Evolving Real-time Heuristic Search Algorithms

Vadim Bulitko
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
bulitko@ualberta.ca

Abstract

Heuristic search is a core area of Artificial Intelligence, successfully applied to planning, constraint satisfaction and game playing. In real-time heuristic search autonomous agents interleave planning and plan execution and access environment locally which make them more suitable for Artificial Life style settings. Over the last two decades a large number of real-time heuristic search algorithms have been manually crafted and evaluated. In this paper we break down several published algorithms into building blocks and then let a simulated evolution re-combine the blocks in a performance-based way. Remarkably, even relatively short evolution runs result in algorithms with state-of-the-art performance. These promising preliminary results open exciting possibilities in the field of real-time heuristic search.

1 Introduction and Related Work

Artificial Life (ALife) settings afford a researcher an intuitive testbed to study autonomous agents. In particular, ALife has been used to study emergence of various cognitive mechanisms, including a source of rewards (Ackley and Littman, 1991) in a Reinforcement Learning setting (Sutton and Barto, 1998). In this paper we propose to use ALife to study heuristic search algorithms. Heuristic search is a core area of Artificial Intelligence with a long history (Hart et al., 1968) and a broad applicability to planning, game-playing and constraint-optimization tasks. Heuristic search algorithms take a search graph and a start and goal states and output a path through the graph that connects the start and the goal. A canonical example is pathfinding on a road map: one can ask their in-car GPS to find a route from Edmonton to Cancun. Shorter routes may be preferred and quick computation times are valued.

We focus on real-time heuristic search — a subclass of agent-centered algorithms (Koenig, 2001) where the agent has to act before a full solution to the search problem is computed and has access only to the environment and data in the vicinity of the agent's current state. In other words, plan execution must be interleaved with the planning process and there is no global view of the world. These constraints would be important in the context of a self-driving car: its steering algorithm needs to issue commands to the steering wheel so many times per second while the GPS is computing the full route, *regardless* of how distant the goal is. Another application of real-time heuristic search is distributed search such as routing in *ad hoc* sensor networks (Bulitko and Lee, 2006).

Starting with LRTA* (Korf, 1990) real-time heuristic search agents interleave three processes: local planning, heuristic learning and move selection. In over the two decades since LRTA*, researchers have explored different methods for looking ahead during the planning stage of each cycle (Koenig and Sun, 2009); different heuristic learning rules (Bulitko; Hernández and Meseguer; Bulitko and Lee; Rayner et al.; Koenig and Sun; Rivera et al., 2004; 2005; 2006; 2007; 2009; 2015) and different move selection mechanisms (Ishida; Shue and Zamani; Shue and Zamani; Shue et al.; Bulitko and Lee; Hernández and Baier, 1992; 1993a; 1993b; 2001; 2006; 2012). Finally, information in addition to the heuristic has been learned during (Bulitko et al.; Sturtevant et al.; Sturtevant and Bulitko; Sharon et al., 2007; 2010; 2011; 2013) and before (Bulitko et al.; Bulitko et al.; Botea; Bulitko et al.; Lawrence and Bulitko, 2008; 2010; 2011; 2012; 2013) the search.

The number of techniques proposed by the researchers in the field of real-time heuristic search is overwhelming. More importantly, the interactions between these techniques are difficult to analyze empirically (Bulitko and Lee, 2006) or theoretically (Sturtevant and Bulitko, 2014). In this paper, we frame the problem of finding a high-performance combination of real-time heuristic search techniques as a survival task. To do so we set up a colony of autonomous real-time heuristic search agents whose genes determine their operation in their life-time. Finding high-quality solutions sustains an agent and eventually allows it to mate and reproduce. During the reproduction new agents are born, each with a slightly different genetic code.

Unlike human researchers in the field of real-time heuristic search, such a simulated evolution has no prior expectations, intuitions or biases. It simply conducts a randomized parallel search of a large space of real-time heuristic search algorithms. Yet, preliminary results in the standard

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/108/1904158/978-0-262-33936-0-ch024.pdf by guest on 20 August 2022

testbed of pathfinding on video-game maps are promising. An evolution run on a desktop computer in under a day led to a new real-time heuristic search algorithm that outperforms the state of the art. The emergence of such new high-performance algorithms appears to be a robust phenomena as we have repeated the evolution process several times, with similar results.

However, evolution has two downsides relative to the traditional process of designing real-time heuristic search algorithms: (i) it does not prove any theoretical properties of the new algorithms and (ii) it does not intuitively explain their performance. Thus, we suggest using the evolution as a computer-assisted exploratory step, to be followed by theoretical analysis and additional manual design.

The rest of the paper is organized as follows. Section 2 formally defines the problem we are attempting to solve. We then review the common framework of heuristic-learning real-time heuristic search algorithms in Section 3 and describe the building blocks in Section 4. We conduct a simulated evolution in a class of real-time heuristic search algorithms in Section 5 and present the empirical results in Section 6. We then conclude with directions for future work.

