PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

WEIGHTED REAL-TIME HEURISTIC SEARCH

NICOLÁS RIVERA ABURTO

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JORGE A. BAIER A.

Santiago de Chile, August, 2013

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

WEIGHTED REAL-TIME HEURISTIC SEARCH

NICOLÁS RIVERA ABURTO

Members of the Committee:
JORGE A. BAIER A.
MARCELO ARENAS S.
CARLOS HERNÁNDEZ U.
PABLO PASTÉN G.

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, August, 2013

To my parents.

ACKNOWLEDGEMENTS

First and foremost I wish to acknowledge my supervisor, Jorge Baier. His help, patience and knowledge was fundamental for the development of this work. Without him, this thesis would have not been finished.

I wish to express my gratitude to the member of my thesis committee: Carlos Hernández by his helpfull ideas (one of these is the core of the thesis), to Marcelo because he rocks, and specially to Pablo Pastén for making the process of defense extreme fast and smooth.

Also I wish to thank my friends at DCC: Gabo, Gonzalo, León, Lete, Martín and Pame. Without them my thesis would have been completed in three months.

I would like to express my gratitude to my girlfriend Tamara and our endless-work afternoons and finally I wish to thank my parents and my brother who always believe in me.

Contents

List of Figures

ABSTRACT

Multiplying the heuristic function by a weight greater than one is a well-known technique in Heuristic Search. When applied to A* with an admissible heuristic it yields substantial runtime savings, at the expense of sacrificing solution optimality. Only a few works have studied the applicability of this technique to Real-Time Heuristic Search, a search approach that builds upon Heuristic Search. In this thesis we present two novel approaches to using weights in Real-Time Heuristic Search. The first one, *weighted lookahead*, is a variant of an existing approach by Shimbo and Ishida. It incorporates weights to the lookahead search phase of the Real-Time Heuristic Search algorithm. The second one, *weighted update*, incorporates the weight to the edges of the search graph during the *learning* phase. Both techniques are applicable to a wide class of Real-Time Heuristic Search algorithms. Here we implement them within LSS-LRTA*, obtaining two new algorithms. We evaluate them in path-planning benchmarks and show that weighted lookahead yields poor results but nevertheless outperforms Shimbo and Ishida's approach. Weighted update, on the other hand, yields performance improvements of up to one order of magnitude both in solution cost and total search time. To illustrate further the generality of weighted update, we incorporate the technique in two other well-known Real-Time Heuristic Search algorithms: LRTA*-LS and daLSS-LRTA*, and we empirically show significant improvements for LRTA*-LS and modest but still important improvements for daLSS-LRTA*. Furthermore, we prove that wLSS-LRTA* terminates finding a solution if one exists, and we analyze the convergence behavior of wLSS-LRTA*, proving w-optimality as well as other bounds that, in practice, are much tighter than w-optimality.

Keywords: Heuristic Search, A*, Weighted A*, Learning Real-Time A*, Dijkstra's Algorithm, Real-Time Heuristic Search

RESUMEN

Multiplicar la función heurística por un peso mayor que uno es una conocida técnica en Búsqueda Heuristica. Cuando se aplica a A* con una heurística admissible ella produce considerables ahorros de tiempo, a costo de sacrificar optimilidad de la solución. Pocos trabajos han estudiado la aplicabilidad de esta técnica a Búsqueda Heurística en Tiempo Real, un enfoque de búsqueda basado búsqueda heurística.

En este tesis, presentamos dos nuevas enfoques para usar pesos en Búsqueda Heurística en Tiempo Real. El primero, *weighted lookahead*, es una variante del existente enfoque de Shimbo e Ishida. Él incorpora pesos a la fase de lookahead del algoritmo de Búsqueda en Tiempo Real. El segundo, *weighted update*, incorpora los pesos en las aristas del grafo de búsqueda durante la fase de learning. Ambas técnicas son aplicables a una amplia clase de algoritmos de Búsqueda en Tiempo Real. Acá las implementamos a LSS-LRTA*, obteniendo dos nuevos algoritmos. Las evaluamos en benchmarks de panificación de caminos y mostramos que weighted lookahead produce malos resultados pero sin embargo supera el enfoque de Shimbo e Ishida. Weighted update, en la otra mano, produce mejoras de rendimiento de hasta un orden de magnitud tanto en costo de solución como tiempo total de búsqueda. Para ilustrar aún más la generalidad de weighted update, incorporamos la técnica a otros dos conocidos algoritmos de Búsqueda en Tiempo Real: LRTA*-LS y daLSS-LRTA* y empiricamente mostramos mejoras significativas para LRTA*-LS y modestas, pero aún importantes mejores para daLSS-LRTA*. Además, mostramos que wLSS-LRTA* termina encontrando una solución si es que una existe, y analizamos el comportamiento a convergencia de wLSS-LRTA*, probando tanto w-optimalidad como otras cotas que, en practica, son mucho más estrechas que w-optimalidad

Palabras Claves: **Búsqueda Heurísica, A*, Weighted A*, Learning Real-Time A*, Algoritmo de Dijkstra, Búsqueda en Tiempo Real**.

Chapter 1. INTRODUCTION

1.1. Introduction

The term *search* is related to the process of finding solutions to incomplete tasks. Human beings solve many search problems daily: for example, we search for our keys, we search for a good time to schedule a date, etc. In Computer Science, search is a fundamental task, indeed, many of the problems in the area can be stated as a search problem. For instance, given a graph, finding a shortest path between two nodes or a cycle of maximum length in the space of all possible cycles (i.e. a Hamiltonian cycle) involve search.

Artificial Intelligence researchers focused on search since the beginning of the discipline. This is because many problems in Artificial intelligence can be modeled as a search problem. For example motion planning, robot navigation or theorem proving. Furthermore, some other disciplines have problems for which Artificial Intelligence's search techniques have been applied, for example Combinatorial Optimization, Software Verification, and Computational Biology.

Many Artificial Intelligence and computer science search problems can be cast as *state space search problems*. In this setting, there is a search space which consists of states and rules to move from one state to another. Starting from an initial state, the problem is to find the rules to be used over and over in order to reach a goal state. The succession of rules applied form a *path*. Usually the aim is to find a path of minimum cost, i.e. a shortest path.

There exist generic algorithms for solving state space search problems. However, in many Artificial Intelligence applications, state spaces are huge, and generic search algorithms would be too inefficient; indeed, slower than most humans. Heuristic Search is an area of Artificial Intelligence that studies how to exploit domain-specific information in order to guide search allowing solutions to be found faster. The main tool used

in the area are *heuristic functions*, which, informally, allow to discriminate between states that "look closer" to the solution between others that "look farther" from the solution. Heuristic functions can be related to the mechanism used by humans to solve large search problems in a reasonable time.

In many applications of Artificial Intelligence it is convenient to consider that an *agent* (which represents a robot, a videogame character, or the like) actually moves through the states of the search space. This is useful when states correspond to physical locations. In this thesis we adopt this view.

1.1.1. State Space Search Problems

A Search Space is a digraph (S, A) where S represents the states of the problem and A is the set of possible transitions between states (usually called agent actions or rules), such that if $(s, t) \in A$ then the agent may perform an action in order to move from state s to state t. A state space search problem or simply a search problem is defined by a tuple $P = (S, A, c, s_0, G)$ where the digraph (S, A) represents the Search Space, $s_0 \in S$ is the initial state and $G \subseteq S$ is a set of goal states. Finally $c : A \to \mathbb{R}^+$ is a function that specifies the cost that the agent have to incur to use some transition of A.

A solution to a search problem is a path that starts in s_0 and ends in a state in G, using the transitions given by A. The optimization problem consists of minimizing the sum of the costs of the transitions made by the agent, i.e. to find the shortest path from s_0 to a state in G (the shortest path over all elements in G). Sometimes, this problem is referred to the Path-Finding Problem.

Search Problems can be solved, in general, using Dijkstra's Algorithm in $O(|S| \log |S| + |A|)$ (Cormen, Leiserson, Rivest, & Stein, 2001). However, in most real-world search problems the number of states ($|S|$) is very large and it is necessary to use other tools to solve those problems more efficiently. To solve search problems problems faster, research in Artificial Intelligence studied ways to exploit domain-specific information.

A well-studied source of domain-specific information are so-called *Heuristics*. A heuristic h is a function that maps S to $\mathbb{R}_0^+$. A heuristic is admissible if and only if $h(s) \leq h^*(s)$, where $h^*(s)$ is the cost of the shortest path from s to G. In other words h is admissible if and only if it underestimates h^*. Finally, we say that a heuristic is consistent if $h(s) = 0$ for all $s \in G$ and for all $s, t \in S$ with $(s, t) \in A$ we have that $h(s) \leq c(s, t) + h(t)$. It is simple to prove that consistent heuristics are also admissible.

The next section shows examples of Search Problems and heuristics for them. Then we show an algorithm that utilizes heuristics to obtain solutions faster.

