# Searching for Real-time Heuristic Search Agents using Asynchronous Evolution

**Toukir Imam**
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
mdtoukir@ualberta.ca

**Abstract**

Path finding is a very important topic with many real life applications. In many situations path finding needs to be done in real time. In real time path finding, the agent needs to make every move under a time constraint. Real time heuristic search algorithms are a popular approach towards solving real time path finding problems. Over the past decade, many algorithms in this field have been developed. One way to find better algorithm is to combine these existing algorithms in an optimal manner. Previous works have used synchronous evolution algorithm to find the optimal combination of these algorithms and their parameters. In this paper, we propose an asynchronous evolution algorithm to find the optimal combination of existing search algorithms and their parameters. Our approach yielded a search algorithm that performs similar to the best search algorithm found by the synchronous evolution.

## 1 Introduction and Related Work

The field of real time heuristic search algorithm is an active one where many different algorithms have been proposed over the past decade. One of the most important among them is Korf's LRTA* (Korf 1990). Because of its robustness and adaptability, it has been extended on and incorporated in many other algorithms since then. Notable extensions of LRTA* include weighted LRTA* (Rivera, Baier, and Hernández 2015), depression avoidance (Hernández and Baier 2012), expendable states (Sharon, Sturtevant, and Felner 2013), and so on. However each of these extensions introduces one or more parameters that need tuning.

An early attempt at combining the different extensions of LRTA* was done by (Bulitko and Lee 2006) which combined backtracking, heuristic weight, and lookahead into one algorithm. However, combining multiple extensions means having multiple parameters that need tuning. This creates a problem where, from many different possible combinations of parameter values, we need to find the search algorithm with the optimal parameters.

(Bulitko 2016b) and (Bulitko 2016a) are two attempts at solving this problem and finding the best search algorithm.



In (Bulitko 2016a), the authors combined backtracking, depression avoidance, expandable states removal, heuristic weighting, and lateral learning into a single algorithm. The resulting algorithm had 6 parameters that needed tuning. The paper proposed a synchronous evolutionary algorithm in an Artificial Life (ALife) setting to find the optimal combination. The best algorithm found by this search was 23 times faster than the LRTA* algorithm. This work was extended in (Bulitko 2016b) to include tabulated search.

ALife has a rich history in studies of autonomous agents due to the intuitive setting it provides. In ALife, time can be modeled in two different ways, synchronously and asynchronously. In synchronous update scheme, time is treated as discrete and all agents are updated together. As a result in synchronous evolution agents from one generation can not interact with agents from another generation. On the other hand, In asynchronous update scheme, agents update independent of each other and agents from one generation can interact with agents from another generation.

A notable use of the asynchronous algorithm is (Ackley and Littman 1991), which used asynchronous evolution to study the effects of learning on evolving autonomous agents. While this work is one of the inspirations behind (Bulitko 2016a) , In (Bulitko 2016a) synchronous evolution is used.

However the ability of inter generation interaction in asynchronous evolution can produce genes that would not be possible in synchronous evolution. Therefore, in this paper, we propose an asynchronous evolution algorithm to find the optimal search algorithm.

The rest of the paper is organized as follows. Section 2 formally defines the problem we are attempting to solve. An extended LRTA* algorithm and its building blocks are analyzed in Section 3. The proposed asynchronous evolution algorithm is described in section 4. Empirical evaluation and results are presented in section 5. Finally discussion and possible future works are given in section 6 and conclude our paper in section 7.

## 2 Problem Formulation

Our algorithm is developed for path finding in Graphs. The graph, $G$ is defined by the tuple $(S, E)$. Here $S$ is the set

of all states in the graph and $E$ is the set of all edges. $G$ is assumed to be undirected and without any self loop. The function $c(s_a, s_b)$ is the cost function which returns the cost of the edge between the two states $s_a$ and $s_b$, where $s_a$ and $s_b$ are in $S$ and there is an edge between them. The cost function is always positive and symmetric. This means, an agent can move from state $s_a$ to $s_b$ or vice versa and incur a cost of $c(s_a, s_b)$. Two states are neighbors if there is an edge between them.

Given two states $s_0$ and $s_g$ in $S$, the path between them is a sequence of states $P = (s_0, s_1, s_2, ....s_g)$ where two consecutive states in $P$ has an edge between them.

