The Eleventh International Symposium on Combinatorial Search
(SoCS 2018)

A Neural Network for Decision Making in Real-Time Heuristic Search

Franco Muñoz,[1] **Miguel Fadic,**[1] **Carlos Hernández,**[2] **Jorge A. Baier,**[1,3]
[1]Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile
[2]Departamento de Ciencias de la Ingeniería, Universidad Andrés Bello, Chile
[3]IMFD Chile

Abstract

Most real-time heuristic search algorithms solve search problems by executing a series of episodes. During each episode the algorithm decides an action for execution. Such a decision is usually made using information gathered by running a bounded, heuristic-search algorithm. In this paper we report on a real-time search algorithm that does not use a search algorithm to choose the next action to be applied. Rather, it uses a neural network whose input is local information about the search graph, comparable to the information that would be used by a bounded search algorithm. We describe a supervised learning approach to training such a network. Our three types of maps from the Moving AI benchmarks, shows that our algorithm is, in some cases, substantially superior to algorithms that have access to the same information about the graph. One of our most important conclusions is that our extended set of features important: indeed, using features beyond the heuristic seems key to achieving good performance.

Introduction

To solve a search problem, most real-time heuristic search (RTHS) algorithms run a loop similar to the one shown in Algorithm 1. Line 2 runs a standard heuristic search algorithm that is bounded in the sense that its execution time is bounded from above by a constant. Line 3 updates the heuristic function. Usually referred to as the *learning step*, the heuristic update is needed for termination but also allows subsequent searches to become more informed. Finally, Lines 4-5 selects and executes the next action (or actions), usually based on the computation carried out in Line 2. An example of an algorithm that conforms to Algorithm 1 is LSS-LRTA* (Koenig and Sun 2009). It uses a bounded A*, allowed to expand at most k states, as a search algorithm (Line 2). We assume $Open$ and $Closed$ are the open list and closed list of such a run. For the update, it uses a variant of the Dijkstra's algorithm to set $h(s) := \min_{s \in Open} c(s, s') + h(s')$, for every state s in $Closed$, where $c(s, s')$ denotes the cost of an optimal path between states s and s'. Finally, it chooses σ as the actions in the path from the current state to the state of least f-value in $Open$ (Line 4).

Algorithm 1: Skeleton of a typical RTHS algorithm

```
1 while the goal state has not been reached do
2     Run a bounded heuristic search algorithm, rooted in the
        current state.
3     Update the heuristic function h, of some states
4     Compute a sequence of actions σ using information
        previously computed in Line 2
5     Perform as many actions from σ as possible, observing the
        environment and updating the search graph suitably
```

The use of a search algorithm in Line 2 seems very ingrained in the RTHS literature. Nevertheless, some RTHS algorithms, especially when run with a low lookahead value k, can be re-interpreted as applying ad hoc decision rules. For example, depression-avoiding version of LSS-LRTA* (daLSS-LRTA*; Hernández and Baier 2012) is a variant of LSS-LRTA* that when run with lookahead $k = 1$ chooses to move to $\arg\min_{s' \in M(s)} c(s, s') + h(s')$, where $M(s)$ are the neighbors of s whose h-value has changed the least among all of the neighbors of s. This decision rule, in real-time grid pathfinding, yields, on average, a one-order-of-magnitude improvement on solution cost over LSS-LRTA*, whose decision rule is to move to $\arg\min_{s' \in N(s)} c(s, s') + h(s')$, where $N(s)$ are the neighbors of s.

So a natural question that arises is: *can other decision rules lead to even better performance?* In this paper we focus on a more specific question, namely: *is it possible to obtain better-than-state-of-the-art performance by replacing the decision module (i.e., Line 4) with a neural network?*

In the rest of this paper, we propose an architecture for a neural network for making real-time decisions in real-time grid navigation in unknown terrain. Its input is information of states in a bounded vicinity around the current state. Among this information, we included the heuristic function, whether or not there are obstacles, and the visit count. The output of the network is which state to move to. In addition, we describe a *supervised imitation learning approach* (Ross, Gordon, and Bagnell 2011) to training such a net that uses, as examples, data generated by a variant of repeated A*. Then, we present an experimental evaluation in which we show that our learned decision rule outperforms state-of-the-art decision rules for planning in unknown terrain. We show a specific configuration that substantially outperforms a state-of-the-art real-time heuristic search algorithm that uses the same amount of information. In addition, we show