1.1.2. Examples

1.1.2.1. TSP

A classic problem in Combinatorial Optimization is the well-known Traveling Salesman Problem (TSP). In this problem we have n cities and a distance matrix d in which entry $d(i, j)$ denotes the distance between the cities i and j. We assume that $d(i, j) \geq 0$ and that $d(i, j) = 0$ if and only if $i = j$. The problem is to find a tour with minimum length that visits each city exactly once, returning to the first city. To formalize TSP as a search problem we use S as the set of incomplete and complete tours starting at an arbitrary city. An action is to add a new city to the incomplete tour and the cost of the action is the distance between the last city of the tour and the city added. The initial state is the tour with the initial city while the set $G \subseteq S$ is the set of complete tours. Given a partial tour $s \in S$ we can compute a heuristic noticing that the cities that are not in the incomplete tour have to be connected by a tour. Then we connect them using the Minimum Spanning Tree (MST), instead completing the tour. Of course this is an underestimation of the distance to G because to complete the tour we have to add a path through those cities, but a path is a tree and then the MST is an admisible heuristic. Notice that obtaining such a heuristic has a complexity of $O(n^2)$ while the TSP is known to be an NP-complete problem in its decision version, and thus there is no known algorithm that solves this problem in polynomial time.

1.1.2.2. Path-Finding in grids

Consider an $n \times m$ grid. Let V be the cells of the grid and $F \subseteq V$ be the set of obstacles. The problem is defined by an initial state s_0 and a goal state s_g, actions (or transitions) move the agent from a cell to one of its neighborting cells that are not obstacles. The cost of the actions is uniform at 1 but they could be different according to the application. For the uniform case a classic heuristic is the so-called *Manhattan distance* which is defined for a state (i, j) as:

$$h(i, j) = |s_g^x - i| + |s_g^y - j|,$$

where $s_g = (s_g^x, s_g^y)$.

Note this heuristic is the distance from (i, j) to s_g when $F = \emptyset$. Clearly this understimates the actual distance, and thus it is an admissible heuristic, when $F \neq \emptyset$.

1.1.3. Exploiting the heuristic function: The A* Algorithm

The most well-known algorithm in Heuristic Search (i.e., search using heuristics) is A* (Hart, Nilsson, & Raphael, 1968). Like the Dijkstra algorithm, and many other algorithms, A* works using the Principle of Suboptimality (Cormen et al., 2001), which states that the shortest path from s_0 to G is equal to the minimum of the sum of the distance from s_0 to a state t, plus the distance from t to G. An alternative phrasing of this principle is that any subpath of the optimal one is itself optimal. The A* algorithm maintains a tentative cost of the shortest path and, in every step, A* attemps to prove that this is the actual cost of the shortest path. Also, in each state s it maintains the following estimation of the optimal distance from s_0 to G through s. That estimation is:

$$f(s) = g(s) + h(s),$$

in which $g(s)$ is the cost of the cheapest known path from s_0 to s and $h(s)$ is the heuristic, which represents an estimate of the shortest distance to G from s. Initially $g(s) = \infty$ for all $s \in S$, except for s_0, for which $g(s_0) = 0$.

A* uses two lists during search: the Open list $\mathcal{O}$ and the Closed list $\mathcal{C}$. Intuitively, a state s is in $\mathcal{C}$ if the algorithm believes it has found the shortest path from s_0 to s while the Open list contains states which potentially could be closed, that is, the current information about them would be enough to close (i.e., add to $\mathcal{C}$) such states. When one of the goal states is closed the execution ends. In each iteration, the algorithm closes the state with minimum f-value. When the algorithm closes a state it is deleted from the Open list, added to the Closed list and then expanded (i.e. generating its neighbors). g-values and f-values of those neighbors is updated and, if it is necessary, they are added to the Open list. The pseudocode for A* is shown in Algorithm 1. Notice that this implementation of A* returns a state in G. If we look at the g-value of such state we obtain the cost of the shortest path from s_0 to G and if we want to know such path we could use recursively the function $parent(\cdot)$. Lines 1 to 6 initialize the algorithm, Lines 7 t o 17 are the main loop. Lines 13 to 15 are the expansion procedure. Line 14 executes $InsertInIpen(s,t)$ which decides if it is necesary to insert t in the Open list and changes its current information if necessary. $InsertInOpen(s,t)$ Pseudo-Code is showed in Algorithm 2

Now we list a number of interesting properties of A*

(i) If h is admisible then A* finds a optimal solution. (Hart et al., 1968)

(ii) If h is consistent then A* needs to expand at most one time each node. (Hart et al., 1968)

(iii) If $h = 0$ then A* is the Dijkstra's Algorithm (by inspection).

(iv) If $h^*(s_0)$ is the optimal solution cost and h is consistent, then A* will expand all states s with $g^*(s)+h(s) < h^*(s_0)$, with $g^*(s)$ the minimum length between s_0 and s. (Hart et al., 1968)

(v) Corollary of the last point is that if $h_1(s) \geq h_2(s)$ for all $s \in S$ and h_1, h_2 are consistent, then A* using h_1 expand fewer states than A* using h_2. (Edelkamp & Schrödl, 2011)

```
Input: A search problem P and a heuristic function h
C := ∅
O := {s_0}
g(s_0) := 0, f(s_0) := h(s_0)
for all s ∈ S \ {s_0} do
    g(s) := ∞
end
while O ≠ ∅ do
    Remove s with minimum f(s) from O and insert it on C.
    if s ∈ G then
        return s
    end
    else
        for each t ∈ Succ(s) do
            InsertInOpen(s, t)
        end
    end
end
```

Algorithm 1: A*

```
Input: States s, t
cost_t = g(s) + c(s, t)
if cost_t > g(t) then
    return
end
parent(t) := s
g(t) := cost_t
f(t) := g(t) + h(t)
if t ∈ C then
    Remove t from C and insert t in O
end
else
    if t ∉ O then
        Insert t in O
    end
end
```

Algorithm 2: InsertInOpen

(vi) As corollary of the last point A* is faster than Dijkstra if the heuristic is consistent.

Note that those properties are useful to understand the difference between A* and Dijkstra's Algorithm. For instance, in our example of Path-Finding in Grids if we take $F = \emptyset$, $s_0 = (s_0^x, s_0^y)$ and $s_g = (s_g^x, s_g^y)$ then Dijkstra's algorithm will expand all states (i, j) of S that satiesfy:

$$|i - s_0^x| + |j - s_0^y| < |s_g^x - s_0^x| + |s_g^y - s_0^y|,$$

and depending on the tie-braking, it might expand several states that satisfy $|i - s_0^x| + |j - s_0^y| = |s_g^x - s_0^x| + |s_g^y - s_0^y|$ while A* with expand a subset of states that satisfies $|i - s_0^x| + |j - s_0^y| = |s_g^x - s_0^x| + |s_g^y - s_0^y|$, depending on the tie-breaking. Well-known practical results state that tie-breaking in favor of states with higher g-values reduce the number of expansions (Edelkamp & Schrödl, 2011). In general, using consistent heuristics, A* expands exactly once all states with $g^*(s) + h(s) < h^*(s_0)$.

Even though A* is a very efficient algorithm, real-world problems have very large Search Spaces, which is a problem to A* in term of time and memory. Variants of A* have been proposed to tackle this problem. One of the most well-known techniques is to add a weight $w > 1$ to the heuristic h in the estimation, ie: instead using $f(s) = g(s) + h(s)$ we use

$$f(s) = g(s) + wh(s),$$

The resulting algorithm is known as weighted A* (wA*) (Pohl, 1970).This algorithm is often used in practice because it used to reduce the number of expansion done by A* at expense of optimality. Indeed, the solution found by wA* are w-optimal; i.e., if the cost of the solution is S, then

$$h^*(s_0) \leq S \leq wh^*(s_0),$$

in practice, the quality of solutions are better than the $wh^*(s_0)$ upper bound.

1.1.4. Real-Time Heuristic Search

Real-Time Search (Korf, 1990) is a different Search paradigm. While in classic Search we assume a complete knowledge of the search graph (S, A), here we don't assume prior knowledge of it (partial knowledge of the graph), but we provide the agent some range of vision that allows the agent to observe the information of the enviroment (commonly the neighborhood of the current state of the agent). In this problem the agent has to explore the graph while it tries to reach G. Usually the agent computes the following three steps until it finds a goal state[1]. First the agent search for a partial path using its current knowledge (lookahead step) then it performs a movement phase (movement step), in which the agent adds new knowledge while it moves in the Seach Space usign such path, and finally it processes the new information (update or learning step). In addition, we incorporate the Real-Time property: the computation that the agent can do in the lookahead and update steps is bounded by a constant (which could be measure in number of operations, time, etc.). Finally if we allow the use of a heuristic we call this problem a Real-Time Heuristic Search Problem. During the learning step the heuristic h is updated to reflect the new knowledge of the agent.

For this class of problems there are two different intesting subjects of study. The first subject is the Single Trial, which is study the performance of the algorithm until it reaches the goal, in particular the First Trial is very important for practical applications. The second subject is the Multiple Trial Run, in which we study of multiple trials: each time the agent reaches the goal state it is moved to the initial state and the trial starts again. In this case, we are interested in the convergence of the heuristic: i.e., h does not change during the execution of a trial and thus the solution does not change either. A multiple-trial run is usually stopped when h converges.

Among the applications of Real-Time Heuristic Search is navigation in video games, in which computer characters are expected to find their path in partially known terrain.

