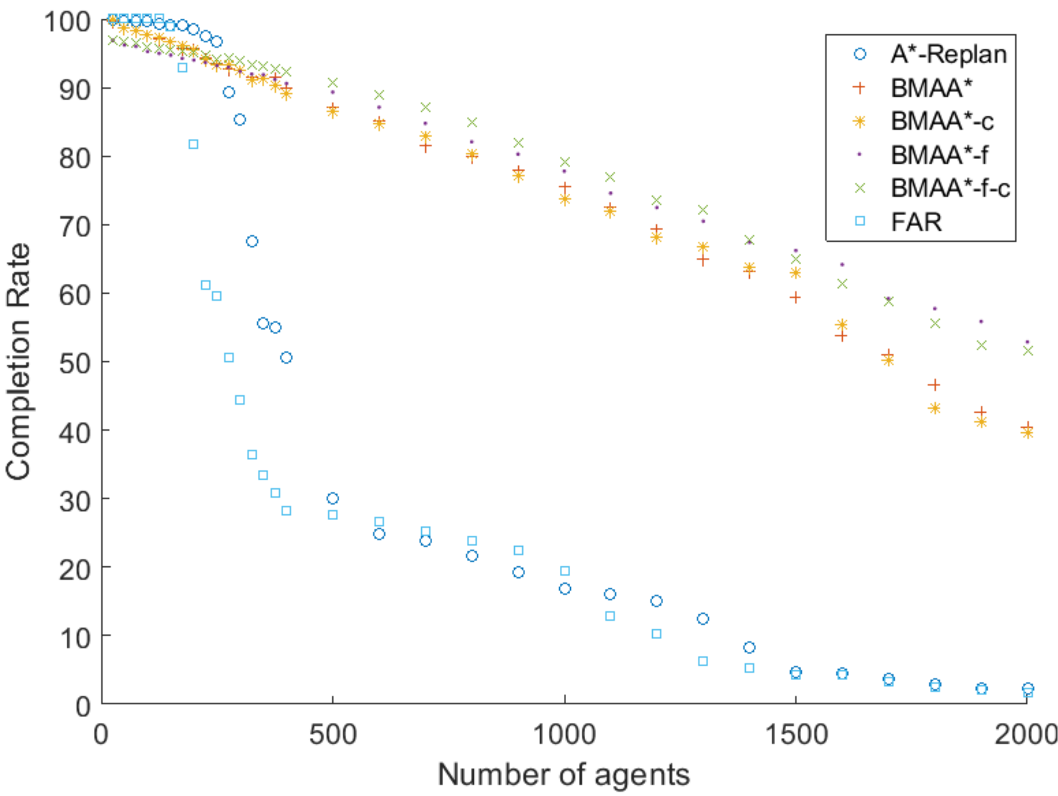

Figure A.2: Completion rates averaged over all MAPF instances.

from *W*arCraft III (WCIII), three maps from *B*aldur's Gate II (BGII) (resized to 512 × 512) and one map from *B*aldur's Gate II in its original size. We create ten MAPF instances for each map with the number of agents ranging from 25 to 400 in increments of 25 and from 400 to 2000 in increments of 200. We assign each agent unique randomly selected start and goal locations which are reachable from each other in the absence of other agents.

A.6.1 Aggregate Completion Rate Results

Figure A.2 shows the completion rates of all MAPF algorithms averaged over all MAPF instances on all maps. The completion rates of all MAPF algorithms decrease as the number of agents increases because the congestion and amount of search (since every agent has to search) increase. A higher congestion makes it more difficult for agents to reach their goal locations, and a higher amount of search makes it more likely that the runtime limit is reached.

