Proceedings, The Thirteenth AAAI Conference on
Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)

# Deep Learning for Real-Time Heuristic Search Algorithm Selection

**Devon Sigurdson, Vadim Bulitko**
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
{dbsigurd | bulitko}@ualberta.ca

**Abstract**

Real-time heuristic search algorithms are used for creating agents that rely on local information and move in a bounded amount of time making them an excellent candidate for video games as planning time can be controlled. Path finding on video game maps has become the de facto standard for evaluating real-time heuristic search algorithms. Over the years researchers have worked to identify areas where these algorithms perform poorly in an attempt to mitigate their weaknesses. Recent work illustrates the benefits of tailoring algorithms for a given problem as performance is heavily dependent on the search space. In order to determine which algorithm to select for solving the search problems on a map the developer would have to run all the algorithms in consideration to obtain the correct choice. Our work extends the previous algorithm selection approach to use a deep learning classifier to select the algorithm to use on new maps without having to evaluate the algorithms on the map. To do so we select a portfolio of algorithms and train a classifier to predict which portfolio member to use on a unseen new map. Our empirical results show that selecting algorithms dynamically can outperform the single best algorithm from the portfolio on new maps, as well provide the lower bound for potential improvements to motivate further work on this approach.

## 1 Introduction and Related Work

Autonomous agent search is a key task in video games. An agent is tasked with finding a path from their starting location to a goal. The agent is 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 immediate neighbors.

Real-time heuristic search algorithms bound the time the agent can use for planning each move. The bounding is done such that the amount of time for each move is independent of the search graph it operates in. By contrast, the time for an A* agent to make its first move is directly tied to the number of states the agent must expand before reaching the goal. Limiting the search to local information of the surrounding area centers the search around the agent and prevents the jumping to desired frontier nodes as seen in A*.

Real-time agents interweave their planning and action as they traverse to the goal. This interweaving of planning and action is done through limiting the agent to information local to it and bounding the amount of time to plan its move. Agents will move based on limited information and before the whole path to the goal is known causing the agent to make suboptimal decisions. During the agent's traversal to the goal it updates its heuristic of each state it visits. These updates enable the agent to find the goal as their heuristics converge to the correct value but makes the agents prone to frequent state revisitation, often referred to as scrubbing. The scrubbing behavior appears irrational or broken as the agent will visit the same state repeatedly in short succession (Hernández and Baier 2012).

In an effort to mitigate scrubbing, agents can attempt to identify when they enter a heuristic depression. A depression is a bounded region in which the contained states have heuristics that are too low in comparison to the surrounding states. Because of these depressed regions, the agent is forced to scrub back and forth until the heuristics rise to the point where the region is no longer depressed (Ishida 1992). Knowing the agent has entered a depressed region they can employ depression avoidance techniques to escape the depression by marking the states in the region with a temporary avoid flag to discourage the agent from exploring the region (Hernández and Baier 2012). Another method for reducing scrubbing is through pruning expendable states, which are states whose neighbors are all reachable without said state (Sharon, Sturtevant, and Felner 2013).

LRTA* originally updated the heuristic value of a state based on the surrounding state with the minimum heuristic estimate to the goal. Different heuristic learning rules have also been researched for alternatives to the original methods used in LRTA* (Bulitko 2016d; 2016a). Weighting the learning of the agent has been explored and shown to help speed up converging to the correct heuristic value (Shimbo and Ishida 2003; Rivera, Baier, and Hernndez 2015).

Over the two decades of research since the seminal LRTA* many new algorithms have been proposed and stud-

Copyright © 2017, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

ied. Despite being developed as standalone algorithms these variations were able to be unified into a single framework (Bulitko 2016a). This framework parameterized which components of the previous algorithms were active, enabling different algorithms to be achieved by varying the parameters of the framework. A downside to having parametrized algorithms is the bewildering combination of the various algorithms and parameters that can be created. These vast combinations can pose a challenge for their use in practice. To overcome this, an automated algorithm selection technique was implemented by Bulitko (2016a) where several variations of uLRTA* were evaluated to determine which combination of parameters was the best single solver overall in their video game map test bed and which algorithm was best on a per-map basis (Bulitko 2016b). Selecting an algorithm on a per-map bases resulted in significant improvements over using the single best solver on all maps (Bulitko 2016b); Bulitko's approach however, required evaluating the potential algorithms on each new domain in order to select which algorithm to use.