[1]Not necessarily in the presented order.

Game-developing companies impose a constant time limit on the amount of computation per move close to one millisecond for all simultaneously moving characters(Bulitko, Björnsson, Sturtevant, & Lawrence, 2010). Another application is highly dynamic robotics in which the agents move too fast and cannot stay in a position for a long time thinking about the next movement. As a way of example, an automatic aircraft has to stay in continuous movement and cannot stop in the air computing the next movement steps, it has to compute movement and learn the enviroment in real-time.

Learning real-time A* (LRTA*)(Korf, 1990), showed in Algorithm 3, is the simplest algorithm for Real-Time Heuristic Search and is the base of almost all other Real-Time Heuristic Search Algorithms. This algorithm works by iteration of the three steps mentioned above. In Line 4, it searches for the best neighbor to move to (lookahead step), in Lines 5,6 the agent updates the value of $h(s)$ (update step), in Line 8 the agent is moved and in Line 2 the agent observes the enviroment (movement step).

Input: A search problem P and a heuristic function h
1 **while** *The agent has not reached the goal* **do**
2 | Observe the enviroment and update the knowledge of the search graph (arcs and states)
3 | $s :=$ current state
4 | $next := \arg\min_{t:(s,t)\in A}[c(s,t) + h(t)]$
5 | **if** $h(s) < c(s, next) + h(next)$ **then**
6 | | $h(s) := c(s, next) + h(next)$
7 | **end**
8 | Move the agent to $next$.
9 **end**

Algorithm 3: LRTA*

1.2. Thesis work

1.2.1. Objetives

In the previous sections we reviewed briefly the basic framework of Heuristic Search and Real-Time Heuristic Search, and we showed the basic algorithms. Also we showed weighted A* which is used to find solutions faster at the expense of optimality. The

research question that we address in this thesis is: *how can weights be incorporated into Real-Time Heuristic Search algorithms to improve search performance.* Even though weighted A* is well understood, and does improve time performance in Heuristic Search applications, rather surprisingly, current proposals for integrating weights into Real-Time Heuristic Search do not have a positive impact on time or solution quality of the first solution returned. Shimbo and Ishida (Shimbo & Ishida, 2003), to our knowledge the only work that has attempted to answer this question earlier, introduced Weighted LRTA* which multiplies the initial heuristic by a weight $w > 1$ and then run LRTA*. In their paper, they show that their technique has modest to poor performance in the first trial but the convergence is faster than the classical LRTA* but w-optimal. Related, but not specifically answering the question, Bulitko (Bulitko, 2004) uses lower-than-one weights applied to the costs of the arcs, thus indirectly giving more importance to the heuristic. He shows the cost of the first solution increases as more importance is given to h but obtain faster convergence than weighted LRTA* and he propides a bound which in proportional to the inverse of the weight.

1.2.2. Approach

In this thesis we propose two new approaches to incorporating weights in Real-Time Heuristic Search. Both approaches are applicable to a variety of Real-Time Heuristic Search algorithms. The first approach, weighted lookahead, uses weights in the lookahead step of the algorithm. It can be seen as a slight variant of Shimbo and Ishida's approach but, as we see later, outperforms it in practice. The second approach, weighted update, achieves the effect of incorporating w in h by using a different learning rule in which, the higher the w, the higher the amount by which the heuristic may be raised in every update (learning) step. Both approaches are very simple to implement in standard Real-Time Heuristic Search algorithms. We incorporate these techniques to LSS-LRTA*, which is a state-of-the-art variant of LRTA* and one of the best algorithm for Real-Time Heuristic Search, to evaluate these approaches.

1.2.3. Results

Our first technique does not provide good results for the first trial while the second technique has very good results for the first trial. We concentrate in theoretical results for the second technique. We prove that the technique applied to LSS-LRTA* has the desirable properties of a good Real-Time Heuristic Search algorithm. We also prove results on convergence. We prove that if the algorithm is run to convergence, then the solution converges in finite time and that the cost of the converged solution is w-optimal, futhermore, we prove that under suitable conditions on the graph, the bound of the solution does not depend on w but on the graph.

Futhermore we apply the second technique to two other Real-Time Heuristic Search algorithms, LRTA*-LS and daLSS-LRTA*, and we show the application of the technique in videogame maps and mazes. We show that for LSS-LRTA* our technique has excellent results finding solution one order of magnitude better than the classic approach and modest to good results for LRTA*-LS and modest results for daLSS-LRTA*.

Finally, we evaluated the performance of the second technique on LSS-LRTA* under multiple trial runs in practice and we show that the convergence is slower than the classical approach.

1.2.4. Conclusion and Future Work

We proposed two approaches that allow exploiting weights in Real-Time Heuristic Search algorithms: weighted lookahead and weighted update. We incorporated weighted update to LSS-LRTA*, a standard Real-Time Heuristic Search algorithm, and showed it does not yield performance improvements. On the other hand, we incorporated weighted update to LSS-LRTA* and showed it may yield superior performance of up to one order of magnitude in some path-finding benchmarks. Performance gains were also observed when incorporating the technique to other algorithms like LRTA*-LS and daLSS-LRTA*, although improvements on the latter algorithm are less impressive.

In addition we thoroughly analyzed some desirable properties of wLSS-LRTA*. In particular, we prove that it terminates when a solution exists. Furthermore we prove wLSS-LRTA* finds w-optimal solutions on convergence, but we also found bounds that can be much tighter, and, indeed, under certain conditions, we found a bound on solution quality that does not depend on w but only on features of the search graph.

Future work includes the incorporation of these techniques to other Real-Time Heristic Search algorithms that use other kind of learning rule, like, for example, RTAA*. Another line of research has to do with how to determine good values of w and whether or not good policies for adjusting the weight dynamically can be devised. Regarding convergence behavior, it seems necessary to study whether or not using dynamic weights will produce faster convergence results at the expense of sacrificing solution quality.

Chapter 2. ARTICLE SUBMITTED TO ARTIFICIAL INTELLIGENCE

2.1. Introduction

Weighted A* (Pohl, 1970) is a well-known search algorithm for solving single-agent, deterministic search problems. Based on A* (Hart et al., 1968), it uses an evaluation function $f(s) = g(s) + wh(s)$ to rank a state s in the search frontier, where $g(s)$ represents the cost incurred to reach s, $h(s)$, is a (heuristic) estimate of the true cost to reach a solution from s, and the *weight* w is a real value greater or equal to one. It can find a solution substantially faster than A* as the weight is increased over 1. However, the cost of returned solutions may increase as w is increased. If the heuristic h is admissible then the cost of the solutions found can be at most a factor w away from optimal.

Weighting the heuristic is a simple but powerful technique that is widely used in state-of-the-art Heuristic Search algorithms. For example, ARA* (Likhachev, Gordon, & Thrun, 2003), an algorithm used in outdoor rover applications, and RWA* (Richter, Thayer, & Ruml, 2010), the search engine underlying LAMA 2011 (Richter, Helmert, & Westphal, 2008)—among the best-performing satisficing automated planners rely on this technique to obtain superior performance.

Real-Time Heuristic Search (Korf, 1990) is an approach to solving search problems under tight computational time constraints. It has applications ranging from video games to highly dynamic robotics. Many Real-Time Heuristic Search algorithms build on Heuristic Search. Indeed, most popular algorithms use A* as a subroutine.

The research question that we address in this article is: *how can weights be incorporated into Real-Time Heuristic Search algorithms to improve search performance.* Even though weighted A* is well understood, and does improve time performance in Heuristic Search applications, rather surprisingly, current proposals for integrating weights into Real-Time Heuristic Search do not have a positive impact on time or solution quality

of the first solution returned. The approach proposed by Shimbo and Ishida (Shimbo & Ishida, 2003)—to our knowledge, the only one that has attempted to answer this question earlier—consists of multiplying the heuristic function by a weight w at the outset of search. Then the problem is solved using LRTA* (Korf, 1990), a standard Real-Time Heuristic Search algorithm. Shimbo and Ishida's empirical evaluation shows poor performance at finding a first solution. Related, but not specifically answering the question, Bulitko (Bulitko, 2004) proposed an approach that uses lower-than-one weights applied to the costs of the graph, thus indirectly giving more importance to the heuristic h in its evaluation function. He shows the cost of the first solution increases as more "importance" is given to h.

In this article we propose two new approaches to incorporating weights in Real-Time Heuristic Search. Both approaches are applicable to a wide range of Real-Time Heuristic Search algorithms. The first approach, *weighted lookahead*, uses Weighted A* in the lookahead step of the Real-Time Heuristic Search algorithm. It can be seen as a slight variant of Shimbo and Ishida's approach but, as we see later, outperforms it in practice. The second approach, *weighted update*, achieves the effect of incorporating w in h by using a different learning rule in which, the higher the w, the higher the amount by which the heuristic may be raised in every update (learning) step. Both approaches are very simple to implement in standard Real-Time Heuristic Search algorithms.