All BMAA* versions have substantially higher completion rates than FAR

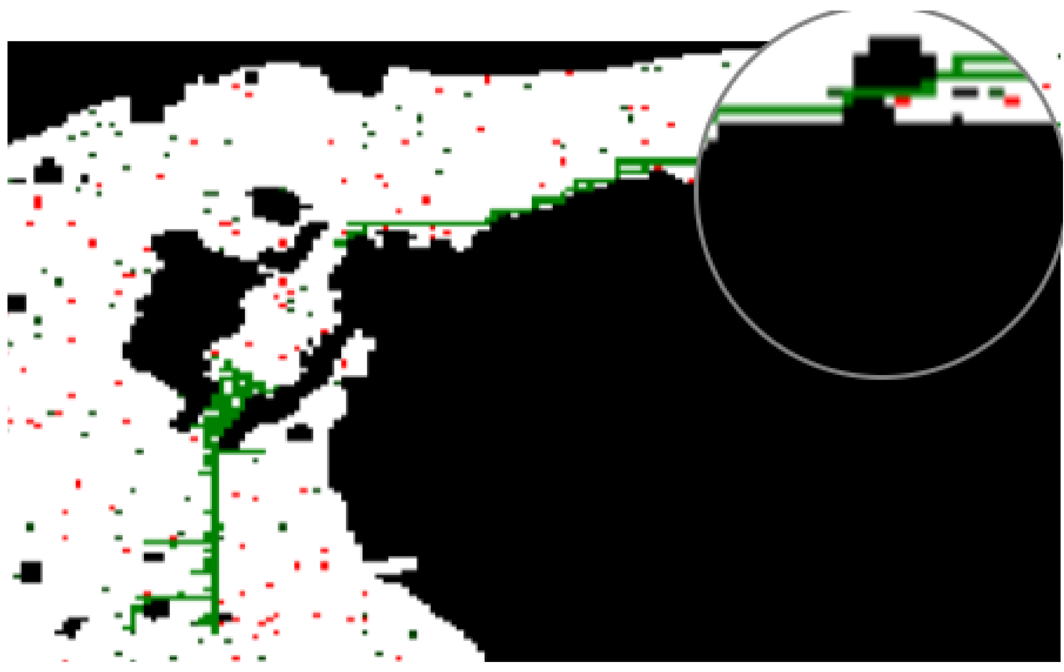

Figure A.3: Issue for FAR: One-cell-wide corridors.

and A*-Replan for more than 200 agents, with the BMAA* versions that can push other agents temporarily off their goal locations being slightly better than the other BMAA* versions. This can be explained as follows: FAR and A*-Replan determine complete paths for the agents, which results in many agents sharing path segments and thus creates congestion around choke points, such as the one-cell-wide corridor in Figure A.3. The BMAA* versions often avoid this behavior, for two reasons: First, the agents of the BMAA* versions have larger travel distances than the ones of FAR and A*-Replan. While the large travel distances of RTHS algorithms are viewed as a major deficiency in the context of single-agent pathfinding, they are beneficial in the context of MAPF since they avoid congestion around choke points. Second, the agents of the BMAA* versions treat the other agents as (moving) obstacles and thus find paths that avoid choke points that are blocked by other agents and thus appear impassable, while the agents of FAR and A*-Replan assume that they can resolve conflicts in their paths with those of other agents and thus move toward choke points.

However, the BMAA* versions also have disadvantages: First, they might

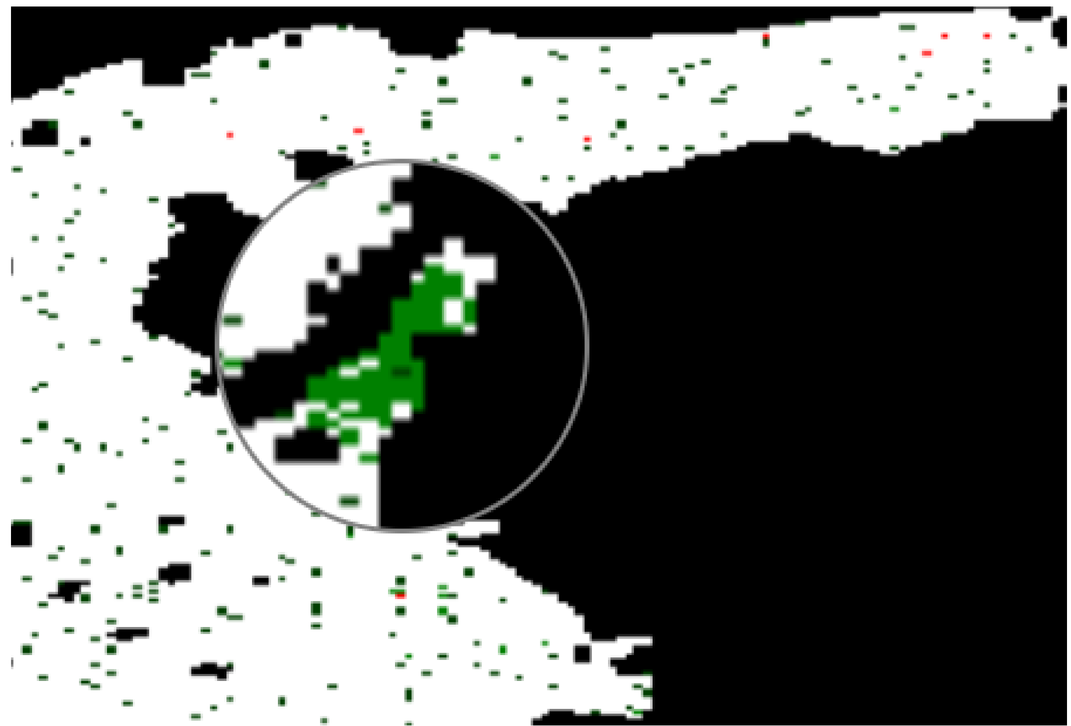

Figure A.4: Issue for BMAA*: Dead ends.