2 Problem Formulation

In line with previous research, we define a *search problem* $\mathbf{S}$ as the tuple (S, E, c, s_0, s_g, h) where S is a finite set of *states* and $E \subset S \times S$ is a set of *edges* between them. S and E jointly define the search graph which is assumed to be undirected: $\forall s_a, s_b \in S\,[(s_a, s_b) \in E \implies (s_b, s_a) \in E]$ and has no self-loops: $\forall s \in S\,[(s, s) \notin E]$. The graph is weighted by the strictly positive edge costs $c : E \to \mathbb{R}^+$ which are symmetric: $\forall s_a, s_b \in S\,[c(s_a, s_b) = c(s_b, s_a)]$. Two states s_a and s_b are *immediate neighbors* iff there is an edge between them: $(s_a, s_b) \in E$; we denote the set of immediate neighbors of a state s by $N(s)$. A *path* P is a sequence of states $(s_0, s_1, \ldots, s_n)$ such that for all $i \in \{0, \ldots, n-1\}$, $(s_i, s_{i+1}) \in E$. We assume that the search graph (S, E) is connected (i.e., any two vertices have a path between them) which makes it safely explorable.

At all times $t \in \{0, 1, \ldots\}$ the agent occupies a single state $s_t \in S$, called the *current state*. The state s_0 is the start state and is given as a part of the problem. The agent can change its current state, that is, move to any immediately neighboring state in $N(s)$. The traversal incurs a travel *cost* of $c(s_t, s_{t+1})$. The agent is said to solve the search problem at the earliest time T it arrives at the goal state: $s_T = s_g$. The *solution* is a path $P = (s_0, \ldots, s_T)$: a sequence of states visited by the agent from the start state until the goal state. The cumulative cost of all edges in a solution is called *solution cost* and is formally defined as $c_A(\mathbf{S}) = \sum_{t=0}^{T-1} c(s_t, s_{t+1})$ for algorithm A. The cost of the shortest possible path between states $s_a, s_b \in S$ is denoted by $h^*(s_a, s_b)$. We abbreviate $h^*(s, s_g)$ as $h^*(s)$. We define *suboptimality* of the agent on a problem as the ratio of the solution cost the agent incurred to the cost of the shortest possible solution: $\alpha(A, \mathbf{S}) = \frac{c_A(\mathbf{S})}{h^*(s_0)}$. For instance, suboptimality $\alpha(\text{LRTA*}, \mathbf{S}) = 2$ means that the agent driven by the LRTA* algorithm found a solution to $\mathbf{S}$ twice as long as optimal. Lower values are preferred; 1 indicates optimality.

The other performance measure we are concerned with in this paper is the *scrubbing complexity* (Huntley and Bulitko, 2013). It is defined as the average number of state visits the agent makes while solving a problem. Formally, let $v_A^{\mathbf{S}} : S \to \mathbb{N} \cup \{0\}$ be the number of state visits the agent driven by algorithm A made while solving a problem $\mathbf{S}$. The scrubbing complexity is then defined over the subset of states that the agent visited at least once: $S_{\text{visited}} = \{s' \in S \mid v_A^{\mathbf{S}}(s') \geq 1\}$ as $\tau(A, \mathbf{S}) = \frac{1}{|S_{\text{visited}}|} \sum_{s \in S_{\text{visited}}} v_A^{\mathbf{S}}(s)$. For instance, $\tau(\text{LRTA*}, \mathbf{S}) = 7.5$ means that while solving problem $\mathbf{S}$, on average the agent driven by the LRTA* algorithm visited a state 7.5 times (states that were not visited at all do not contribute to the average). Lower values of $\tau(\mathbf{S})$ are preferred since re-visiting states tends to look irrational to an external observer. This is a major reason why real-time heuristic-search methods are hardly used for pathfinding in actual video games. Instead, game developers prefer non-real-time heuristic search such as variants of path-refinement A* (Sturtevant, 2007).

In its operation the agent has access to a heuristic $h : S \to [0, \infty)$. The heuristic function is a part of the search problem specification and is meant to give the agent an estimate of the remaining cost to go. Unlike much literature in the field, we do *not* assume admissibility or consistency of the initial heuristic but require that $h(s_g) = 0$. The search agent can modify the heuristic as it sees fit as long as it remains non-negative and the heuristic of the goal state s_g remains 0. The heuristic at time t is denoted by h_t; $h_0 = h$.

We say that a search agent is *real time* iff its computation time between its moves is upper-bounded by a constant independent of the number of states in the search space. We will additionally require our search algorithms to be *agent centered* (Koenig, 2001) insomuch as they have access to the heuristic, states and edges only in a bounded vicinity of the agent's current state and the bound is independent of the number of states.