We implement both approaches on top of the state-of-the-art algorithm LSS-LRTA* (Koenig & Sun, 2009), producing two new algorithms. We evaluate the algorithms over standard videogame path-finding tasks. We show that the weighed lookahead approach, like Shimbo and Ishida's approach, yields both worse solutions and worse running times as w increases. On the other hand, we show that weighted update does yield benefits in *both* solution quality *and* runtime. Improvements are up to one order of magnitude when the algorithm parameter (a measure of the search effort per iteration) is small. The fact that improvements are observed in both solution quality and search time is rather interesting, since weights usually increase solution cost obtained by Heuristic Search

algorithms. Furthermore, we evaluate our best-performing technique theoretically, and (1) show that algorithms that use weighted update, under certain conditions, will always find a solution if one exists, along with other relevant properties, and (2) show that wLSS-LRTA* converges to a w-optimal solution, and provide tighter bounds.

To illustrate the wider applicability of weighted update we also incorporated the technique into LRTA*-LS (Hernández & Meseguer, 2007) and daLSS-LRTA* (Hernández & Baier, 2012). In path-finding tasks we show that weights produce performance gains in these algorithms too. We leave, however, out of the scope of this article a detailed theoretical analysis of these algorithms.

This article significantly extends a previous conference publication (Rivera, Baier, & Hernández, 2013). Among the material not included in the previous publication is what follows.

- An analysis of properties of wLSS-LRTA* at convergence. Specifically, we prove w-optimality, and bounds that can possibly be tighter (Theorems 2.5 and 2.6). One of these results, proves a bound that is *independent* of the value of the weight used, which, nevertheless, is tighter than w-optimality in practice.
- An empirical evaluation of wLSS-LRTA* at convergence in a game map.
- An extension of the existing empirical evaluation by incorporating results over mazes, and by including a new algorithm: $w$daLSS-LRTA*.

The remainder of the article is organized as follows. The next section introduces background on Real-Time Heuristic Search, LSS-LRTA*, and notation that will be used in the remainder of the article. Then we describe the two proposed approaches to incorporating weights and how they can be implemented within LSS-LRTA*. We continue with a thorough theoretical evaluation the weighted update approach both for the first solution and for convergence. Then we evaluate wLSS-LRTA* empirically and show how weighted update can be incorporated into LRTA*-LS and daLSS-LRTA* and show the impact on their performance. We then discuss relevant aspects on the

performance of the algorithms we analyze. The paper finishes with a summary and conclusions.

2.2. Background

A search problem P is a tuple (S, A, c, s_0, G), where (S, A) is a digraph that represents the search space. The set S represents the *states* and the arcs in A represent all available actions. We assume that S is finite, that A does not contain elements of form (s, s), and that (S, A) is a strongly connected graph. In addition, we have a cost function $c : A \mapsto \mathbb{R}^+$ which associates a cost with each of the available actions. For all $s \in S$ we define $Succ(s) = \{t \in S : (s, t) \in A\}$. Finally $G \subseteq S$ is a set of goal states. We define a path π as a sequence of vertices $t_1 t_2 \dots t_n$ such that $(t_i, t_{i+1}) \in A$ for all $i \in \{1, .., n-1\}$. A simple path is a path with no repeated vertices. In this paper we assume every path to be a simple path. We use $V(\pi)$ to refer to the vertices of a path π and we define the length of a path π as $|V(\pi)|$.

The cost of a path $\pi = t_1 t_2 \dots t_n$ is $\sum_{i=1}^{n-1} c(t_i, t_{i+1})$. We denote by d the shortest-path distance in S induced by cost function c. Given a subset T of S we define the frontier of T as $\partial T = \{s \in S \setminus T : \exists t \in T \text{ such that } (t, s) \in A\}$. Furthermore, we define d_T as the shortest-path distance restricted to paths of the form $t_0 \dots t_n$, where $t_0, \dots, t_{n-1} \in T$ and $t_n \in T \cup \partial T$.

A heuristic function $h : S \mapsto [0, \infty)$ associates to each state s an approximation $h(s)$ of the cost of a path from s to a goal state. We denote by $h^*(s)$ the minimum distance from s to a goal state. A heuristic h is *consistent* if and only if $h(s_g) = 0$ for every $s_g \in G$ and for any $s \in S$ it holds that $h(s) \leq c(s, t) + h(t)$ for every $t \in Succ(s)$. If h is consistent then $h(s) \leq d(s, t) + h(t)$ for all $s, t \in S$. Furthermore, if h is consistent it is easy to prove that it is also *admissible*; i.e., $h(s)$ underestimates $h^*(s)$. We say that a state t *justifies* the h-value of state s if $h(s) = c(s, t) + h(t)$.

We assume familiarity with the A* algorithm (Hart et al., 1968): $g(s)$ denotes the cost of the path from the start state to s, and $f(s)$ is defined as $g(s)+h(s)$. The f-value,

g-value and h-value of s refer to $f(s)$, $g(s)$, and $h(s)$ respectively. Furthermore, the variable $Closed$ contains a set of nodes that have been expanded, and $Open$ contains the set of nodes generated by the algorithm that are not in $Closed$. We also use the fact that, after A* expands a node, $Open = \partial Closed$, which is simple to prove by induction on the number of A* iterations.

Finally, given a function $f : A \subseteq S \to \mathbb{R}$, we denote by $\arg\min_{s \in A}[f(s)]$ the subset of elements of A that minimize f over A. Sometimes, abusing notation, we write $t = \arg\min_{s \in A}[f(s)]$ instead of $t \in \arg\min_{s \in A}[f(s)]$.

2.2.1. Real-Time Search

In Real-Time Heuristic Search, the objective is to move an agent from an initial state to a goal state. Between each movement, the computation carried out by the algorithm should be bounded by a constant. An example situation is path-finding in a priori unknown grid-like environments. In this situation, we assume the agent knows the dimensions of the grid but not the location of the obstacles before the search is started.

Most Real-Time Heuristic Search algorithms iterate three steps until they find the solution. In the *lookahead* step, the agent runs a heuristic search algorithm to search for a next move. In the *movement* step, the agents moves to a different position. If the environment is initially unknown, in the movement step the agent also updates its knowledge about the search graph. Finally, in the *update* step, the agent will update the h-value of some of the states in the search space. The update step is usually necessary to guarantee that the algorithm will find a solution. The performance of Real-Time Heuristic Search algorithms is sensitive to the way in which the heuristic is updated (see e.g., (Koenig & Likhachev, 2006)). Finally, we remark the order in which the three steps are carried out depends on the particular algorithm.

Our experimental evaluation focuses on path-finding in grid-like, a priori unknown terrain. Real-Time Heuristic Search algorithms in a priori unknown environments assume that prior to search the agent knows the structure of the graph but does not know the cost function c. While moving through the environment, however, the agent can observe that some arcs in the graph have a cost that is *greater* than the cost it currently knows. In our experiments we undertake the *free-space assumption* (Zelinsky, 1992; Koenig, Tovey, & Smirnov, 2003), a standard assumption about the initial knowledge of the agent, whereby the terrain is initially assumed obstacle-free. The agent, on the other hand, can observe obstacles in the immediate neighborhood of the current cell. When obstacles are detected, the agent updates its cost function accordingly, by setting the cost of reaching a previously unknown obstacle cell to infinity.

Even though our experimental evaluation focuses on path finding, our techniques are implemented over general Real-Time Search algorithms. To use these algorithms in partially known environments, one can a generalized free-space assumption for arbitrary graphs (Rivera, Illanes, Baier, & Hernández, 2013).

LSS-LRTA* (Algorithm 4) is a generalization of the well-known LRTA* algorithm (Korf, 1990). Its lookahead procedure invokes a bounded A* algorithm which expands at most k nodes. At the end of A* the states in $Closed$ are usually referred to as the *local search space*. After lookahead, the h-values of the states in the interior of the local search space are updated. The update formula (Eq. 2.1; Alg. 4) is such that the resulting h-value of s is the maximum possible value that still preserves consistency (Koenig & Sun, 2009). Finally, in the movement step, the algorithm moves the agent as far as possible towards the best state in $Open$, observing the environment, and removing those arcs that lead in or out of states that have been newly observed as obstacles.[1]

A modified version of Dijkstra's algorithm (Algorithm 5) is invoked by LSS-LRTA* in the update step (Line 3). It receives a region of nodes I as input and recomputes

[1]LSS-LRTA*, as described by Koenig and Sun (Koenig & Sun, 2009), modifies the search graph in a different but procedurally equivalent way when observing new obstacles, setting some of the arcs in the graph to infinity. Here we adopt disconnection of such cells to simplify our theoretical analysis.

Input: A search problem P, a heuristic function h, and a lookahead parameter k

while *the agent has not reached a goal state* **do**

  Lookahead: Perform an A* search rooted at the current state. Stop as soon as k nodes have been expanded and added to $Closed$. Furthermore, if just before extracting a node from $Open$ a goal state s_g has minimum f-value in $Open$, stop A* before extracting s_g from $Open$.

  Update: Update the h-values of each state s in $Closed$ such that

$$h(s) := \min_{t \in Open} d_{Closed}(s,t) + h(t). \tag{2.1}$$

  Movement: Let $next$ be the state with lowest f-value in $Open$. Move towards $next$ along the path identified by A*. While moving, observe the environment and update the cost function when new obstacles are found. Stop as soon as $next$ is reached or when an obstacle blocks the next state in the path to $next$.

end

Algorithm 4: LSS-LRTA*

the h-values of states in I by interpreting the h function as the cost of a shortest path between the frontier ∂I and I. As a result, the algorithm sets h-values of states in I according to Equation 2.1 in Algorithm 4. It can be formally shown that this algorithm actually sets the h values according to the update equation (see e.g. (Koenig & Sun, 2009; Hernández & Baier, 2012)).