move agents into dead ends, such as the one shown in Figure A.4, if the initial heuristic values are misleading (resulting in depressions in the heuristic value surface). This well-known issue for RTHS algorithms is addressed by them updating their heuristic values. Several recent RTHS techniques attempt to reduce the travel distances but, of course, agents exploring new areas in imperfect manners could also be viewed as realistic in some cases. Second, even the BMAA* versions that can push other agents temporarily off their goal locations might not be able to move all agents to their goal locations when other agents on their paths are unable to vacate their own goal locations (Figure A.5).

A.6.2 Per-Map Results

Tables A.1-A.3 show the three performance measures for all MAPF algorithms averaged over all MAPF instances for each of the maps separately since the map features affect the performance of the MAPF algorithms. The best results are highlighted in bold.

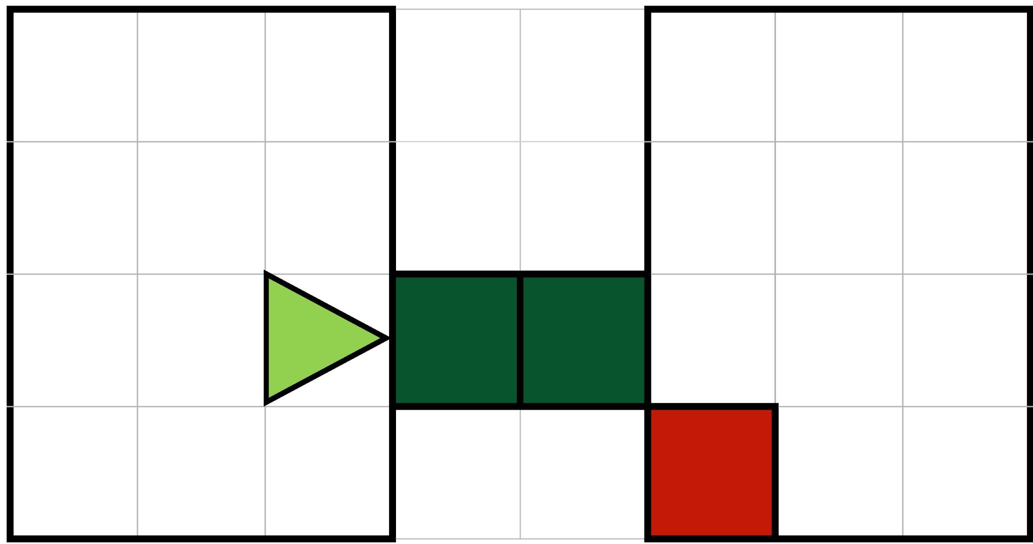

Figure A.5: Unsolvable MAPF instance for the BMAA* versions, where the triangular agent has to move to its red goal location while the dark green square agents are already at their own goal locations in a one-cell-wide corridor.

Per-Map Completion Rate Results

Table A.1 shows that BMAA*-f-c has the highest completion rates on seven out of the ten maps but the completion rate of BMAA*, for example, is 15 percent larger than the one of BMAA*-f-c on map DAO-lak307d.

Table A.1: Completion rates averaged over all MAPF instances for each map.

Map Name	A*-Replan	BMAA*	BMAA*-c	BMAA*-f	BMAA*-f-c	FAR	Overall
BGII-AR0414SR (320*281)	45	87	87	85	**89**	32	71
BGII-AR0414SR (512*512)	14	80	79	82	**83**	07	58
BGII-AR0504SR (512*512)	08	51	51	**62**	**62**	05	40
BGII-AR0701SR (512*512)	08	48	49	64	**65**	06	40
WCIII-blastedlands (512*512)	14	**85**	**85**	78	80	03	58
WCIII-duskwood (512*512)	08	58	58	**67**	**67**	03	43
WCIII-golemsinthemist (512*512)	10	59	59	**72**	**72**	04	46
DAO-lak304d (193*193)	19	39	38	**53**	51	27	38
DAO-lak307d (84*84)	60	**79**	77	68	64	60	68
DAO-lgt300d (747*531)	12	65	65	**77**	**77**	10	51
Overall	20	65	65	71	71	16	51

Per-Map Completion Time Results

The completion rates of FAR and A*-Replan drop substantially for more than 200 agents, as shown in Figure A.2. We thus limit the number of agents to

200 since most agents then reach their goal locations. We assign the remaining agents a completion time of 30 seconds. Table A.2 shows that BMAA*-f has the lowest completion times on five maps and BMAA* has the lowest completion times on the remaining four maps.