Therefore, we can define a search problem in the graph $G$, between two states, $s_0$ and $s_g$ can be defined using the tuple $(S, E, c, s_0, s_g, h)$. Here $h$ is the initial heuristic of the graph.

Any searching algorithm $A$ that solves this problem returns a path, $(s_0, s_1, s_2, ....s_g)$, between $s_0$ and $s_g$. The cost of this path is $c_A(s_0, s_g) = \sum_{i=0}^{i=g-1} c(s_i, s_{i+1})$. If the lowest possible cost of a path between $s_0$ and $s_g$ is $h^*(s_0, s_g)$, then the efficiency of the algorithm, $A$ on this problem is,

$$\sigma_A = \frac{c_A}{h^*(s_0, s_g)}$$

$\sigma_A$ is called the suboptimality of algorithm $A$.

Lower suboptimality is better. The lowest possible suboptimality is 1, which would mean the solution is the optimal solution.

## 3 Extended LRTA* with Building Blocks

Algorithm 1 is a modified version of the real time heuristic search with building blocks provided in (Bulitko 2016a), which itself is an extension of the basic LRTA* algorithm from (Korf 1990). This algorithm includes 6 control parameters.

Parameters $wh$, $wc$, $b$, $lop$ control how the new heuristic is learned.

Parameter $wh$ and $wc$ are the heuristic weights (Rivera, Baier, and Hernández 2015) which speed up the learning process and minimize re-visiting of states. We used two separate weights for $c$ and $h$ (Algorithm 1 line 7).

Parameter $lop$ is the learning operator. In addition to the $min$ operator, which is used in (Korf 1990), we allow for using $max$, $avg$, and $median$ operators.

The learning operator can be applied over a single state $s_t$ or over a portion of the neighborhood of $s_t$. The parameter $b$ or *beam width* determines over how much of the neighborhood the learning operator is applied to. The range for $b$ is $[0, 1]$

Parameters $da$ and $expendables$ control the movement of the agent.

$da$ enables or disables the depression avoidance. Depression avoidance is a method of determining if the current state has inaccurately low heuristic and guiding the agent away from there. The depression avoidance described in algorithm 1 is based on the work of (Hernández and Baier 2012).

The parameter $expendable$ enables or disables removing locally expendable states from the search space. Locally expendable states are defined as states whose neighbors can be all reached from each other within the immediate neighborhood (Sharon, Sturtevant, and Felner 2013). Locally expendable states are only removed when the heuristic of the state has been updated (Algorithm 1 line 10).

In this paper we introduced a new parameter $learningRate$ which controls the update of the heuristic. The $learningRate$ parameter determines how much confidence we have on the newly calculated heuristic. Higher $learningRate$ means more confidence on the new heuristic value. The value of $learningRate$ is a real number between 0 and 1. A $learningRate$ of 0 would mean the heuristics never updating. On the other hand, a $learningRate$ of 1 would mean replacing the old heuristic completely by the new one.

Based on these parameters, we can express a search algorithm compactly using the form $lop_b(wc \cdot c + wh \cdot h) + learningRate + da + E$ where the last two are optional.

**Algorithm 1:** Search Algorithm With Building Blocks

**input** : search problem $(S, E, c, s_0, s_g, h)$,control parameters $wc, wh, b, lop, da, expendable, learningRate$
**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
7  $h_{new}(s_t) \leftarrow \max\{h_t(s_t), \underset{s \in N_b^f}{lop}(s_t)(wc \cdot c(s_t, s) + wh \cdot h_t(s))\}$
8  $h_{t+1}(s_t) \leftarrow learningRate \cdot h_{new}(s_t) + (1 - learningRate) \cdot h_t(s_t)$
9
10  **if** $expendable$ & $h_{t+1}(s_t) > h_t(s_t)$ & $\epsilon(s_t)$ **then**
11  remove $s_t$ from the search graph
12  $s_{t+1} \leftarrow \arg\min_{s \in N(s_t)}(c(s_t, s) + h_t(s))$

## 4 Asynchronous Evolution

Our asynchronous evolution algorithm is inspired by (Ackley and Littman 1991). For the asynchronous evolution, each search algorithm, as described in section 3, is represented by an agent. In this section, the search algorithm that is encompassed by an agent is refereed to as the agent's gene.