We say that a search agent is *complete* iff it solves any search problem as defined above. That is, it is required to terminate in the goal state s_g at time $T < \infty$. Since we deal with randomly generated algorithms which may or may not be complete, we impose an upper bound on their travel cost. Any algorithm whose suboptimality on a problem exceeds $\alpha_{\max}$ is said to not solve a problem. We implement this by monitoring the agent's travel cost and terminating an agent as soon as it reaches or exceeds $\alpha_{\max} h^*(s_0)$. The resulting suboptimality is then recorded and contributes to the average suboptimality of the agent over a set of problems.

The problem we tackle in this paper is to develop a real-

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/108/1904158/978-0-262-33936-0-ch024.pdf by guest on 20 August 2022

time heuristic search algorithm that has low suboptimality (α) and low scrubbing complexity (τ).

3 Basic Real-time Heuristic Search

As Section 1 presented, many ways of improving on LRTA* (Korf, 1990) towards the two measures have been proposed. In this paper we specifically focus on heuristic learning and movement rules. 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) and allow the agent to update its heuristic only in its current state. In other words, our local search space and the local learning spaces are limited to the agent's current state. With these limitations, LRTA* becomes Algorithm 1.

Algorithm 1: Basic Real-time Heuristic Search

input : search problem (S, E, c, s_0, s_g, h)
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 $\quad s_{t+1} \leftarrow \arg\min_{s \in N(s_t)} (c(s_t, s) + h_t(s))$
5 $\quad h_{t+1}(s_t) \leftarrow \max\left\{h_t(s_t), \min_{s \in N(s_t)} (c(s_t, s) + h_t(s))\right\}$
6 $\quad t \leftarrow t + 1$
7 $T \leftarrow t$

A search agent following the algorithm begins in the start state s_0. It then executes a fixed loop until it reaches the goal s_g (line 3). At each iteration of the loop, the agent expands the current state s_t by generating its immediate neighbors $N(s_t)$ (local planning). It computes its action by selecting the next state s_{t+1} among the neighbors to minimize the estimated cost of traveling to the goal through that neighbor (line 4). Ties among neighbors that have the same $c + h$ values are broken with a tie-breaking schema that is consistent over state revisits. Then, in line 5, the agent updates (learns) its heuristic in the current state from $h_t(s_t)$ to $h_{t+1}(s_t)$. Note that the explicit maximum of the state's old heuristic value and the new value causes the heuristic to never decrease. Such a maximum is unnecessary if the heuristic is consistent and is commonly omitted in the literature. We do not assume our heuristic to be consistent and hence put the maximum in explicitly. The agent then changes its current state to the neighbor and the cycle repeats.

4 Building Blocks

To define the space of real-time heuristic search algorithms, we first abstract the base algorithm (Algorithm 1) into a search algorithm template (Algorithm 2). The template still has the main loop (line 3) which the agent executes until it gets to the goal state. Within the loop the agent repeatedly executes the movement (line 4) and the learning (line 5) rules. The rules can include the following building blocks.

Algorithm 2: Search Algorithm Template

input : search problem (S, E, c, s_0, s_g, h)
output: path $(s_0, s_1, \ldots, s_T)$ such that $s_T = s_g$

1 $t \leftarrow 0$
2 $h_t \leftarrow h$
3 **while** $s_t \neq s_g$ **do**
4 $\quad s_{t+1} \leftarrow$ new s due to a movement rule
5 $\quad h_{t+1}(s_t) \leftarrow \max\{h_t(s_t), \text{new } h \text{ due to a learning rule}\}$
6 $\quad t \leftarrow t + 1$
7 $T \leftarrow t$

4.1 Movement Rule Building Blocks

For the movement rule we use line 4 in the base Algorithm 1 with the following possible blocks on top:

Backtracking (Shue and Zamani; Shue and Zamani; Shue et al.; Bulitko and Lee, 1993a; 1993b; 2001; 2006) causes the agent to move back to the previous state on its path when the heuristic is updated. For simplicity we do not consider the learning quota T parameter of SLA*T and LRTS and move the agent back to the previous state as soon as the heuristic function is updated (i.e., learning takes place). The intuition of backtracking is that upon detecting an inaccurate heuristic value the agent should not only update it in the current state but also in the previous states whose heuristic values may be dependent on it.

Depression avoidance (Hernández and Baier, 2012) detects whether the current state is a part of a heuristic depression — an area of the state space where the heuristic values are inaccurately low — and tries to guide the agent out of such a depression. The problem of heuristic depressions was identified early on (Ishida, 1992) and linked to state-revisitation and increased solution cost (Huntley and Bulitko, 2013). In our building block we use the method implemented in daLRTA* by Hernández and Baier (2012). Specifically, to select a neighbor of the current state as its next state, the agent considers only the states $N_{\text{min learning}}(s_t)$ where the amount of learning to-date is minimal:

$$\begin{aligned} N_{\text{min learning}}(s_t) &= \{s \in N(s_t) \mid |h_0(s) - h_t(s)| = \mu\} \\ \mu &= \min_{s \in N(s_t)} |h_0(s) - h_t(s)|. \end{aligned} \tag{1}$$

Then the agent selects its next state s_t from the set $N_{\text{min learning}}(s_t)$ in the usual fashion (i.e., the one that minimizes $c(s_t, s) + h_t(s)$ in line 4, Algorithm 1).