Input: A region of states I
Effect: If $s \in I$, $h(s)$ is set to $\min_{t \in \partial I} d_I(s,t) + h(t)$

$R := I \cup \partial I$
for each $s \in I$ **do** $h(s) := \infty$
while $R \neq \emptyset$ **do**
  Let t be the state with lowest h-value in R
  for each $s \in I$ *such that* $t \in Succ(s)$ **do**
    if $h(s) > c(s,t) + h(t)$ **then**
      $h(s) := c(s,t) + h(t)$
    end
  end
  remove t from R
end

Algorithm 5: Modified Dijkstra's Algorithm

2.3. Weighted Real-Time Heuristic Search

Now we describe two approaches to incorporating weights into the heuristic function of Real-Time Heuristic Search algorithms. As we show later, the weighted lookahead approach does not produce good results in our benchmark problems. We think however that it deserves to be discussed here because it is the obvious way in which the idea of Weighted A* can be adapted to Real-Time Heuristic Search, and thus relevant conclusions can be obtained by analyzing it theoretically and empirically.

2.3.1. Weighted Lookahead

The *weighted lookahead* approach consists of using Weighted A* in the lookahead phase of the Real-Time Heuristic Search algorithm. It is directly applicable to any Real-Time Heuristic Search algorithm that uses A* in its lookahead phase, but may also be applied to algorithms that use different lookahead procedures.

In this paper we consider incorporating it into LSS-LRTA* and we call the resulting algorithms LSS-LRTwA*. Straightforwardly, LSS-LRTwA* differs from LSS-LRTA* in that Weighted A* instead of A* is called in Line 2 of Alg. 4, with the stop condition left intact.

2.3.2. Weighted Update

A possible reason that explains why Weighted A* finds solutions more quickly than regular A* is that in multiplying the heuristic by a factor $w \geq 1$, the heuristic becomes more *accurate*, in a significant portion of the search space. This is sensible since in many search problems heuristics sometimes grossly underestimate the true cost to reach a solution ((Wilt & Ruml, 2012) provides a good, up-to-date analysis). Real-Time Heuristic Search problems are no different from Heuristic problems in that respect as usually inaccurate heuristics are available. Thus by multiplying the heuristic by a factor greater than 1 one would expect the heuristic to become more accurate in many parts of the search space. Unfortunately, as we show later, incorporating weights in the

lookahead, as done by the previous approach, does not work well in practice. Here we consider incorporating weights to h in an alternative way.

The main idea underlying *weighted update* is to make h increase by a factor of w using the update procedure of the Real-Time Heuristic Search algorithm. To accomplish this, in the update phase we run the standard update algorithm (i.e., Dijkstra) but in a *modified* region I, in which the cost of the arcs between states in I is multiplied by w. As a consequence, for each state s in the interior of the update region I, the heuristic is updated using the following rule:

$$h(s) := \min_{t \in \partial I} w d_I(s,t) + h(t). \tag{2.2}$$

To produce wLSS-LRTA*, we simply change the implementation of Dijkstra's algorithm to consider a weighted cost function.

2.4. Theoretical Analysis

In the previous section we proposed two techniques for real-time heuristic search using a heuristic h multiplied by a factor w greater than 1. Now we focus on the properties of the resulting algorithms. There are a number of properties that real-time search algorithms usually satisfy. Among the most important are termination and convergence after multiple trials. The proofs for these theorems usually rely on the fact that the heuristics are either consistent or admissible. In our techniques however, the heuristic function does not remain consistent and thus one cannot use the same proofs for these results.

In the remainder of the section is divided in two parts. In the first we show an analyze the behavior of the algorithm when the problem is to find a solution after a single run of the algorithm. In the second, we analyze convergence properties, which are interesting when the objective is to find improved solutions by running the algorithm multiple times. Our analysis is focused on the weighted update technique because this is the one that we observed provides good results.

2.4.1. Properties for a Single Run

Here we analyze to what extent some important properties, like finding a solution when one exists, are preserved by our approaches even when the effective heuristic used during search may become inconsistent and hence inadmissible.

As mentioned above, we cannot ensure that the heuristic remains consistent, but we can prove that it remains w-consistent. Furthermore, we can guarantee that the heuristic will remain w-admissible if it is initially inadmissible. The definitions for w-consistency and w-admissibility follow.

Definition 2.1. *Given $w \geq 1$, we say h is w-consistent iff for each pair s, t of connected states $h(s) \leq h(t) + wc(s,t)$, and, for every goal state s_g, $h(s_g) = 0$.*

Definition 2.2. *Given $w \geq 1$, we say h is w-admissible iff for each $s \in S$ we have $h(s) \leq wh^*(s)$.*

Analogous to the case of regular consistency, given that h is w-consistent we can prove h is w-admissible. Henceforth, we assume, without loss of generality, that there is a single goal s_g.

Theorem 2.1. *If h is w-consistent then h is w-admissible.*

PROOF. Let $s \in S$, and let $\pi = t_1 t_2 \dots t_n$, with $t_1 = s$ and $t_n = s_g$, be the shortest path from s to s_g. Since t_n is the goal state $h(t_n) = 0$. Because h is w-consistent, it holds that

$$h(t_i) - h(t_{i+1}) \leq wc(t_i, t_{i+1}), \quad \text{for every } i \in \{1, \dots, n-1\}.$$

Summing up all of the inequations defined above, we obtain:

$$\begin{aligned} h(s) = h(t_1) - h(t_n) &= \sum_{i=1}^{n-1} h(t_{i+1}) - h(t_i) \\ &\leq w \sum_{i=1}^{n-1} c(t_i, t_{i+1}) = wc(\pi) = wh^*(s). \end{aligned}$$

□

We now turn our attention to prove that any algorithm that can (correctly) incorporate our weighted update will terminate. First we prove the following intermediate result.

Lemma 2.1. *Let h be a w-consistent heuristic. If we apply Modified Dijkstra's Algorithm using wc as the graph's cost function in a set of states L, then the value of h will not decrease.*

PROOF. Let us denote by h' the new heuristic function after running the weighted update Djikstra Algorithm on region L. Note that $h = h'$ in $S \setminus L$. Let $\bar{L}$ be the set of all states in L whose h-value has decreased. We prove that $\bar{L} = \emptyset$ by contradiction. Assume $\bar{L} \neq \emptyset$ and let $l \in \bar{L}$ be a state with minimum h' value in L. By correction of Dijkstra's Algorithm any vertex $s \in L$ satisfies $h'(s) = \min_{t \in Succ(s)}[h'(t) + wc(s,t)]$. Now let $u = \arg\min_{t \in Succ(l)}[h'(t) + wc(l,t)]$ then:

$$h'(l) = h'(u) + wc(l,u), \tag{2.3}$$

and because $c > 0$, from Equation (2.3) we obtain:

$$h'(l) > h'(u).$$

Now because l has the lowest h'-value of all states in $\bar{L}$, we conclude that $u \notin \bar{L}$. By definition of $\bar{L}$ it must be the case that $h'(u) \geq h(u)$. Using that $h'(u) \geq h(u)$ and that h is w-consistent we write:

$$h'(l) = h'(u) + wc(l,u) \geq h(u) + wc(l,u), \tag{2.4}$$

but because h is w-consistent, $h(u) + wc(l,u) \geq h(l)$, so using Inequation (2.4) we conclude $h'(l) \geq h(l)$, which contradicts the fact that $h'(l) < h(l)$. We conclude that $\bar{L}$ is empty and that no state in L decreases its h-value. □

Now we establish that the property of w-consistency is preserved by the algorithm. The proof follows from the two following Lemmas.

Lemma 2.2. *If h is a w-consistent heuristic then h remains w-consistent after running a w-weighted update.*

PROOF. As in the proof above, let h' denote the updated heuristic. We want to establish $h'(s) \leq h'(t) + wc(s,t)$, for all $(s,t) \in A$. Let $s \in S$. We divide the proof in 3 cases:

Case 1: If $s \in I$ then

$$h'(s) = \min_{t \in Succ(s)} h'(t) + wc(s,t)$$

then for all t such that $(s,t) \in A$ we have that $h'(s) \leq h'(t) + wc(s,t)$.

Case 2: If $s \notin I$ and $s \neq s_g$ and $(s,t) \in A$. Since h is not updated outside of I, $h'(s) = h(s)$. Because h is consistent and $h(t) \leq h'(t)$ (by Theorem 2.1), the following inequality holds:

$$h'(s) = h(s) \leq h(t) + wc(s,t) \leq h'(t) + wc(s,t).$$

Case 3: if $s \notin I$ and $s = s_g$ then $h'(s) = h(s) = 0$. Then, since $c \geq 0$, it trivially holds that $h'(s) \leq h'(t) + wc(s,t)$, for all $(s,t) \in A$. □

When the domain is partially known an LSS-LRTA* agent may discover that the search graph is different from what it initially thought as it moves through the environment. We assumed in the Section 2.2 that in pathfinding applications states that are blocked (i.e., obstacles) are disconnected from the search graph in the movement phase as soon as they are discovered. When removing arcs of the search space, consistency and w-consistency are trivially preserved. In more general setting, consistency is also preserved if in the movement phase the costs of the arcs are increased by the algorithm. This could happen because the agent had, at the outset of search, an optimistic estimate

of arc costs instead of the true arc costs. In such cases, consistency is also preserved by the algorithm, as the next Lemma proves.