Algorithm 2 describes the main part of the asynchronous algorithm. Line 1 to 5 in Algorithm 2 creates the initial agent population by creating $K$ random genes and assigning each to an agent. Algorithm 3 describes how each agent is assigned a search problem. When a search problem is assigned to an agent, the agent is also given an energy value. This energy value is calculated using one of the energy

models described in section 4.1.

Line 7-23 of algorithm 2 is the main loop, which continues as long as the agent population is not zero. Line 12 of algorithm 2 advances all agents by one step. The $runStep()$ function is a variation of algorithm 1 which is modified to run for only one step. The $runStep()$ function also deducts the cost incurred by the agent to take that step from the agent's energy.

Line 13-14 of algorithm 2 removes from the population any agent that has spent all of its energy. This ensures the removal of unfit genes.

For those agents that have reached the goal state, two offspring are created. The first one is created using the recombination algorithm described in algorithm 4. For recombination, the partner agent is chosen by searching for agents with the most similar energy value. The first of the two mutation rate control parameters ($mR1$) is used for this offspring.

The second offspring is created without any recombination. The second mutation rate control parameter ($mR2$) is used for this offspring.

Line 21-23 of algorithm 2 is used to keep the population size at a certain number by removing agents with low energy.

Finally, line 9-10 of algorithm 2 gradually decreases the energy multiplier ($eM$). This results in an environment that gets more difficult over time as agents get less and less energy to solve problems. This also ensures that at some point the main loop will stop.

Once the main loop ends. line 24 of algorithm 2 finds the agent that solved the maximum number of problems among all the agents that ever existed. The gene of this agent is the output of the asynchronous evolution algorithm.

## 4.1 Energy Model

Each agent is given a certain amount of energy to solve a problem (Line 7 of algorithm 3). We considered three different energy models to determine how much energy should an agent receive to solve a problem.

**Fixed energy model :** In fixed energy model, agents get the same amount of energy to solve any problem. The equation for the fixed energy model is :

$$energy = C \times eM$$

Here $C$ is a constant and $eM$ is the energy multiplier.

**Optimal travel cost (OTP) energy model :** In OTP energy model, the energy an agent receives is proportional to the optimal travel cost of the problem. The equation for this model is :

$$energy = h^*(s_0, s_g) \times eM$$

Line 7 of algorithm 3 shows the OTP energy model.

**A* difficulty energy model :** A* difficulty is a measurement of the difficulty of a problem with respect to the A* algorithm. The A* difficulty of a problem is defined as :

**Algorithm 2:** Asynchronous Evolution

**input** : control parameters $(K, eL, eM, dG, mR1, mR2)$
**output**: fittest gene( $gFit$)

1 initialize set of agents, $P \leftarrow \emptyset$
2 **for** $i \leftarrow 1$ **to** $K$ **do**
3  create random gene $g_i$
4  $a_i \leftarrow AssignProblem(g_i, 0, eM)$
5  $P \leftarrow P \cup \{a_i\}$
6 $step \leftarrow 0$
7 **while** $|P| \neq 0$ **do**
8  $step \leftarrow step + 1$
9  **if** $step \ \% \ eL = 0$ **then**
10  $eM \leftarrow eM - dG$
11  **for** *agent in* $P$ **do**
12  runStep(agent)
13  **if** *agent.energy* $= 0$ **then**
14  $P \leftarrow P - \{agent\}$
15  **if** *agent has reached goal state* **then**
16  $agent_2 \leftarrow$ Find from $P$ the agent with closest $energy$
17  $newAgent_1 \leftarrow Recombination(agent, agent_2, mR1, eM)$
18  $newAgent_2 \leftarrow AssignProblem(agent.gene, mR2, eM)$
19  $P \leftarrow P \cup \{newAgent_1, newAgent_2\}$
20  $agent \leftarrow AssignProblem(agent.gene, 0, eM)$
21  **if** $|P| \geq K$ **then**
22  sort $P$ by agent.energy
23  remove from $P$ first $|P| - K$ agents
24 $fitAgent \leftarrow$Among all agents ever created, find the agent that solved maximum number of problems
25 $gFit \leftarrow fitAgent.gene$