Removing expendable states (Sharon et al., 2013) is meant to reduce the size of the search space by eliminating a state whose immediate neighbors can be all reached from each other within the immediate neighborhood. Such states are called locally expendable. In line with Sharon et al. (2013) we remove a locally expendable state from the search graph only when the heuristic of the state is updated by the agent.

4.2 Learning Rule Building Blocks

For the learning rule, we extend the basic mini-min update of LRTA* (line 5 in the base Algorithm 1) with the following

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/108/1904158/978-0-262-33936-0-ch024.pdf by guest on 20 August 2022

possible blocks:

Heuristic weighting is meant to accelerate the learning process and thus discourage the agent from re-visiting states (a major problem with heuristic depressions). We modify the weighting rule of Rivera et al. (2015):

$$h_{t+1}(s_t) \leftarrow \max\left\{h_t(s_t),\ \min_{s\in N(s_t)}\left(w\cdot c(s_t,s)+h_t(s)\right)\right\} \quad (2)$$

by moving the weight outside of the $c+h$:

$$h_{t+1}(s_t) \leftarrow \max\left\{h_t(s_t),\ w\cdot\min_{s\in N(s_t)}\left(c(s_t,s)+h_t(s)\right)\right\} \quad (3)$$

which further increases the updates to the heuristic of the current state. We do so as our preliminary tests have shown the modified rule to be more robust and thus easier to tune the weight for. Note that $w \geq 1$ means that the heuristic may become inadmissible and inconsistent.

Learning operators other than the $\min$ can be used in the learning rule (3) if admissibility/consistency is not a requirement. We allow our agents to replace the $\min$ with avg, median or $\max$.

Lateral learning allows the learning operator ($\min$, avg, median, $\max$) in the learning update rule (3) to be taken over a portion of the neighborhood $N(s_t)$. Exploratory experiments have shown that doing so can improve the agent's performance. We define the portion using a parameter b called *beam width*. Specifically, the partial neighborhood N_b^f of a state s_t is defined as the b fraction of the neighborhood $N(s_t)$ with the lowest f values:

$$N_b^f(s) = \left(s^1,\ldots,s^{\lfloor b|N(s_t)|\rfloor}\right) \quad (4)$$

where $\left(s^1,\ldots,s^{\lfloor b|N(s_t)|\rfloor},\ldots,s^{|N(s_t)|}\right)$ is the immediate neighborhood sorted in the ascending order by their $f=c+h$. For instance, s^1 has the lowest $f(s^1)=c(s_t,s^1)+h(s^1)$ value in the set $\{f(s)\mid s\in N(s_t)\}$ whereas $s^{|N(s_t)|}$ has the highest f value in that set. Clearly, for $b=1$ we get the full neighborhood: $N_1^f(s_t)=N(s_t)$. For $b=0$ we define $N_0^f(s_t)$ as the neighbor with the lowest f: $\{s^1\}$.

4.3 Putting the Building Blocks Together

With these building blocks, the template Algorithm 2 becomes Algorithm 3. The main loop is the same as before (line 3). Inclusion of the building blocks is determined by the control parameters w, b, lop, da, expendable, backtrack as follows. If the depression avoidance block is present in the agent (da $= true$) then line 5 temporarily sets the neighborhood to only the states where the amount of learning $|h_t(s)-h_0(s)|$ is minimal. The learning rule in line 6 covers heuristic weighting, learning operator and lateral learning, using the control parameters w, lop and b.

If the expendable block is present in the agent (expendable $= true$) then in line 8 the current state is removed from the graph if there was learning in it and it is indeed locally expendable (denoted by the predicate $\mathcal{E}$). Finally, if the agent learned in the current state and the backtracking block is present (backtrack $= true$) then the agent will move back to the previous state in line 10. Otherwise it moves forward in line 12. If there is no previous state (i.e., $s_t = s_0$) then the agent stays put.

If at any time the neighborhood $N(s_t)$ becomes empty (i.e., the agent has no moves to pick from) then the agent quits without producing a solution. Such unsolved problems contribute $\alpha_{\max}h^*(s_0)+\min_{s_a,s_b\in S}c(s_a,s_b)$ to the agent's statistics on the solution cost/suboptimality.

In the rest of the paper we compactly denote any such algorithm by listing its building blocks as $w\cdot\text{lop}_b(c+h)$+backtrack+da+E where the last three parts are optional.