This approach of developing specialized solvers that excel at a specific problem achieved state-of-the-art results not only in real-time-heuristic search but in other fields such as satisfiability problems (Kotthoff 2014; Loreggia et al. 2016). The underlying concept behind the success of these approaches is the algorithm selection problem (Rice 1976). Selecting the correct algorithm or configuration of an algorithm from a portfolio for a given problem can be automated using tools such as Sequential Model-Based Optimization for General Algorithm Configuration (SMAC) and sampling approaches illustrated in per-maps selection.

These algorithm selection techniques operate in an off-line matter where in order to select an algorithm on a new domain the portfolio must be evaluated on the new domain. This requires the new domain to be known sufficiently ahead of time for it to be used in practice where in games this may not always be conceivable. To alleviate this off-line drawback deep learning image networks have been trained to take a visual representation of a satisfiability problem to select which algorithm from a portfolio is the best solver for that problem (Loreggia et al. 2016). This approach was able to achieve better results than using the single best solver from the training data in the satisfiability study. Given the visual aspect of video game maps, using the same approach should be a valid avenue for alleviating the off-line requirement that currently exists for real-time heuristic search algorithms.

Dynamic algorithm selection is well suited for video games given the rise of procedurally generated maps and widely available game engines like Unreal Engine and Unity. In both cases the game maps that potential agents must traverse is not fixed and may not have sufficient time to sample the algorithms in the portfolio on the new game maps. Being able to dynamically select algorithms could lead to performance gain over using the best single solver on the previous maps.

The contribution of this paper is demonstrating a method for using an off-the-shelf deep learning neural network to select an algorithm for a given problem well enough to outperform the use of a single algorithm. We organize the rest of the paper as follows. First we formalize the definition for a real-time heuristic search in Section 2. We then explain the search algorithm framework for which we form the portfolio of algorithms from in Section 3. We then explain our methodology for dynamically selecting algorithms for a given problem in Section 4 and discuss the varying results for map and problem based selection in Section 6 and 7. Limitations of our approach and future work directions are presented in Section 8 followed by conclusions.

## 2 Problem Formulation

We re-use the problem formulation of Bulitko (2016c) and reproduce it below for the reader's convenience. A *search problem* is a tuple ($S$, $E$, $c$,$s_0$, $s_g$, $h$) where $S$ is a finite set of $states$ and $E \in S \times S$ is a set of $edges$ between them. $S$ and $E$ jointly define the search graph. The search graph is stationary, undirected and connected (and thus safely explorable). No state has an edge leading to itself. Each edge ($s_a$,$s_b$) $\in E$ is weighted by the $cost$ $c(s_a, s_b) = c(s_b, s_a) > 0$. The agent begins in the start state $s_0$ and changes its current state by traversing edges (i.e., taking actions). The cumulative cost of all edges it traverses prior to reaching the goal state $s_g$ is the $solution\ cost$. The $suboptimality$ ($\alpha$) is the ratio of the solution cost produced by an agent to the cost of the shortest possible path, $h^*(s_0)$. Lower values are desired; the value of 1 indicates optimality.

The agent has access to a heuristic $h$ which is an estimate of the remaining cost to the 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 $h(s_g) = 0$. The heuristic at time $t$ is denoted by $h_t$; the initial heuristic $h_0 = h$ is included in the problem description.

The objective is to find a real-time heuristic search algorithm with the lowest expected suboptimality. The expectation is empirically approximated by $sample\ suboptimality$: running an algorithm on a set of benchmark problems and averaging suboptimality of the solutions. Per-problem sub-optimality is capped as $\alpha_{max} \geq 1$.

## 3 Search Framework

Section 1 introduced some of the ways of improving LRTA* towards lower solution suboptimality. In this paper we specifically focus on how to select which parametrized real-time heuristic search algorithm (Algorithm 1) (Bulitko 2016b) to use based on the search space. To isolate the problem, we fix the lookahead at 1 (i.e., allow the agent to consider only the immediate neighbors of its current state during the planning stage). We use the same parametrized LRTS as Bulitko (2016a) and Bulitko (2016b) which takes the following input parameters $w$, $w_c$, $b$, lop, `da`, `expendable`.