**Algorithm 3:** Assign Problem (OTP energy model)

**input** : control parameters $(gene, mR, eM)$
**output**: agent $(a)$

1 create random problem, $PROB \leftarrow (S, E, c, s_0, s_g, h)$
2 create agent $a$
3 assign problem $PROB$ to agent $a$,
4 $(a.S, a.E, a.c, a.s_0, a.s_g, a.h) \leftarrow PROB$
5 assign $gene$ to agent $a$, $a.gene \leftarrow gene$
6 mutate $a.gene$ using $mR$
7 assign energy to $a$, $a.energy \leftarrow h^*(s_0, s_g) \times eM$

**Algorithm 4:** Recombination

**input** : agent1, agent2,mutationRate $(a1, a2, mR, eM)$
**output**: agent $(a)$

1 $gene \leftarrow$ half gene from $a1$ + half gene from $a2$
2 $a \leftarrow AssignProblem(gene, mR, eM)$

$$A^*diff = \frac{\text{Number of states expanded during A* algorithm}}{\text{Number of states in the optimal path}}$$

In A* difficulty model, the amount of energy an agent receives is :

$$energy = A^*diff \times h^*(s_0, s_g) \times eM$$

## 5 Empirical Evaluation

We compared our asynchronous evolution algorithm against the synchronous evolution algorithm proposed in (Bulitko 2016a). We used total state expansion (TSE) during evolution as a measure of the scale of the evolution. The total state expansion (TSE) is the total number of states visited by the agents during evolution. Evolutions with higher TSE takes longer to finish compared to evolutions with lower TSE.

For the asynchronous evolution algorithm, the scale of the evolution, and thus TSE, can be controlled by 4 parameters, $K$, $eL$, $eM$, and $dG$. For synchronous evolution algorithm, the scale of the evolution is controlled by 3 parameters, population size, number of problems per generation, and number of generations.

### 5.1 Problem Set

We used the problem set provided in (Bulitko 2016a), which is a subset the Moving AI problem set created by (Sturtevant 2012). The problem set contains 342 maps from the games StarCraft, WarCraft III, Baldur's Gate II, and Dragon Age: Origins. As shown in figure 1, the maps are 2D grids with black squares representing obstacle and white squares representing obstacle free space. Provided there are no obstacles, an agent can move in 8 directions from any state. The cardinal moves cost the agent 1 while the diagonal moves cost $\sqrt{2}$.
For both algorithms, the evolution was carried out on a set of 34200 problems over the 342 maps. The evaluation of the searching algorithm found in the evolutionary algorithms were done on randomly selected 4293 problems from a set of unseen 493298 problems over the same 342 maps. The initial heuristic $(h)$ used is the obstacle free octile distance.

### 5.2 Results

In our first experiment, we compared the fixed, OTP, and A* difficulty energy models. For each of the energy models, the evolution was set up with $K = 20$, $eL = 1000000$, $eM = 8$, and $dG = O.3$. The experiment was then run 10 times. The results of the experiment over the 10 runs are given in table 1.

Form table 1 we can see that the A* difficulty energy model performs better than the other two in minimum, average, and maximum suboptimality. The fixed energy model performs the worst of the three models. Therefore, we took

Figure 1: Sample map from the Moving AI problem set

Table 1: Comparison of the energy models

| **Model** | **min** $\sigma$ | **avg** $\sigma$ | **max** $\sigma$ |
|---|---|---|---|
| Fixed energy | 23.23 | 36.63 | 52.84 |
| OTP energy | 20.59 | 28.51 | 45.29 |
| A* difficulty energy | 20.03 | 23.18 | 30.76 |

the A* difficulty model and compared it against the Synchronous evolution algorithm. We compared the two algorithm over different scales of the evolution. For the asynchronous evolution algorithm, we fixed $K = 20$, $eM = 8$, and $dG = 0.3$. and varied $eL$ to get evolutions of differing scale. For synchronous evolution, we fixed the population size to be 20, and varied the number of generation and number of problems per generation to get evolutions of differing scale.

Figure 2 shows the results of this experiment. the x-axis shows the total number of states expanded during evolution and the y-xis is the average suboptimality. From the graph we can see that the synchronous evolution algorithm outperforms the asynchronous algorithm for every scale of the evolution. Moreover the synchronous evolution algorithm converges towards suboptimality 19 as the evolution is scaled up. However the asynchronous algorithm fails to converge towards any value.