Lemma 2.3. *If the movement phase of a Real-Time Heuristic Search algorithm may only increase costs in the search graph, then w-consistency is preserved by the movement phase.*

PROOF. Let c' denote the cost function after the movement phase. Since costs may only increase, $c \leq c'$. If $(s,t) \in A$ and h is w-consistent then $h(s) \leq wc(s,t) + h(t)$, which implies $h(s) \leq wc'(s,t) + h(t)$. □

Theorem 2.2. *If h is initially w-consistent, then it remains w-consistent along the execution of a Real-Time Heuristic Search algorithm that uses a w-weighted update and whose movement phase may only increase the costs of arcs in the search graph, and whose lookahead phase does not change h or c.*

PROOF. Straightforward from Lemmas 2.2 and 2.3, and the fact that the lookahead phase does not change h or c. □

Now we focus on our termination result, and assume we are dealing with a Real-Time Heuristic Search algorithm that satisfies the conditions of Theorem 2.2. For notational convenience, let h_n denote the heuristic function at the beginning of the n-th iteration of the algorithm. An important intermediate result is that eventually h converges.

Lemma 2.4. *h eventually converges; that is, there exists an $N \in \mathbb{N}$ such that $h_{n+1} = h_n$ for all $n \geq N$.*

PROOF. Let c_* denote the minimum cost arc in (S, A). h_n is a bounded non decreasing series, thus by elementary calculus the series converges pointwise, that is, $h_n(s)$ converges for all $s \in S$. Moreover, if the $h(s)$ increased in some iteration n, then $h_{n+1}(s) - h_n(s) > wc_*$ and hence the h-value of s cannot increase more than $\frac{h^*(s)}{c_*}$ times. Convergence therefore is reached for a finite number N. □

Now, we stablish the main result of this subsection:

Theorem 2.3. *wLSS-LRTA* reaches s_g if the heuristic is initially w-consistent.*

PROOF. Suppose the assertion is false. Since by Lemma 2.4 $\{h_n\}$ converges, at some iteration $h = h_n$ does not change anymore and the agent enters a loop. Assume $\pi = t_1 t_2 \dots t_n t_1$ is such a loop, and let $\pi' = t'_1 t'_2 \dots t'_m, t'_1$ be the states at which the agent runs a lookahead step. Without loss of generality, assume t'_1 is one of the states of π' with smallest heuristic value. Let L be the local search space of the algorithms. Since h does not change, there exists a state $t \in \partial L$ and such that $h(t'_1) = h(t) + w d_L(t'_1, t)$, otherwise h would be updated.

Since wLSS-LRTA* decides to move to the best state in ∂L, and that such a state is t'_2, we know that

$$h(t) + d_L(t'_1, t) \geq h(t'_2) + d_L(t'_1, t'_2), \tag{2.5}$$

But we have that $h(t'_1) = h(t) + w d_L(t'_1, t)$. Substituting in $h(t)$ in Inequation (2.5), we obtain:

$$h(t'_1) - w d_L(t'_1, t) + d_L(t'_1, t) \geq h(t'_2) + d_L(t'_1, t'_2)$$

Finally, because t'_1 has a lowest h-value in π', we have that $h(t'_1) \leq h(t'_2)$ and thus:

$$h(t'_2) - w d_L(t'_1, t) + d_L(t'_1, t) \geq h(t'_2) + d_L(t'_1, t'_2).$$

Then,

$$-w d_L(t'_1, t) + d_L(t'_1, t) \geq d_L(t'_1, t'_2),$$

and, rearranging, we obtain:

$$w \leq 1 - \frac{d_L(t'_1, t'_2)}{d_L(t'_1, t)} < 1,$$

which is a contradiction with the fact that $w \geq 1$. Thus, the agent cannot enter an infinite loop. □

Remark 2.1. *The proof of Theorem 2.3 applies to any algorithm whose movement phase moves the agent to the state with lowest f-value of the frontier of the local search space.*

2.4.2. Properties for Multiple-Trial Runs

In this section we analyze properties of our algorithms when running multiple search trials. In this mode, each time the agent reaches the goal, it is moved back to the initial state and the search algorithm is invoked again, without resetting the heuristic function h. Each run of the algorithm that moves the agent from the initial state to a goal state is called a *trial*. A multiple-trial run is said to *converge* iff the heuristic h *converges*, i.e., h does not change during the execution of a trial. A multiple-trial run is usually stopped when h converges.

Multiple-trial runs of LSS-LRTA* are known to converge if the initial heuristic is consistent. Furthermore, they converge to an optimal solution since the agent traverses an optimal path in the last trial (Korf, 1990; Koenig & Sun, 2009).

In this section we analyze multiple-trial properties of wLSS-LRTA*. Specifically, we prove that multiple-trial runs of wLSS-LRTA* also converge. We prove that if $w > 1$ our algorithms find a w-optimal solution (i.e., a solution that can exceed the optimal by at most a factor of w). Furthermore we provide bounds for the cost of the solution at convergence that can be in practice much tighter than w-optimality. Indeed in Theorem 2.5 we prove that the solution cost at convergence exceeds the optimal cost by a factor that can be computed from h during runtime. In grids, we have observed that such a value is low even when the algorithm is run with a high value for w. In addition, when the search graph satisfies the triangle inequality, we prove a bound on solution cost at convergence that depends on the ratio between the highest-cost and the lowest-cost arc in the graph but not on w. As a corollary, we obtain that in graphs with uniform costs, an optimal solution is returned at convergence, independent of the value of w.

In the following, we assume $s_0, ..., s_N$ are the states at which the agent performs a lookahead step in a run after convergence has been reached, that starts in the initial state and reaches the goal. In addition, for every $i \in \{0, ..., N-1\}$, we denote by C_i the set of states in the A*'s *Closed* list after the search started from s_i finishes. To simplify notation, we write d_i instead of d_{C_i}. Finally, we assume that the initial heuristic is w-consistent, which implies that the converged heuristic also is w-consistent.

Our first result is w-optimality. We need the following intermediate result.

Lemma 2.5. *In a run of wLSS-LRTA* at convergence, for each $i \in \{0, ..., N-1\}$, it holds that:*

$$h(s_i) \geq h(s_{i+1}) + d_i(s_i, s_{i+1}).$$

PROOF. Note that for each i there is some $t_i \in \partial C_i$ such that

$$h(s_i) = h(t_i) + w d_i(s_i, t_i)$$

but since A* chooses to move to s_{i+1} instead t_i we have:

$$h(s_{i+1}) + d_i(s_i, s_{i+1}) \leq h(t_i) + d_i(s_i, t_i) \leq h(t_i) + w d_i(s_i, t_i) = h(s_i),$$

which finishes the proof. □

Theorem 2.4. *The cost of the solution found by wLSS-LRTA* at convergence is w-optimal.*

PROOF. Note that the cost of the path found by wLSS-LRTA* is:

$$\sum_{i=0}^{N-1} d_i(s_i, s_{i+1}),$$

and by Lemma 2.5:

$$\sum_{i=0}^{N-1} d_i(s_i, s_{i+1}) \leq \sum_{i=0}^{N-1} h(s_i) - h(s_{i+1}) \leq h(s_0) - h(s_g) = h(s_0).$$

However h is w-consistent and thus w-admissible, hence:

$$\sum_{i=0}^{N-1} d_i(s_i, s_{i+1}) \leq w h^*(s_0),$$

which means the solution found is w-optimal. □

The argument used to prove Theorem 2.4 is generic since it does not depend on the cost structure of the graph. Below we present two possibly tighter bounds for the cost of the solutions found by wLSS-LRTA* at convergence. One bound can be computed as soon as wLSS-LRTA* has converged (Theorem 2.5), and the other can be determined from the cost structure of the graph (Theorem 2.6).

Henceforth, we use C_i with the meaning defined above and we denote by t_i the state of ∂C_i such that:

$$h(s_i) = w d_i(s_i, t_i) + h(t_i). \tag{2.6}$$

I.e., t_i is the state whose h-value determines the h-value of s_i after the weighted update; therefore, t_i always exists. Finally, the following results use the following two inequalities:

$$h(s_{i+1}) + d_i(s_i, s_{i+1}) \leq h(t_i) + d_i(s_i, t_i) \tag{2.7}$$

$$h(s_{i+1}) + w d_i(s_i, s_{i+1}) \geq h(t_i) + w d_i(s_i, t_i) \tag{2.8}$$

Inequality (2.7) holds because in each A* run, s_{i+1} is preferred over t_i for expansion. Moreover, Inequality (2.8) holds because of the update algorithm.

To prove our first bound we need two intermediate results that are proven below.