The agent will deploy depression avoidance (Hernández and Baier 2012) techniques if the parameter `da` $= true$, as shown in line 4. If used then line 5 will have the agent's neighboring states $N(s_t)$ temporarily set to include states which have minimal amounts of learning($|h_t(s) - h_0(s)|$) (Bulitko 2016a). This is done to prevent frequent state revisitation by discouraging agents from revisiting the same states right away.

Line 6 consists of the learning rule which utilizes weighted heuristics (Bulitko 2016c; Rivera, Baier, and Hernández 2013) and lateral learning (Bulitko 2016a). These parameters are controlled with $w$, $w_c$, lop, and $b$. The learning operator is represented by lop which consists of min, max, median, and 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 state to the neighboring states. The lateral learning portion of line 6 is defined by the agent's neighborhood $N_b^f$ as the $b$ fraction of the neighborhood $N(s_t)$ with minimum $f$ values:

$$N_b^f(s) = (s^1, ..., s^{\lfloor b|N(s_t)| \rfloor})$$

where $(s^1, ..., s^{\lfloor b|N(s_t)| \rfloor}, ..., s^{|N(s_t)|})$ is the immediate neighborhood sorted in the ascending order by their $f$ values. A $b$ value of 1 represents the full neighborhood(all the connecting states), while a value of 0 represents the single neighboring state with lowest $f$ value.

When the control parameter `expendable` is active and a state is deemed expendable (Sharon, Sturtevant, and Felner 2013) then it is pruned from the graph, as shown in line 8. In order for a state to be expendable (denoted by $\varepsilon(s_t)$ ) all its immediate neighbors must be reached from each other within the immediate neighborhood; learning must have also occurred in line 6. The agent moves to its new state in line 9.

**Algorithm 1:** Parametrized Real-time Heuristic Search

**input** : search problem $(S, E, c, s_0, s_g, h)$, control parameters $w, w_c, b,$ lop, da, expendable
**output**: path $(s_0, s_1, \ldots, s_T), s_T = s_g$

1 $t \leftarrow 0$
2 $h_t \leftarrow h$
3 **while** $s_t \neq s_g$ **do**
4  **if** da **then**
5  $N(s_t) \leftarrow N_{minlearning}(s_t)$
6  $h_{t+1}(s_t) \leftarrow$ $\max\left\{h_t(s_t), w \cdot lop_{s \in N_b^f(s_t)}(w_c \cdot c(s_t, s) + h_t(s))\right\}$
7  **if** expendable & $h_{t+1}(s_t) > h_t(s_t)$ & $\varepsilon(s_t)$ **then**
8  remove $s_t$ from the search graph;
9  $s_{t+1} \leftarrow \arg\min_{s \in N(s_t)}(c(s_t, s) + h_t(s))$
10  $t \leftarrow t + 1$
11 $T \leftarrow t$

### 3.1 Space of Algorithms

We use the same control parameters as Bulitko (2016b) 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 in which our method will choose from.

## 4 Our Approach

The best algorithm on a search problem for algorithm selection can be selected through evaluating all the algorithms in the portfolio. However, this is computationally expensive and defeats the purpose if to solve one problem faster several slower algorithms are ran to determine the best choice. Our approach leverages the advances in image classification to select which algorithm to use for a given search problem. Instead of running the algorithms in the portfolio on these new maps the game designer runs them through a trained image classifier that selects the algorithm to use for the input problems. Modern classifiers have very high accuracy at making correct classifications for the examples that trained the network. This enables achieving close to the best suboptimality for the portfolio for all the training examples.

As the previous work showed, evolving an algorithm (Bulitko 2016a) to perform well overall can be outperformed by explicitly selecting an algorithm on a per-map basis (Bulitko 2016b). This concept of selecting a specific algorithm for a given problem has been studied extensively. Kotthoff (2014) recently provided a comprehensive survey on algorithm selection for combinatorial search. Our work takes this concept and provides a general technique for selecting algorithms on new problems without sampling algorithms in order to make the selection as Bulitko (2016b) did. Our technique is generalizable to any chosen performance metric but for this study we focus on $suboptimality$.