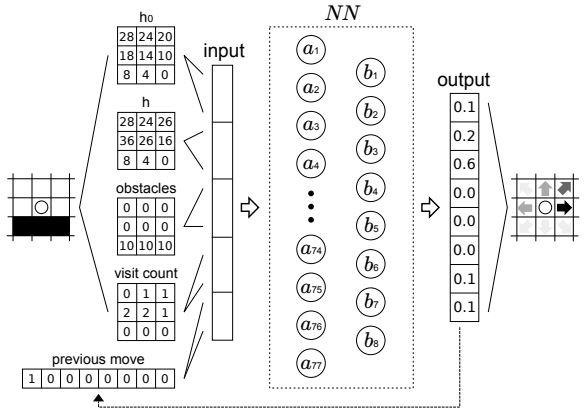


Figure 1: Architecture of the neural network using $\alpha = 1.75$ resulting in 77 neurons for the first layer.

that in two groups of maps our solutions are substantially better than those obtained by a state-of-the-art algorithm that spends a similar amount of time per planning episode. One of our most important conclusions is that each feature we used allows us to gain some performance. What is most interesting is that h, the heuristic function, does not seem to be the most important feature; rather, the visit counter seems to provide more effective guidance.

Real-Time Grid Navigation

Our algorithm is designed for real-time grid path planning on an 8-connected grid, in which horizontal, vertical, and diagonal moves are possible. We assume *unknown terrain*; that is, the agent does not know the location of the obstacles in advance, which can only be observed from a neighboring cell (i.e., visibility range is 1). The objective is to move the agent from an initial cell to a goal cell.

The Neural Network

For our algorithm we use a feedforward neural network (NN). The input to the NN is information about the current state and its successors (described below as *Input Features*). The output is the a decision; in our case, a next move to be made.

Input Features We use a set of features describing the 3×3 window that includes the current state, and the 8 immediate successors. There are many options that we could have included. At the very least we wanted to use the heuristic function, since this function is the only source of information for traditional RTHS algorithms. As our research advanced we were curious as to whether or not using additional features would add to performance. We eventually decided to test the following features.

1. $\mathbf{h}_0$: The initial octile distance from each cell to the goal. The values are min-normalized, that is, normalized by subtracting the minimum value in the 3×3 window.
2. **h**: The min-normalized heuristic value of every cell using the same update rule of LSS-LRTA*$(k = 1)$. (More details in the next section.)
3. **obstacles**: Contains a 0 if no obstacle is present in the cell, and a 10 otherwise.
4. **visits**: The number of times the agent has visited the cell.
5. **previous move**: A vector of size 8 where each position represents one of the 8 possible moves, with a 1 in the position corresponding to the previous move made by the agent to reach the current state and a 0 in all remaining positions.

For **h** and $\mathbf{h}_0$ we ignore the obstacles, so all cells are treated as empty for these features. Moreover, each window of features is rotated such that the element with value 0 (there is always one, since it is min-normalized) are rotated so that the value 0 ends up being located in one of two selected positions (depending on whether it is a diagonal or cardinal position).

Architecture Features are concatenated in a single vector and go through a fully connected ReLU layer whose output size is αA, where α is a real constant and A the size of the input vector. Finally, the output layer has a softmax activation with output size 8, which is a one-hot codification of the movement that the agent should make given the input. Figure 1 illustrates the architecture.

An NN-based RTHS Algorithm

Algorithm 2 presents a pseudo-code of NNRT, an NN-based RTHS algorithm we propose. NNRT can be viewed as a standard RTHS algorithm that takes into account the fact that the decision module (i.e. the NN) may not, on its own, guarantee completeness.