Lemma 2.6. *For all $i \in \{0, ..., N-1\}$ it holds that $d_i(s_i, s_{i+1}) \geq d_i(s_i, t_i)$ and $h(t_i) \geq h(s_{i+1})$. Moreover $d_i(s_i, s_{i+1}) = d_i(s_i, t_i)$ if and only if $h(s_{i+1}) = h(t_i)$.*

Proof. Inequality (2.8) can be rewritten as:

$$h(t_i) + w d_i(s_i, t_i) \leq h(s_{i+1}) + d_i(s_i, s_{i+1}) + (w-1) d_i(s_i, s_{i+1}), \tag{2.9}$$

but using Inequality (2.7) in the right-hand side of Inequality (2.9) we obtain:

$$h(t_i) + wd_i(s_i, t_i) \leq h(t_i) + d_i(s_i, t_i) + (w-1)d_i(s_i, s_{i+1}).$$

Subtracting $h(t_i) + d_i(s_i, t_i)$ we obtain:

$$(w-1)d_i(s_i, t_i) \leq (w-1)d_i(s_i, s_{i+1}),$$

and using $w > 1$ we obtain

$$d_i(s_i, t_i) \leq d_i(s_i, s_{i+1}) \tag{2.10}$$

Now we add Inequalities (2.10) and (2.7), and we obtain $h(t_i) \geq h(s_{i+1})$. The same applies for the if and only if part, we substitute the equations in both Inequalities (2.7) and (2.8) to obtain the desired result. □

In the remainder of the section we define $D_i = h(t_i) - h(s_{i+1})$, and $D = \sum_{i=0}^{N-1} D_i$. The value D_i represents the difference between the h-value of the state that updates the value of s_i and the h-value of the state at which the agent actually moves to. D, on the other hand, is simply the sum of those differences throughout the path from the initial state to the goal.

Lemma 2.7. *In a run of wLSS-LRTA* at convergence, it holds that:*

$$\sum_{i=0}^{N-1} d_i(s_i, t_i) \leq h^*(s_0) - \frac{D}{w}.$$

Proof. Using the definition for D, we rewrite Equation (2.6) as:

$$h(s_i) - h(s_{i+1}) = wd_i(s_i, t_i) + D_i,$$

and given that $s_N = s_g$ and that $h(s_g) = 0$, we obtain:

$$h(s_0) = w \sum_{i=0}^{N-1} d_i(s_i, t_i) + D.$$

Finally, by Lemma 2.6, $D_i \geq 0$, and thus $D \geq 0$. Finally, because h is w-admissible, $h(s_0) \leq wh^*(s_0)$, we conclude that:

$$w \sum_{i=0}^{N-1} d_i(s_i, t_i) + D \leq wh^*(s_0),$$

from where it is straightforward to obtain the desired inequality. □

Now we establish a bound on the solution that depends on both D and w.

Theorem 2.5 (D-Bound)**.** *Let C be the cost of a solution found by a run of wLSS-LRTA* at convergence. Then,*

$$C \leq h^*(s_0) + \frac{w-1}{w} D$$

Proof. Using Equation (2.7) and the definition of D_i we obtain:

$$d_i(s_i, s_{i+1}) \leq h(t_i) - h(s_{i+1}) + d_i(s_i, t_i) = d_i(s_i, t_i) + D_i$$

and then:

$$C = \sum_{i=0}^{N-1} d_i(s_i, s_{i+1}) \leq \sum_{i=0}^{N-1} (d_i(s_i, t_i) + D_i),$$

given that $\sum_{i=0}^{N-1} D_i = D$, we write:

$$C \leq \sum_{i=0}^{N-1} d_i(s_i, t_i) + D \tag{2.11}$$

By adding up Inequality (2.11) with the inequality of Lemma 2.7, we obtain the desired inequality. □

Note that the last bound is not uniquely defined, because each D_i depends on the election of t_i and, algorithmically, we can optimize this bound by letting the update algorithm choose a t_i that minimizes $h(t_i)$. In addition, note that $D = 0$ implies that the solution found by the algorithm is optimal.

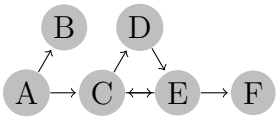


FIGURE 2.1. A graph in which the arcs connecting ACEF is a straight path.

For the next result, it is useful to have the following definitions. In particular a *straight paths* is, intuitively, a path that does not allow shortcuts. Formally,

Definition 2.3. *We say that a path $\pi = t_1 t_2 \dots t_n$ in a graph (V, A) is a* straight path *iff $(t_i, t_k) \notin A$ for all $k > i + 1$.*

As an example, consider Figure 2.1, in which ACEF is a straight path while ACDEF is not.

Definition 2.4. *We say a graph $G = (V, A, c)$ satisfies the triangle inequality iff for all $(s, t) \in A$, it holds that $d(s, t) = c(s, t)$.*

Thus, the triangle inequality is satisfied if and only the shortest path between two adjacent nodes is always given by the arc that connects them. As a way of example, 4-connected and 8-connected grids with $\sqrt{2}$ as diagonal cost are graphs that satisfy the triangle inequality.

The following result proves that if the sequence of nodes expanded by A* is a straight path in a graph that satisfies the triangle inequality then the f-value of a node in the Open list cannot decrease.

Lemma 2.8. *Let $\pi = t_0 t_1 \dots t_n$ be a straight path of a graph, and assume that when A* is run with t_0 as a start node it eventually has exactly the nodes in π in its Closed list. Let O denote the set of states in the Open list just before t_n is expanded. After expanding t_n the f-values of the states in O do not decrease.*

PROOF. Assume the contrary, and let u be a node in O whose f-value has decreased after expanding t_n. Note that since the h-value does not change through execution, this

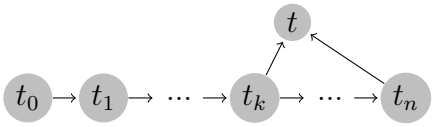


FIGURE 2.2. If the costs of the graph satisfy the triangle inequality $c(t_k, t) \leq c(t_n, t) + \sum_{i=k}^{n-1} c(t_i, t_{i+1})$.

means the g-value has decreased after expanding t_n. Because the nodes that have been expanded form a straight path, just before expanding t_n, the g-value of u—which we denote by $g^+(u)$—is the cost of a path from t_0 to u which has to be of the form $t_0 t_1 \dots t_k u$, for some $k < n$. Thus,

$$g^+(u) = c(t_k, u) + \sum_{i=0}^{k-1} c(t_i, t_{i+1}). \tag{2.12}$$

After expanding t_n, the new g-value of u—which we denote by $g^-(u)$—must correspond to the cost of a path reaching u through t_n. Thus,

$$g^-(u) = c(t_n, y) + \sum_{i=0}^{n-1} c(t_i, t_{i+1}). \tag{2.13}$$

However the g-value has decreased, therefore $g^-(u) < g^+(u)$, which implies that

$$c(t_n, u) + \sum_{i=k}^{n-1} c(t_i, t_{i+1}) < c(t_k, u),$$

which violates the triangle inequality. See Figure 2.2 for a depiction. □

The following Lemma implies that when the heuristic has converged, each A* run indeed expands nodes along a straight path in the search graph.

Lemma 2.9. *Consider a run of A* at convergence and let $L = (t_0, t_1, ..., t_{n+1})$ be such that t_i is the i-th state expanded by A*. If the search graph (S, A, c) satisfies the triangle inequality then*

(i) *$t_0 t_1 \dots t_{n+1}$ is a straight path in (S, A, c), and*
(ii) *$t_{i+1} = \arg\min_{t \in Succ(t_i)} [h(t) + c(t_i, t)]$ for all $i \in \{0, \dots, n\}$.*

PROOF. The proof is by induction on the length of L. For the base case, if $n = 0$, the list has only one element and thus satisfies trivially the conditions of the lemma.

For the induction, we assume $t_0 t_1 \dots t_n$ is a straight path in the search graph. Furthermore, we assume t_{n+1} is the next state selected for expansion.

Because the elements of L induce a path in the search graph, it holds that $g(t_n) = \sum_{i=0}^{n-1} c(t_i, t_{i+1})$, and thus:

$$f(t_n) = h(t_n) + \sum_{i=0}^{n-1} c(t_i, t_{i+1}) \tag{2.14}$$

Since the h-value of t_n does not change after the update, it holds that:

$$h(t_n) = \min_{t \in Succ(t_n)} [wc(t_n, t) + h(t)]. \tag{2.15}$$

Now we define

$$u = \arg \min_{t \in Succ(t_n)} [c(t_n, t) + h(t)]. \tag{2.16}$$

The rest of the proof is divided in four steps.