### 4.1 Algorithm Generation

In order to dynamically select algorithms we first need to create algorithms to form a portfolio. We take a simple approach to algorithm generation by creating 1000 algorithms from the algorithm space defined in Section 3.1. Each control parameter was selected uniform randomly in the same fashion as Bulitko (2016b) did. While the space of algorithms that we chose from are variations of parameterized uLRTA*, seen in Figure 1, conceptually these can be thought of as separate algorithms and many of the combinations were originally developed as standalone algorithms.

### 4.2 Portfolio Formation

The large performance gains seen in state-of-the-art algorithms come from specialized algorithms that excel at a very specific set of problems through sacrificing generality (Bulitko 2016b). Since our algorithm selection is done in an on-line fashion selecting one of these specialized algorithms outside of their specialty can reduce the performance of the algorithm selection approach.

Algorithm portfolios define the algorithms which are able to be selected. Ensuring any selection from a portfolio has a specific property can be achieved by having all algorithms in the portfolio each individually having the desired property. This can be used to ensure no matter what selection is made desired results such as completeness or upper bounds on suboptimality are met.

To reduce the potential harm from selecting an incorrect algorithm we reduce the portfolio size from the possible 1000 different variations to a smaller subset. We form

our subset of algorithms using a greedy portfolio generation shown in Algorithm 2. A similar greedy subsetting approach was applied to selecting a subset of heuristic functions for the traditional A* algorithm in order to reduce the computation time required for evaluating these heuristic functions through considering a smaller subset (Lelis et al. 2016).

Algorithm 2 forms a portfolio by greedily adding the algorithm which maximizes the portfolios performance under perfect selection. The algorithm takes a set of algorithms $A$ with a target portfolio size $n$ indicating the maximum number of members in the portfolio. The subsetted portfolio of algorithms is represented by the set $T$ and initialized to a null set in line 1. Line 3 finds algorithm $a$ from the set of algorithms $A$ which has the lowest suboptimality in combination with portfolio $T$ defined by $f$. The best algorithm is then added to the set in line 4.

**Algorithm 2:** Portfolio Formation

**input** : a set of algorithms $A$, target portfolio size $n$
**output**: portfolio $T$
1 $T \leftarrow \emptyset$
2 **for** $i \in \{1, \ldots, n\}$ **do**
3 $\quad a \leftarrow \arg\min_{a \in A} f(T, a)$
4 $\quad T \leftarrow T \cup \{a\}$

## 5 Empirical Evaluation

Our evaluation was conducted on search graphs representing grid-based video game maps. The search graph for each map is an 8-connected grid. Diagonal moves have a cost of $\sqrt{2}$ with cardinal moves having a cost of 1. The game maps are taken from the Moving AI benchmark (Sturtevant 2012). This consists of 342 maps from *Dragon Age: Origins*, *StarCraft*, *WarCraft III*, *Baldur's Gate II* (rescaled to $512 \times 512$). We evaluate the performance of these algorithms by running each of them over 17100 problems from the 342 maps in the Moving AI benchmark (Sturtevant 2012).

### 5.1 Dynamic Algorithm Selection

In an ideal setting the user would enable the game to run all of its optimizations, but they likely would want to play their map right away. To gain the advantages of using algorithm selection for optimizing real-time search performance we treat dynamic algorithm selection as a traditional image classification problem. The input for all of our classifications are a $227 \times 227$ pixel image of the problem the agent will be traversing. The output is the index of the algorithm, in this case corresponding to the algorithm to be used to solve the search problem. For the image classification we used AlexNet (Krizhevsky, Sutskever, and Hinton 2012), a popular deep-learning neural network, pre-trained on the ImagenNet dataset included in MATLAB 2017a Neural Network Toolbox.

The classification for the algorithm selection can be done at different levels of problem granularity. Bulitko (2016b) studied the selection problem at per-map and per-problem level, and showed the more granular of a selection the more specialized an algorithm can be resulting in higher performance. In particular, we consider the algorithm selection for four different granularities.