Algorithm 3: Real-time Heuristic Search w/ Building Blocks

input : search problem (S,E,c,s_0,s_g,h), control parameters w, b, lop, da, expendable, backtrack
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 $N(s_t)\leftarrow N_{\text{min learning}}(s_t)$
6 $h_{t+1}(s_t)\leftarrow \max\left\{h_t(s_t),\ w\cdot\operatorname*{lop}_{s\in N_b^f(s_t)}\left(c(s_t,s)+h_t(s)\right)\right\}$
7 **if** expendable & $h_{t+1}(s_t)>h_t(s_t)$ & $\mathcal{E}(s_t)$ **then**
8 remove s_t from the search graph
9 **if** backtrack & $h_{t+1}(s_t)>h_t(s_t)$ **then**
10 $s_{t+1}\leftarrow s_{t-1}$
11 **else**
12 $s_{t+1}\leftarrow\arg\min_{s\in N(s_t)}\left(c(s_t,s)+h_t(s)\right)$
13 $t\leftarrow t+1$
14 $T\leftarrow t$

5 Simulated Evolution

We had briefly experimented with an ERL-style asynchronous evolution (Ackley and Littman, 1991) without an explicit fitness function or generations. However, given our current heuristic search code base, it was computationally prohibitive and we switched to the traditional style evolution with discrete generations and an explicit fitness function. Section 7 briefly discusses the alternative.

The simulated evolution proceeds as per Algorithm 4. It starts in line 2 with forming the initial population P_0 of K agents. Each agent is represented by its gene which encodes the building blocks used by the agent. Technically the gene is a vector $(w,b,\text{lop},\text{da},\text{expendable},\text{backtrack})$. Each of six components of the initial agents' genes is picked

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/108/1904158/978-0-262-33936-0-ch024.pdf by guest on 20 August 2022

Algorithm 4: Evolution of Search Agents

input : search problems, batch size B, max suboptimality $\alpha_{\max}$, population size K, number of generations M
output: genome of p_{oldest}

1 $t \leftarrow 0$
2 create population P_0 of size K with random genes
3 **for** $t = 1, \ldots, M$ **do**
4 **for** $p \in P_{t-1}$ **do**
5 $\phi(p) \leftarrow \alpha(p)$ over B problems, truncated at $\alpha_{\max}$
6 sort P_{t-1} by ϕ
7 $C \leftarrow \text{children}(P_{t-1}(1, \ldots, K/2))$
8 $P_t \leftarrow P_{t-1}(1, \ldots, K/2) \cup C$
9 update p_{oldest}

uniformly randomly from their respective ranges. Some of the ranges are domain specific and we will list them in Section 6.2. Binary gene components (e.g., $\mathtt{da}$ which takes on the values of $true$ and $false$) are represented as a continuous value in $[0, 1]$. The algorithm converts it to binary by rounding. For instance, an agent with the $\mathtt{da}$ gene of 0.4 will not have depression avoidance block as $\text{round}(0.4) = 0$. However, an agent with the $\mathtt{da}$ gene of 0.7 will perform depression avoidance as $\text{round}(0.7) = 1$. The same schema is used for the learning operator lop gene which takes on a continuous value in $[1, 4]$ but is rounded to the nearest discrete value inside the search agent to select the operator ($1, 2, 3, 4$ encode min, avg, median and $\max$ correspondingly).

During each of the M generations each agent of the population is evaluated on B problems picked randomly from search problems given to the evolution. The fitness of the individual p is denoted by $\phi(p)$ and is the average suboptimality of the agent p on the B problems (line 5). Each run is truncated if the agent exceeds the suboptimality $\alpha_{\max}$ as explained earlier in the paper.

The population is then sorted by the agents' fitness in line 6. The top half of the population, $P_{t-1}(1, \ldots, K/2)$, are included in the next generation's population P_t and are allowed to reproduce. The bottom half are removed and are replaced by $K/2$ children in line 8.

The set of children C is created by randomly picking two parents from the top half of P_{t-1} and performing a cross over on their genes in line 7. In other words, each gene value of the child has a $50/50\%$ chance of coming from the father or the mother. The child's gene values are then mutated by adding a Gaussian noise of zero mean and the standard deviation of $1/100$th of the size of the gene range (e.g., $(4 - 1)/100 = 0.03$ for the lop gene). If adding Gaussian noise pushes the gene value outside of the valid range then the value is clipped. For instance, if a child inherited $\text{lop} = 4$ from its parent but the mutation noise made it $4 + 0.03 = 4.03$ then it will be brought back down to 4.

Finally, line 9 updates the running oldest agent p_{oldest} (i.e., the one who has survived the most generations so far) with an older one, if such is found in the current generation. The genome of the earliest oldest agent is the output of the evolution. Ties between agents of the same age are broken in the favor of lower suboptimality.