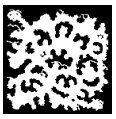
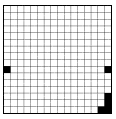
Algorithm 2: Our NN-based RTHS algorithm

```
1  procedure NNRT
       Input : A grid navigation problem; a neural network, N; a
               complete RTHS algorithm A
2      s_c ← initial cell of the problem
3      while the goal state has not been reached do
4          Update the h-value of the current state, s_c, with:
             h(s_c) := min_{s' ∈ N(s_c)} c(s, s') + h(s')
5          if visits(s_c) < M then
6              next ← move computed by network N
7          else
8              next ← move computed by RHTS algorithm A
9          visits(s_c) ← visits(s_c) + 1
10         Perform a move towards next, updating the search graph
             with observed changes in the immediate neighborhoods
11         s_c ← next
```

The algorithm moves using the output of the NN except when the current cell has been visited M or more times, in which case it uses A, a complete RTHS algorithm, as an oracle subroutine to obtain a move. The reason why we need to use an additional, complete RTHS algorithm is to avoid loops: our training mechanism does not guarantee that the resulting neural network yields a decision rule capable of solving any given RTHS problem.

Proposition 1 *NNRT is complete; that is, it will solve any given solvable problem under the assumption that, as an ob-*

Table 1: Our approach versus some well-known RTHS algorithms.

W3	LRTA	wLRTA(8)	daRTAA*(1)	daRTAA*(4)	daRTAA*(4)	NN_0-W3	NN-W3	
Suboptimality	16.11	3.25	2.73	2.99	2.99	8.05	2.98	
Time per Eps.	0.48	0.89	0.59	1.92	1.92	1.91	2.16	
% Oracle Moves						18.78%	4.01%	
BG	**LRTA**	**wLRTA(8)**	**daRTAA*(1)**	**daRTAA*(4)**	**daRTAA*(13)**	**NN_0-BG**	**NN-BG**	
Suboptimality	67.48	12.37	7.81	6.52	3.95	44.73	3.83	
Time per Eps.	0.48	0.83	0.55	1.99	6.80	1.65	1.96	
% Oracle Moves						32.47%	17.11%	
Rooms	**LRTA**	**wLRTA(8)**	**daRTAA*(1)**	**daRTAA*(4)**	**daRTAA*(80)**	**NN_0-Rooms**	**NN-Rooms**	
Suboptimality	54.24	8.22	6.38	5.72	1.50	6.94	1.43	
Time per Eps.	0.47	0.77	0.53	1.84	30.76	1.74	2.08	
% Oracle Moves						26.71%	3.06%	

servation is made, the costs of the arcs of the search space may only increase.

Observe this proposition means that NNRT can solve any real-time grid navigation problem when the map is fixed, but initially unknown.

Training via Supervised Imitation Learning

We use supervised imitation learning (SIL) as the learning approach for the network. In SIL, the input examples describe how an expert would solve the task in various scenarios. The objective of learning is thus to learn to behave like the expert. In our experiments, we attempted to use a *dagger* (Ross, Gordon, and Bagnell 2011), a state-of-the-art approach to SIL. Dagger did not yield better results than an approach we describe below, which, to the best of our knowledge, is novel.

Our expert—henceforth referred to as RT-Train—is an algorithm that is a variant of Repeated A* (Koenig and Likhachev 2002), suitable for unknown environments. Like Repeated A*, it searches for an optimal path over the graph in memory, and then performs the steps in such a solution, observing the map and updating its memory as it moves. This is carried out until a move is not possible, in which case it replans and the cycle is repeated. Unlike Repeated A*, each time it performs a move, it updates the heuristic of the current state, just like Algorithm 2 does. Observe that this algorithm has some commonalities with standard LSS-LRTA*($k = 1$), since it updates h in the same way, updates the memory in the same way, and realizes the path is blocked only when it cannot perform a move (that is, it has a visibility window limited to the immediate neighborhood). However, RT-Train it is not an RTHS algorithm since it searches for a complete path to the goal, thus is capable of computing an optimal path given the current map. RT-Train updates the heuristic to provide our network with an example that would more accurately reflect the value of h during execution. Other features like visit counts are also updated.

For training we use a set Π of maps. For each map we generate two sets of random problems P_{train} and P_{val}. We denote by EX the list of examples used for training. Training can be seen as divided in two steps: *initial training* and *retraining*.

Initial Training We run RT-Train over a subset P_{init} of P_{train}. We add to EX one example per move of RT-Train. Using these examples, we train a first model.

Retraining We run a modified version of NNRT using the last trained model. In each iteration, this version of NNRT checks whether or not the situation it is looking at (given by the network's features) is in EX. Let EX' be an empty list. If the current situation is not in EX then it runs M steps of RT-TRAIN. Each of those M steps generates a new example that is stored in EX'. When every problem in P_{train} has been solved, let EX be $EX \cup EX'$. We train our network with the examples in EX, we store the trained model and repeat the retraining step unless we have reached a maximum of retraining steps.