**Algorithm 3:** Dynamic Algorithm Selection

**input** : map $m$, Neural Network $n$
**output**: a
1 $p_i \leftarrow P(a_i \in a^*_A(m))$
2 $[p1, ...p_n] \leftarrow$ n(m)
3 $a \leftarrow \arg\max([p_1, ...p_n])$

## 6 Map-based Algorithm Selection

For all of the classifications based on selecting an algorithm from a map we run a trials which create a split that separates $75\%$ of the problems for a training set and $25\%$ for a test set. For the map-level classifications the input is an image of the map. Each training input is assigned a label corresponding to the best algorithm from the portfolio for that map. The $25\%$ test set and the performance of the algorithms are completely withheld from both the training of the network and portfolio formation. Using the training set's map-algorithm pairs we train the network. The maps are rescaled to match the $227 \times 227$ pixel input of the network. We repeat the splits over $20$ trials to reduce the chance that the test and training split happened to be favorable.

We then use the newly trained network to select which algorithm from the portfolio to use for each of the remaining $25\%$ of the maps as shown in Algorithm 3. The network uses a softmax final layer to give a probability distribution of each label as shown in line 2, over the $n$ algorithms in the portfolio. This can be conceptualized as how likely the network thinks each label is. For example, in the case where a network has four potential labels the distribution for algorithms 1, 2, 3, and 4 could be $0.24$, $0.25$, $0.25$, and $0.26$ respectively. The network would recommend using algorithm $4$ even though it is only slightly more confident in that label. Line 3 selects the algorithm to use for the inputed map image given the probability distribution. We refer to a perfect selector as an `oracle` meaning that it always selects the correct label for the input.

### 6.1 Game-type Algorithm Selection

The highest aggregation we consider is the type of game the agent will be traversing. For this classification the input is an image representing the map and the output is the algorithm which performed best on that game type. Moving AI contains role-playing games (RPG) and real-time strategy (RTS) games meaning we are using the 2 best out of the $1000$ algorithms from the training data for our selection. There are $231$ role-playing game maps in Moving AI from *Baldur's Gate II* and *Dragon Age: Origins*. The remaining $111$ maps are real-time strategy games from *StarCraft* and *WarCraft III*.

Our game-type classification achieved an average accuracy of $96 \pm 2.2\%$ for predicting which type of game the map is from. The average suboptimality achieved by our network and the oracle selector was $12.33 \pm 1.0$ while the suboptimality achieved by the best single algorithm was $12.58 \pm 1.1$

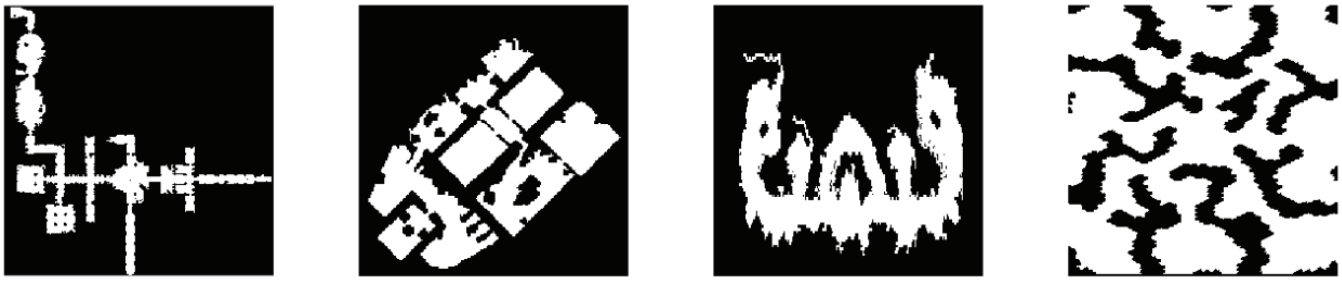

Figure 1: Sample maps from *Dragon Age: Origins*, *Star Craft*, *WarCraft III*, and *Baldur's Gate II*.

in comparison. Using the network resulted in better performance 70% of the time for our 20 trials.