The best algorithm found by the asynchronous evolution have a suboptimality of **19.75** achieved by the gene $median_{0.95}(4.37c+6.23h)+0.04+E$ over the entire Moving AI dataset. Compared to this the best algorithm found by (Bulitko 2016a) have a suboptimality of **19.53** over the entire Moving AI dataset.

## 6 Discussion and Future Works

While the results show that the A* difficulty energy model is better than the other two, it clearly shows that the asynchronous evolution is outperformed by the synchronous evolution in terms of average suboptimality. The reason for this could be that while the synchronous evolution penalizes the

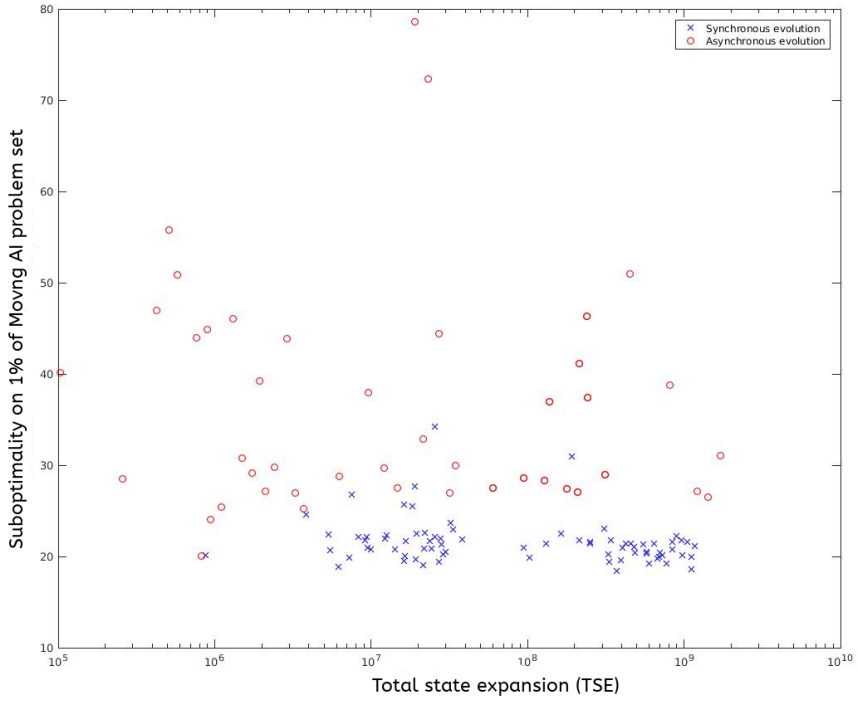


Figure 2: Asynchronous (A* difficulty energy model) vs synchronous Evaluation

searching algorithm for having bad average suboptimality directly, the asynchronous evolution does so only indirectly. One way to confirm this would be to change the fitness function so that they are equivalent in both algorithms. This could be one of the possible future works.

Moreover, there are several ways to try to improve the performance of the asynchronous evolution. A dynamically adjustable birth rate could control the population size by adjusting the birth rate of new agents with the death rate. This will result in less population culling and may improve results.

Another possible improvement could be using a different energy model. From table 1 we can see that the energy model has significant impact on the performance of the evolutionary algorithm. While the A* difficulty model performs the best, the relation between the difficulty measure of an A* agent and a LRTA* agent is not clear. A better energy model may be proposed that better reflects the difficulty of a LRTA* agent. Finally we can add more building blocks to our search algorithm. Adding lookahead (Koenig and Sun 2009) can significantly improve the performance of the search algorithm.

## 7 Conclusions

In conclusion, we can say that, while the asynchronous evolutionary algorithm is outperformed by the synchronous evolutionary algorithm, we can still learn a lot about the nature of search algorithms and the nature of evolutionary algorithms by studying the reasons behind their difference in performance. Further studies should be carried out to analyze and improve the asynchronous evolutionary algorithm.

## References

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

Bulitko, V., and Lee, G. 2006. Learning in real time search: A unifying framework. *JAIR* 25:119–157.

Bulitko, V. 2016a. Evolving real-time heuristic search algorithms. In *ALIFEXV*, (in press).

Bulitko, V. 2016b. Searching for real-time search algorithms. In *SoCS*, (in press).

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

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

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

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

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

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *IEEE TCIAIG* 4(2):144 – 148.