6 Empirical Evaluation

The traditional testbed for real-time heuristic search is pathfinding in video games, where the planning time per move is limited to a few milliseconds for *all* simultaneously operating agents. The standard way (Sturtevant, 2012) of representing game maps is as a two-dimensional discrete grid where each grid cell is either available for the agent to pass through (i.e., vacant, shown white in Figure 1) or blocked by an obstacle (black). At each moment of time, the agent occupies a single vacant cell of the map which determines the agent's current state. The agent changes its state by moving from its current grid cell to one of its vacant neighbors, incurring a travel cost. In this paper we use the standard eight-connected maps where cardinal moves cost 1 and diagonal moves cost $\sqrt{2}$. A problem is solved when the agent enters the goal cell. At the beginning of each problem, the agent starts with the octile distance as the heuristic. Octile distance is the cost of the shortest path between a given cell and the goal cell *if* no cells are blocked by obstacles.

Figure 1: A video-game map from *Dragon Age: Origins*.

We implemented all algorithms in a mixture of C and MATLAB code run on Intel i5/i7-based desktop computers. Parts of the code were run in parallel on 4-6 CPU cores.

6.1 Problem Set

We used the benchmark problems from the Moving AI set (Sturtevant, 2012). We treated the water terrain type as obstacle and excluded all problems which thereby became unsolvable (e.g., the start state is in an obstacle cell). This resulted in 493298 problems situated on 342 maps. The maps were from the video games *StarCraft*, *WarCraft III*, *Baldur's*

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/108/1904158/978-0-262-33936-0-ch024.pdf by guest on 20 August 2022

Gate II (maps scaled up to 512×512) and *Dragon Age: Origins* (Figure 1).

6.2 Evolution

We conducted three evolution runs. The first run was of 50 generations each of 200 agents. Each agent of each generation was evaluated on 200 random problems. The maximum suboptimality $\alpha_{\max}$ was set to 1000. The maximum age of any agent was 14 generations and it was first achieved in generation 47 by the agent $8.223 \cdot \min_{0.341}(c+h)$+E. The run took approximately 6 hours, with the agents solving 92 problems/second.

The second run was of 25 generations each of 200 agents. Each agent of each generation was evaluated on 400 random problems. The maximum suboptimality $\alpha_{\max}$ was set to 1000. The maximum age of any agent was 11 generations and it was first achieved in generation 12 by the agent $7.952 \cdot \min_{0.720}(c+h)$+da+E. The run took approximately 5 hours, with the agents solving 110 problems/second.

The third run was of 200 generations each of 100 agents. Each agent of each generation was evaluated on 100 random problems. The maximum suboptimality $\alpha_{\max}$ was set to 1000. The maximum age of any agent was 16 generations and it was first achieved in generation 72 by the agent $8.061 \cdot \text{avg}_{0.029}(c+h)$+E. The run took approximately 9 hours, with the agents solving 63 problems/second.

We evaluated the resulting algorithms on non-overlapping sets of 10000 random problems. The suboptimality cutoff was set to $\alpha_{\max} = 10^5$ which allowed all of them to solve all problems. The results are found in Table 1 and suggest that the evolved individuals are similar. For the subsequent evaluation we picked $8.223 \cdot \min_{0.341}(c+h)$+E algorithm.

Table 1: Results of the three evolution runs. Means and standard errors of the mean are listed.

Algorithm	Suboptimality α	Scrubbing τ
$8.223 \cdot \min_{0.341}(c+h)$+E	20.70 ± 0.4782	1.17 ± 0.0036
$7.952 \cdot \min_{0.720}(c+h)$+da+E	21.17 ± 0.5220	1.18 ± 0.0037
$8.061 \cdot \text{avg}_{0.029}(c+h)$+E	21.35 ± 0.5366	1.18 ± 0.0038

6.3 Systematic Search

We also ran systematic search in the space of the possible algorithms. To do so, we tabulated the weight space w in 15 increments from 1 to 10. We tabulated the beam width b in 15 increments from 0 to 1. For each of those combinations we tried all four learning rule operators ($\min$, avg, median and $\max$) and all three movement rule blocks: depression avoidance, backtracking and removing expendable states. In total we created $15 \times 15 \times 4 \times 2 \times 2 \times 2 = 7200$ agents. Each of them was run on the same 100 random problems. The three agents with the lowest suboptimality were: $\max_{0.714}(c+h)$+E, $\max_{0.143}(c+h)$+da+E and $2.286 \cdot \text{avg}_{0.286}(c+h)$+da+E. Their suboptimalities over 100 problems were 14.56, 14.87 and 15.14 respectively.

As before, we evaluated the resulting algorithms on non-overlapping sets of 10000 random problems. The suboptimality cutoff was again set to $\alpha_{\max} = 10^5$ which allowed all three algorithms to solve all problems. The results are found in Table 2 and suggest that the evaluation on the 100 problems during the systematic search is not representative of the algorithms' true performance. For instance, the second algorithm was selected for its second-best suboptimality of 14.87 as measured on the 100 problems. However, as measured on 10000 problems, it performed substantially worse (suboptimality of 30.44).