### 6.2 Per-game Algorithm Selection

We next consider the task of selecting the best algorithm from our 1000 possible algorithms for each of the 4 Moving AI games in our 256 map training set. Conceptually each map is assigned an algorithm meaning the algorithm selection can be thought of as selecting which game the map is most similar too and using the corresponding algorithm. The remaining 86 maps are considered to be new maps with no information regarding which game they are from. Our classifier will predict which game a new map could be from, and then assign the corresponding best algorithm to the newly classified map. We use the best algorithm for that game to measure the suboptimality on the newly classified map. In this test the best static algorithm is the algorithm which did best across all 256 maps. The oracle 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.

Our ability to classify which game a map was from obtained a lower accuracy than the game-type classification with a accuracy of $90 \pm 3.1\%$. The average suboptimality for the oracle selector is $12.52 \pm 1.3$, our network selection resulted in an average suboptimality of $12.54 \pm 1.3$, and using the best static algorithm suboptimality of $12.72 \pm 1.2$. Selecting an algorithm dynamically resulted in better performance 65% of the time over our 20 trials.

### 6.3 Per-map Algorithm Selection

Per-map differs from per-game and game-type classification in that it does not rely on preconceived labels such as game-type. Instead we use Algorithm 2 to determine our portfolio. Each map is assigned the label corresponding to the best performing algorithm from the portfolio. The oracle in this scenario represents both a classifier with perfect accuracy but also a lower bound for performance when using this technique, as the correct label is the best performing algorithm from the portfolio. 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 $12.20 \pm 1.2$ in comparison to the best static algorithm achieving a result of $12.27 \pm 1.3$. The oracle selector which always selected the best algorithm out of the 2 achieved a suboptimality of $11.27 \pm 1.1$. Over the 20 trials the dynamic selection achieved better results than the static algorithm 60% of the time.

## 7 Problem Based Algorithm Selection

Problem based algorithm selection is the ability to select an algorithm for an individual problem as opposed to selecting an algorithm for all problems on a map. This can result in better performance for a map. To help the network focus on selecting an algorithm for each individual search problem we reduce the classification to a single map.

The network is trained using images of a single map but with information regarding the individual search problem also embedded on the map image. In particular we added a green $3 \times 3$ square pixel at the start and another red $3 \times 3$ square pixel at the goal location and a 1 pixel width line connecting the start and goal. We train the network by feeding in the problem images with the corresponding algorithm to use for the input problem. The newly trained network is then used in the same fashion as described in Section 6.

Training a network for each individual map increases the resources required for algorithm selection because the training process is repeated for each map. Our testing was limited to a map from *Dragon Age: Origin's* map from the Moving AI benchmark which was slightly smaller ($207 \times 196$) than the networks input dimensions and had 1169 problems on the map. Using a portfolio size of 2 again the suboptimality achieved by the network was $14.20 \pm 0.7$ in contrast to the best static algorithm achieving a result of $14.77 \pm 0.5$. The accuracy of network was $70 \pm 2.3\%$. If the network had the 100% accuracy of the oracle selector the performance achieved would have been $11.35 \pm 0.5$ suboptimality. The 20 trials of the per-problem testing achieved a better average performance than the static algorithm 90% of the time.

## 8 Current Limitations and Future Work

Our work provides several examples with varying degrees of performance gains and classification accuracy in using dynamic algorithm selection. The potential performance improvements were limited by not developing specialized solvers but instead selecting out the 1000 algorithms tested. Given the high degree of accuracy for per-game and game-

| | | Suboptimality | | |
|---|---|---|---|---|
| Algorithm Selection Type | Accuracy | Static | ANN | Oracle |
| Game-type | $96 \pm 2.2\%$ | $12.58 \pm 1.1$ | $12.33 \pm 1.0$ | $12.33 \pm 1.0$ |
| Per-game | $90 \pm 3.1\%$ | $12.72 \pm 1.2$ | $12.54 \pm 1.3$ | $12.52 \pm 1.3$ |
| Per-map | $55 \pm 5.3\%$ | $12.27 \pm 1.3$ | $12.20 \pm 1.2$ | $11.27 \pm 1.1$ |
| Per-problem | $70 \pm 2.3\%$ | $14.77 \pm 0.5$ | $14.2 \pm 0.7$ | $11.35 \pm 0.5$ |

Table 1: Results of Algorithm Selection: Compares using a static algorithm, our network (ANN) and a perfect selector (Oracle) as well as reports classification accuracy for our network.

type classification using an optimization tool such as SMAC to truly create specialized solvers may lead to lower suboptimality. The number of game types and maps evaluated were limited to the Moving AI benchmark. Ideally, we would have several maps from multiple games across a diverse set of genres to examine if the same classification accuracy can be achieved on a larger test bed.