Table A.2: Completion times (in seconds) averaged over all MAPF instances for each map.

Map Name	A*-Replan	BMAA*	BMAA*-c	BMAA*-f	BMAA*-f-c	FAR	Overall
BGII-AR0414SR (320*281)	2.8	**1.2**	5.1	2.2	5.6	3.8	3.5
BGII-AR0414SR (512*512)	8.8	3.6	6.6	**3.0**	6.8	12.9	7.0
BGII-AR0504SR (512*512)	12.3	8.6	12.7	**6.3**	12.5	16.0	11.4
BGII-AR0701SR (512*512)	12.7	4.0	5.4	**3.2**	4.5	15.0	7.5
WCIII-blastedlands (512*512)	8.8	**1.4**	1.5	2.2	2.3	21.0	6.2
WCIII-duskwood (512*512)	12.5	4.1	5.8	**3.7**	5.5	21.1	8.8
WCIII-golemsinthemist (512*512)	11.1	4.2	5.9	**3.0**	4.2	19.0	7.9
DAO-lak304d (193*193)	4.5	6.7	15.1	7.9	11.4	**3.2**	8.1
DAO-lak307d (84*84)	**0.2**	**0.2**	**0.2**	0.5	0.3	0.6	0.3
DAO-lgt300d (747*531)	8.3	**.4**	1.6	2.2	2.4	10.5	4.4
Overall	8.2	3.5	6.0	**3.4**	5.5	12.3	6.5

Per-Map Travel Distance Results

We again limit the number of agents to 200 since most agents then reach their goal locations. We assign the remaining agents their travel distances when the runtime limit is reached. Table A.3 shows that FAR has the lowest travel distances on nine maps.

Table A.3: Travel distances averaged over all MAPF instances for each map.

Map Name	A*-Replan	BMAA*	BMAA*-c	BMAA*-f	BMAA*-f-c	FAR	Overall
BGII-AR0414SR (320*281)	663	554	557	620	639	**130**	527
BGII-AR0414SR (512*512)	661	1538	1557	2080	2115	**224**	1363
BGII-AR0504SR (512*512)	407	2167	2231	3671	3783	**227**	2089
BGII-AR0701SR (512*512)	562	973	967	1267	1287	**322**	896
WCIII-blastedlands (512*512)	299	376	376	775	784	**268**	480
WCIII-duskwood (512*512)	367	1179	1188	1712	1737	**257**	1073
WCIII-golemsinthemist (512*512)	530	1205	1206	1371	1369	**285**	994
DAO-lak304d (193*193)	2154	1425	1460	1258	1295	**148**	1290
DAO-lak307d (84*84)	578	**38**	39	125	95	47	154
DAO-lgt300d (747*531)	435	403	404	592	603	**289**	454
Overall	666	986	998	1347	1371	**225**	932

A.7 Conclusions

Our paper considered an important problem faced by artificial intelligence in many video games, namely MAPF. We reviewed recent related work and argued for the use of real-time heuristic search. We then contributed a new real-time MAPF algorithm, BMAA*, which is of modular design and can be enhanced with recent flow-annotation techniques. BMAA* has higher completion rates and smaller completion times than FAR at the cost of longer travel distances, which is a good trade-off since NPCs reaching their goal locations via possibly longer paths is less noticeable by players than NPCs not reaching their goal locations at all. Finally, we discussed what makes MAPF difficult for different algorithms, paving the road to per-problem algorithm selection techniques in the spirit of recent research in the context of single-agent pathfinding (Vadim Bulitko, 2016b; Sigurdson and Bulitko, 2017).

Overall, BMAA* demonstrates the promise of real-time heuristic search for MAPF. Its main shortcoming is its large travel distances compared to the ones of FAR. Several recent RTHS techniques attempt to reduce the travel distances for single-agent pathfinding (Vadim Bulitko and Sampley, 2016) and thus might also be able to reduce the travel distances for BMAA*. Examples include search space reduction techniques (Hernández and Baier, 2012; Hernandez et al., 2017), precomputation techniques (Cohen et al., 2016; Lawrence and Bulitko, 2013) and initialization techniques for the heuristic values, which might help to reduce the dead-end problem shown in Figure A.4.