Table 2: Results of the three evolution runs. Means and standard errors of the mean are listed.

Algorithm	Suboptimality α	Scrubbing τ
$\max_{0.714}(c+h)$+E	19.56 ± 0.5237	1.16 ± 0.0035
$\max_{0.143}(c+h)$+da+E	30.44 ± 1.4305	1.22 ± 0.0075
$2.286 \cdot \text{avg}_{0.286}(c+h)$+da+E	20.47 ± 0.5109	1.13 ± 0.0026

Such misleading estimates of the algorithm's performance appear a necessary downside of the systematic search. Indeed, to evaluate all 7200 algorithms in a reasonable amount of time, each can be evaluated only on a small set of problems. In contrast, our evolution looks for the oldest individuals which had to remain in the top half of the performance in *each* generation to make it to the next. For instance, the best algorithm found by the evolution, $8.223 \cdot \min_{0.341}(c+h)$+E, lasted for 14 generations and thus had its performance evaluated on 14×200 problems.

For the subsequent evaluation we picked $\max_{0.714}(c+h)$+E as the output of the systematic search.

6.4 Competing Algorithms

We selected two groups of published algorithms as competitors. In the first group, we have LRTA* (Korf, 1990) and its weighted variant wLRTA* (Rivera et al., 2015). To select the weight in wLRTA*, we ran $w \in \{1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\}$ on 6000 non-overlapping problems each and found that $w = 128$ gave the lowest suboptimality.

In the second group we have existing algorithms that explicitly aim to escape heuristic depressions quicker than LRTA*. Specifically, we evaluated the myopic versions (lookahead of 1) of aLRTA*, daLRTA* (Hernández and Baier, 2012), its weighted version w-daLRTA* (Rivera et al., 2015) and its combination with removing expendable states, daLRTA*+E (Sharon et al., 2013). We did not include f-LRTA* or f-LRTA*+E in our evaluation since both versions were found inferior to daLRTA* by Sharon et al. (2013). To select the weight parameter for w-daLRTA*, we ran $w \in \{1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\}$ on 6000 non-overlapping problems each and found that $w = 7$ gave the lowest suboptimality.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/108/1904158/978-0-262-33936-0-ch024.pdf by guest on 20 August 2022

6.5 Competition Results

We ran the eight algorithms, each on 30000 non-overlapping problems randomly selected from the benchmark set. For each algorithm we measured its suboptimality α and scrubbing τ. The suboptimality cutoff was set to $10^5 \cdot h^*(s_0)$ and all algorithms solved all problems under the cutoff. For the systematic search we used algorithm $\max_{0.714}(c+h)$+E. For the evolution we used $8.223 \cdot \min_{0.341}(c+h)$+E. The means and standard errors of the mean are listed in Table 3.

Table 3: Performance of the algorithms: sample mean $\pm$ standard error of the mean.

Algorithm	Suboptimality α	Scrubbing τ
LRTA*	455.58 ± 7.2984	16.24 ± 0.1423
aLRTA*	349.21 ± 6.3649	11.04 ± 0.1175
daLRTA*	44.98 ± 0.7425	1.98 ± 0.0104
wLRTA*	39.81 ± 0.5458	1.83 ± 0.0078
wdaLRTA*	30.95 ± 0.4521	1.48 ± 0.0050
daLRTA*+E	31.82 ± 0.7782	1.21 ± 0.0041
systematic search	19.75 ± 0.2836	1.17 ± 0.0020
evolution	20.79 ± 0.2809	1.17 ± 0.0021

The new algorithms found by evolution and systematic search appear to produce shorter solutions and scrub less than the classical and contemporary competing algorithms.

Compared to the existing algorithms, the new algorithms use a combination of more aggressive learning rules, $\max_{0.714}(c+h)$ or $8.223 \cdot \min_{0.341}(c+h)$, with marking expendable states. Naturally, either of these combinations could have been found manually by the researchers but since it is time-consuming and tedious to try all combinations of building blocks, researchers tend to focus on a few, guided by their experience and intuition. Evolution and systematic search lack either and operate entirely based on algorithm performance (within the human-defined algorithm space).

7 Future Work

The promising preliminary results reported above open a few avenues for future work. First, while we tested the evolved algorithms on problems that they may not have seen during the evolution process, it is still possible that the resulting algorithms work so well merely because they overfit to the video-game pathfinding maps. Thus, the next step is to explore how effective the process is in other real-time heuristic search domains and across domains.