Our greedy method to select a portfolio only works if the classification accuracy is sufficiently high. When increasing the portfolio size past two with the same technique, the algorithms added often performed poorly across the majority of maps but excelled only at one and were added because the portfolios are formed under the assumption of a perfect selector. This resulted in larger portfolios creating worse results. Creating a portfolio formation algorithm that would only add algorithms that passed a baseline requirement could help allow for larger portfolio sizes. Another extension is to have the approach to select from algorithms outside of the framework we used as well.

As mentioned in Section 5.1, our network is trained as a simple 0-1 classification problem meaning the network's loss function punishes choosing any algorithm that is not the optimal algorithm equally. One way to overcome this is by replacing the loss function used in AlexNet with one that incorporates a weighted loss.

## 9 Conclusions

In this paper, we built on the recent work which used evaluating algorithms on a domain in order to select which algorithm to use for the remaining problems on that domain. We showed that networks could be used to achieve better results than always sticking with a single algorithm. We also showed the degree of accuracy a neural network could classify which type of game a map was from in addition to which game it belonged to. Neural networks are an excellent tool that is readily available to game developers as there are several libraries to implement a wide range of networks. Using an off-the-shelf configuration of AlexNet we were able to obtain successful results and provide possible methods in our future works for how a game developer could potentially increase the results even further.

## Acknowledgments

We thank Shelby Carleton, Delia Cormier, and the anonymous reviewers for providing valuable feedback. The funding was provided by NSERC.

## References

Bulitko, V. 2016a. Evolving real-time heuristic search algorithms. In *Proceedings of the Fifteenth International Conference on the Synthesis and Simulation of Living Systems*, 108–116.

Bulitko, V. 2016b. Per-map algorithm selection in real-time heuristic search. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*, 143–149.

Bulitko, V. 2016c. Searching for real-time search algorithms. In *Proceedings of the Symposium on Combinatorial Search*, 121–123.

Bulitko, V. 2016d. Weighted lateral learning in real-time heuristic search. In *Proceedings of the Symposium on Combinatorial Search*, 10–19.

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

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

Ishida, T. 1992. Moving target search with intelligence. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 525–532.

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. In *AI Magazie*, volume 35, 48–60.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In Pereira, F.; Burges, C. J. C.; Bottou, L.; and Weinberger, K. Q., eds., *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc. 1097–1105.

Lelis, L. H. S.; Franco, S.; Abisrror, M.; Barley, M.; Zilles, S.; and Holte, R. C. 2016. Heuristic subset selection in classical planning. In *Proceedings of the Twenty-Fifth Inter-*

*national Joint Conference on Artificial Intelligence*, 3185–3191.

Loreggia, A.; Malitsky, Y.; Samulowitz, H.; and Saraswat, V. A. 2016. Deep learning for algorithm portfolios. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*.

Rice, J. R. 1976. The algorithm selection problem. *Advances in Computers* 15:65 – 118.

Rivera, N.; Baier, J. A.; and Hernández, C. 2013. Weighted real-time heuristic search. In *Proceedings of Autonomous Agents and Multiagent Systems*, 579–586.

Rivera, N.; Baier, J. A.; and Hernndez, C. 2015. Incorporating weights into real-time heuristic search. *Artificial Intelligence* 225:1 – 23.

Sharon, G.; Sturtevant, N. R.; and Felner, A. 2013. Online detection of dead states in real-time agent-centered search. In *Proceedings of the Symposium on Combinatorial Search*, 167–174.

Shimbo, M., and Ishida, T. 2003. Controlling the learning process of real-time heuristic search. *AI* 146(1):1–41.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144 – 148.