After retraining, for each of the NN models trained, we run NNRT over P_{val}. We calculate NNRT's suboptimality as $S_{\text{NN}}/S_{\text{RT-Train}}$, where S_{NN} is the total number of moves employed by NNRT. $S_{\text{RT-Train}}$ is the number of moves needed by RT-Train. We select the model that has obtained the best suboptimality score.

Experimental Evaluation

The objectives of our evaluation were threefold. First, we wanted to understand the impact of the retraining phase on performance. Second, we wanted to see whether or not our approach could achieve state-of-the-art performance on different types of maps. Third, we wanted to understand which features had influence in performance.

To achieve the second objective, we compare our algorithm to depression-avoiding algorithms, which are good representatives of the state of the art in real-time grid navigation. In the analisys we compare NNRT versus algorithms that use the same information, versus algorithms that spend a comparable time per episode, and versus algorithms that obtain a comparable suboptimality.

We implemented our neural network in Python using the Keras (Chollet 2015) library. For the training phase, we used a batch size of 32, a learning rate of 0.001 and 5 epochs. As optimizer we used the Adam algorithm (Kingma and Ba 2014) with categorical crossentropy as loss function. Also, for first layer we set $\alpha = 1.75$. We ran 50 retrainings and pick the model with the best suboptimality with respect to the validation set. To generate training data we used $M = 6$.

Table 2: Suboptimality obtained for trained models with different combinations of features in Rooms

Features	Suboptimality
obstacle + h_0	39.26
obstacle + h_0 + last move	4.94
obstacle + h_0 + visit_count	1.81
obstacle + h_0 + last move + visit count	1.52
obstacle + h_0 + h	10.38
obstacle + h_0 + h + last move	4.74
obstacle + h_0 + h + visit count	1.79
obstacle + h_0 + h + last move + visit count	1.43

These parameters were chosen because they were the ones that produced best results during our preliminary tests. For all tests we used daRTAA(1) as oracle.

We trained different neural networks for different types of maps. To that end, we generated three groups of 10 maps from MovingAI (Sturtevant 2012). We separated each group in 8 maps that were used for training, and 2 maps that were used for testing. The groups were composed of maps from World of Warcraft III (WC3), Baldurs Gate II 512 (BG) and Rooms 32 (ROOMS). For each group we trained different models: NN-W3, NN-BG and NN-Rooms. We used $(|P_{\text{init}}|, |P_{\text{train}}|)$ equal to $(40, 400)$, $(40, 400)$, and $(80, 800)$, respectively. For each group, we used $|P_{\text{val}}| = 400$.

For each of the two test maps, we generated 1000 problems. For the sake of comparison, we ran LSS-LRTA* (k = 1), wLRTA*(8) (Rivera, Baier, and Hernández 2015) and daLRTA* (Hernández and Baier 2012) with several lookahead values. All algorithms were implemented in C, including functions required to use the neural network. We used OpenBLAS (Wang et al. 2013) to implement the matrix operations. We ran the test over a single thread process on an Intel Core i5 Linux machine.

Table 1 presents the results of our NN aproaches and a selection of RTHS algorithms. Specifically, we include three configurations of daRTAA. The first one, daRTAA(1) is an algorithm whose decisions are made by looking at the same 3×3 window as NNRT. The next configuration, daRTAA(*) a configuration of daRTAA whose runtime per episode is similar to the runtime per episode (Time per Eps.) of NNRT. The last daRTAA configuration varies between groups of maps and corresponds to an algorithm that obtains a suboptimality score similar to our algorithm. For NNRT, we also include the percentage of oracle moves; that is, those moves decided by the oracle RTHS algorithm. (A video with a comparison on the WC3 map is available at https://streamable.com/3ych9.)

We include two versions of our algorithms to assess the impact of the retraining phase: NN_0 is the model after initial training, and NN is the best model found. We observe a substantial difference in the suboptimality score across all groups. NN makes fewer oracle calls than NN_0.