Second, additional real-time heuristic search techniques such as deeper lookahead of LSS-LRTA* (Koenig and Sun, 2009) can be made available to the evolution as building blocks. To make the competition fair, additional planning time per move should be incorporated into the fitness function. Another possibility to curb computational complexity is to assume that computations are subject to errors. For instance, one can introduce access errors to the heuristic which is realistic when heuristic values are stored in the environment (a la pheromone in ant-colony algorithms) and the agents may not always read/write them perfectly. Then robust algorithms are likely to evolve (Ackley, 2013).

Finally, we can attempt to evolve robust search agents specifically for a class of problems. Following Ackley and Small (2014) we can have the agents continuously evolving in an asynchronous environment, similarly to the ERL testbed of Ackley and Littman (1991). In such a setting there would be no discrete generations and no explicit fitness function. Instead, each agent will maintain a health value which is depleted during the search and replenished when the agent reaches its goal, before taking on the next search problem. An agent which lived long enough and healthy enough is given an opportunity to reproduce. The domain-specificity would allow us not only to evolve the algorithms but also their innate heuristic function which can be encoded in the genes using function approximation.

8 Conclusions

We presented possibly the first application of evolutionary search to real-time heuristic search algorithms. We did so by breaking down classical and contemporary real-time heuristic search algorithms into building blocks for the evolution to pick from. In a large-scale evaluation in video-game pathfinding, evolved real-time heuristic search agents outperformed manually designed published algorithms.

Acknowledgments

We appreciate discussions with Valeriy Bulitko and feedback from the conference reviewers. The idea of lateral learning was inspired by bitpattern heuristic update rules of Alexander Sampley.

References

Ackley, D. and Littman, M. (1991). Interactions between learning and evolution. *Artificial life II*, 10:487–509.

Ackley, D. H. (2013). Beyond efficiency. *Communations of the ACM*, 56(10):38–40.

Ackley, D. H. and Small, T. R. (2014). Indefinitely scalable computing = artificial life engineering. In *Proceedings of The Fourteenth International Conference on the Synthesis and Simulation of Living Systems (ALIFE 14) 2014*, pages 606–613. MIT Press.

Botea, A. (2011). Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 122–127.

Bulitko, V. (2004). Learning for adaptive real-time search. *CoRR*, cs.AI/0407016.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/108/1904158/978-0-262-33936-0-ch024.pdf by guest on 20 August 2022

Bulitko, V., Björnsson, Y., and Lawrence, R. (2010). Case-based subgoaling in real-time heuristic search for video game pathfinding. *Journal of Artificial Intelligence Research*, 39:269–300.

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

Bulitko, V., Luštrek, M., Schaeffer, J., Björnsson, Y., and Sigmundarson, S. (2008). Dynamic control in real-time heuristic search. *Journal of Artificial Intelligence Research*, 32:419–452.

Bulitko, V., Rayner, D. C., and Lawrence, R. (2012). On case base formation in real-time heuristic search. In *Proceedings of Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 106–111.

Bulitko, V., Sturtevant, N., Lu, J., and Yau, T. (2007). Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research*, 30:51–100.

Hart, P., Nilsson, N., and 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.

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

Hernández, C. and Meseguer, P. (2005). LRTA*(k). In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1238–1243.

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

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

Koenig, S. (2001). Agent-centered search. *Artificial Intelligence Magazine*, 22(4):109–132.

Koenig, S. and 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.

Lawrence, R. and Bulitko, 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.

Rayner, D. C., Davison, K., Bulitko, V., Anderson, K., and Lu, J. (2007). Real-time heuristic search with a priority queue. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 2372–2377.

Rivera, N., Baier, J. A., and Hernández, 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*, pages 167–174.

Shue, L.-Y., Li, S.-T., and Zamani, R. (2001). An intelligent heuristic algorithm for project scheduling problems. In *Annual Meeting of the Decision Sciences Institute*.

Shue, L.-Y. and Zamani, R. (1993a). An admissible heuristic search algorithm. In *International Symposium on Methodologies for Intelligent Systems (ISMIS-93)*, volume 689 of *LNAI*, pages 69–75.

Shue, L.-Y. and Zamani, R. (1993b). A heuristic search algorithm with learning capability. In *ACME Transactions*, pages 233–236.

Sturtevant, N. (2007). Memory-efficient abstractions for pathfinding. In *Proceedings of Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 31–36.

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

Sturtevant, N. R. and Bulitko, V. (2011). Learning where you are going and from whence you came: H- and G-cost learning in real-time heuristic search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 365–370.

Sturtevant, N. R. and Bulitko, V. (2014). Reaching the goal in real-time heuristic search: Scrubbing behavior is unavoidable. In *Proceedings of the Symposium on Combinatorial Search*, pages 166–174.

Sturtevant, N. R., Bulitko, V., and Björnsson, Y. (2010). On learning in agent-centered search. In *Proceedings of the Conference on Autonomous Agents and Multiagent Systems*, pages 333–340.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/108/1904158/978-0-262-33936-0-ch024.pdf by guest on 20 August 2022