NN vs algorithms that use the same information LRTA*, wLRTA* and daLRTA* use the same information as NN since all of them expand only one node before making a decision (and thus have access to the same 3×3 window). We observe NN obtains the best suboptimality except for WC3, where daLRTA* is slightly better. This happens because heuristic depressions in WC3 are small, so daLRTA* avoidance mechanism is effective. The other algorithms that use the same information require less time than NNRT since NNRT performs matrix multiplications to decide the next move.

NN vs algorithms that spend a comparable time per episode

As is shown in the table, daRTAA*(4) has a similar average time per episode with NN in all maps. However, NN outperforms daRTAA(4) in terms of suboptimality, especially in Rooms. **NN vs algorithms that obtain a comparable suboptimality** daRTAA(4), daRTAA(13) and daRTAA(80) obtain a comparable suboptimality with NN in WC3, BG and Rooms, respectively. The time per episode of daRTAA(13) and daRTAA(80) is as much as 14.8 times higher than that obtained by NNRT. This is explained by the large lookahead value. On the other hand, the time per episode of daRTAA(4) in similar to NN due to small lookahead.

Impact of Features on Performance For our last objective we evaluated the performance of NNRT using different sets of features (Table 2). Using only h_0 yields worst performance but still allows us to outperform LRTA*, which uses the updated heuristic (h, not h_0), to make decisions. We observe that adding more features always yields better performance. The visit counter and the previous move are critical features for our approach; more so than h. It is interesting to note that no real-time heuristic search algorithm we know of uses such features for decision-making.

Related Work

Machine learning (ML) techniques has been applied before to real-time heuristic search. Huntley and Bulitko (2013) use ML to predict performance and algorithm parametrization. Kiesel, Burns, and Ruml (2015) use ML to learn what lookahead parameter to use in a particular situation for a version of LSS-LRTA* with dynamic lookahead. Related is also an approach that automatically searches for a configuration of an RTHS algorithm to optimize performance (Bulitko 2016; Sigurdson and Bulitko 2017).

A body of work exists on the application of ML to offline search. An example is heuristic function learning (Arfaee, Zilles, and Holte 2011; Lelis et al. 2012). Finally, also related is recent work in deep reinforcement learning (Mnih et al. 2015), in which a neural net is used to learn a Q function which is related to our decision-making neural net.

Summary and Conclusions

We presented a novel real-time heuristic search algorithm, NNRT, that carries out no search but instead makes move decisions using a neural network that receives as input limited information about the map. Our training approach has two phases: an initial training with examples generated by a version of repeated A* and a retraining phase that adds more examples drawn from an actual run of the algorithm. Our evaluation shows that retraining is important, and that our algorithm can substantially outperforms state-of-the-art

algorithms that use the same information. An important conclusion is that features not currently used by real-time search algorithms, like visit count, seem key to performance, more so than the heuristic function.

Acknowledgements

We thank Vadim Bulitko for his comments on an earlier draft of this paper. We acknowledge support from Fondecyt via grant number 1161526.

References

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artificial Intelligence* 175(16-17):2075–2098.

Bulitko, V. 2016. Searching for real-time heuristic search algorithms. In Baier, J. A., and Botea, A., eds., *Proceedings of the 9th Symposium on Combinatorial Search (SoCS)*, 121–122. AAAI Press.

Chollet, F. 2015. Keras. https://github.com/fchollet/keras.

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

Huntley, D. A., and Bulitko, V. 2013. Search-space characterization for real-time heuristic search. *CoRR* abs/1308.3309.

Kiesel, S.; Burns, E.; and Ruml, W. 2015. Achieving goals quickly using real-time search: Experimental results in video games. *Journal of Artificial Intelligence Research* 54:123–158.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *CoRR* abs/1412.6980.

Koenig, S., and Likhachev, M. 2002. D* lite. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, 476–483.

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

Lelis, L. H. S.; Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2012. Learning heuristic functions faster by using predicted solution costs. In *Proceedings of the 5th Symposium on Combinatorial Search (SoCS)*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

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

Ross, S.; Gordon, G. J.; and Bagnell, D. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligenceand Statistics (AISTATS)*, 627–635.

Sigurdson, D., and Bulitko, V. 2017. Deep learning for real-time heuristic search algorithm selection. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 108–114.

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

Wang, Q.; Zhang, X.; Zhang, Y.; and Yi, Q. 2013. Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, 25:1–25:12. New York, NY, USA: ACM.