Step 1 Our first step is to prove that $f(u) < f(t_n)$. Because $w > 1$, it holds that

$$h(t_n) > c(t_n, u) + h(u) \tag{2.17}$$

And, moreover, since costs are positive we have that:

$$h(t_n) > h(u) \tag{2.18}$$

Since after expanding t_n, state u is in A*'s Open list we know that its f-value is at most the one that would result by considering that the cheapest path towards u goes through t_n. This allows us to write:

$$f(u) \leq g(t_n) + c(t_n, u) + h(u) = c(t_n, u) + h(u) + \sum_{i=0}^{n-1} c(t_i, t_{i+1}) \tag{2.19}$$

But because of Inequation (2.17), we can substitute $c(t_n, u)+h(u)$ by $h(t_n)$ in Inequation (2.19), obtaining:

$$f(u) < h(t_n) + \sum_{i=0}^{n-1} c(t_i, t_{i+1}) = f(t_n) \tag{2.20}$$

This finishes the first step. Before continuing with the next step, let O_n denote the contents of A*'s Open list just before expanding t_n.

Step 2 In the second step, we prove that $u \notin \{t_1, \ldots, t_n\}$. Clearly, $u \neq t_n$. For each $i \in \{1, \ldots, n-1\}$ we have that the h value of t_i does not change in the update process and thus satisfies:

$$h(t_i) = \min_{t \in Succ(t_n)} [wc(t_i, t) + h(t)] > \min_{t \in Succ(t_n)} [c(t_i, t) + h(t)],$$

but using the induction hipotesis in the second part of the lemma:

$$\min_{t \in Succ(t_n)} [c(t_i, t) + h(t)] = c(t_i, t_{i+1}) + h(t_{i+1}).$$

then $h(t_i) > h(t_{i+1})$. Finally, if $u = t_i$ for some $i \in \{1, \ldots, n-1\}$ by using Inequation (2.18) we obtain that:

$$h(u) > h(t_n) > h(u),$$

which is a contradiction and thus $u \neq t_i$ for all $i \in \{1, \ldots, n\}$.

Step 3 In the third step, we prove that $u \notin O_n$. Note that since t_n is chosen for expansion in O_n, then

$$f(s) \geq f(t_n), \quad \text{for all } s \in O_n \tag{2.21}$$

Assume $u \in O_n$. Then, by Lemma 2.8 the f-value of u does not decrease once we expand t_n, so we have that Inequation (2.21) applies and we can write that $f(u) \geq f(t_n)$, which, together with Inequation (2.20) implies that $f(u) = f(t_n)$. Since $f(u)$ has not changed after expanding t_n, then just before expanding t_n it was also the case that $f(u) = f(t_n)$. Furthermore, since A* breaks ties in favor of states with greater g-value, it must have been that $g(u) \leq g(t_n)$. But, since $f(u) = f(t_n)$, this implies that $h(u) \geq h(t_n)$, which contradicts Inequation (2.18). We conclude that it cannot be the case that $u \in O_n$.

Step 4 Now we prove that the state to be expanded after t_n, t_{n+1}, was not generated by A* before u; i.e., $t_{n+1} \notin O_n$. Indeed, assume $t_{n+1} \in O_n$. Note that this implies $u \neq t_{n+1}$. Further, note that Lemma 2.8 and Inequation (2.21) imply that after and before expanding t_n, it holds that $f(t_{n+1}) = f(t_n)$. Since after expanding t_n, t_{n+1} is preferred for expansion over u, then $f(t_{n+1}) \leq f(u)$. But since $f(u) \leq f(t_n)$, we conclude

$$f(t_{n+1}) = f(u). \tag{2.22}$$

On the other hand, just before expanding t_n both t_n and t_{n+1} are in the Open list and t_n is preferred by A* for expansion; thus, it must be the case that $g(t_n) \geq g(t_{n+1})$. On the other hand, u is expanded via t_n and therefore $g(u) > g(t_n)$, from where we conclude that $g(u) > g(t_{n+1})$. Thus, right after expanding t_n, both u and t_{n+1} are in the Open list, with the same f-value (Equation 2.22) and thus u, rather than t_{n+1}, should be preferred for expansion. This contradicts the fact that A* chooses t_{n+1} for expansion.

Step 4 concludes the proof for Condition 1 of this lemma since it holds that $t_{n+1} \notin O_n$, which means that t_{n+1} is a state that is a successor of t_n which is not a successor of any other t_i, for every $i < n$. This ensures that $t_0 t_1 \ldots t_n t_{n+1}$ is a straight path.

To prove Condition 2 of the lemma, observe that since t_{n+1} is the next state to be expanded and that is on a straight path, then:

$$f(t_{n+1}) = h(t_{n+1}) + \sum_{i=0}^{n} c(t_i, t_{i+1}) \leq h(u) + c(t_n, u) + \sum_{i=0}^{n-1} c(t_i, t_{i+1}) = f(u), \tag{2.23}$$

which simplifies to:

$$h(t_{n+1}) + c(t_n, t_{n+1}) \leq h(u) + c(t_n, u).$$

Given the definition of u (Equation 2.16), we conclude that:

$$t_{n+1} = \arg \min_{t \in Succ(t_n)} [c(t_n, t) + h(t)],$$

which concludes the proof for the lemma. □

Lemma 2.10 (Simulation Lemma)**.** *Let $\pi = s_0 s_1 \dots s_N$ with $s_N = s_g$ be the path traversed by wLSS-LRTA* run with lookahead parameter equal to $k > 1$, and with a version of A* that breaks ties towards states with greater g-values. Let π' be the path traversed by wLSS-LRTA* run with lookahead parameter equal to 1, breaking ties towards states in π. If the search graph (S, A, c) satisfies the triangle inequality then $\pi = \pi'$.*

PROOF. The proof is by induction on N. Let $\pi' = t_0 t_1 \dots t_M$.The base case ($M = 0$) is trivial since $s_0 = t_0$. Suppose that $s_0 \dots s_n = t_0 \dots t_n$. We prove that $s_{n+1} = t_{n+1}$. State s_n was expanded by some iteration of A* in a wLSS-LRTA* run using with lookahead parameter k. Then, by Condition 2 of Lemma 2.9, the next state s_{n+1} is $\arg\min_{s \in Succ(t_n)}[h(s) + c(t_n, s)]$. Moreover, wLSS-LRTA* with lookahead parameter equal to 1 will indeed prefer some of $\arg\min_{s \in Succ(t_n)}[h(s) + c(t_n, s)]$ and because the tie-breaking of wLSS-LRTA* with lookahead parameter equal to 1 is the path π we prefer to move to s_{n+1}, therefore $t_{n+1} = s_{n+1}$. □

Theorem 2.6. *Let P be a search problem whose search graph satisfies the triangle inequality. Let C be the cost of a solution found by wLSS-LRTA* at convergence when run with lookahead value k, on a search graph that satisfies the triangle inequality. Then $C \leq \frac{c^*}{c_*} h^*(s_0)$.*

PROOF. By Lemma 2.10, we consider a simulation of wLSS-LRTA* by a run of wLSS-LRTA* with lookahead 1 and we apply our analysis to the simulation. Notice that for all i, $(s_i, t_i) \in A$ and $(s_i, s_{i+1}) \in A$ because $k = 1$. Moreover $d(s_i, s_{i+1}) = c(s_i, s_{i+1})$ and the same holds for (s_i, t_i). Finally, since $c_* \leq d(s_i, t_i)$ and $d(s_i, s_{i+1}) \leq c^*$, it holds that:

$$d(s_i, s_{i+1}) \leq \frac{c^*}{c_*} d(s_i, t_i),$$

which allows us to write

$$C = \sum_{i=1}^{N-1} d(s_i, s_{i+1}) \leq \frac{c^*}{c_*} \sum_{i=1}^{N-1} d(s_i, t_i) \tag{2.24}$$

Substituting the Inequation of Lemma 2.7, we obtain

$$C \leq \frac{c^*}{c_*}\left(h^*(s_0) - \frac{D}{w}\right), \tag{2.25}$$

which implies the desired inequality because $D/w \geq 0$ (cf. Lemma 2.6). Finally using that S is the same for wLSS-LRTA* with lookahead k and for the simulation we conclude the proof. □

Corollary 2.1. *Let P be a search problem whose search graph has unit costs. Then the solution found by wLSS-LRTA* is optimal, independent of w.*

Proof. Straightforward from Theorem 2.6. □

The previous theorems provide tighter convergence bounds for wLSS-LRTA* than w-optimality. Note that when it is possible to compute D, one should use Inequation 2.25 in order to compute a bound instead of that given by Theorem 2.6.

2.5. Empirical Evaluation of wLSS-LRTA* and LSS-LRTwA*

We evaluate our algorithms in the context of goal-directed navigation in a priori unknown grids (Zelinsky, 1992; Koenig et al., 2003). The agent always senses the blockage status of its eight neighboring cells and can then move to any one of the unblocked neighboring cells. We use eight-neighbor grids in the experiments since they are often preferred in practice, for example in video games (Bulitko, Björnsson, Sturtevant, & Lawrence, 2011).The cost for horizontal or vertical moves is one, and cost for diagonal moves is$\sqrt{2}$. The user-given h-values are the octile distances (Bulitko & Lee, 2006).

We used twelve maps from deployed video games and 3 acyclic mazes with corridors of varying width to carry out the experiments. The first six game maps are taken from the game *Dragon Age*, and the remaining six are taken from the game *StarCraft*. The game maps and mazes maps were retrieved from Nathan Sturtevant's pathfnding

repository.[2] We average our experimental results over 200 test cases with a reachable goal cell for each map. For each test case, the start and goal cells are chosen randomly. All the experiments were run in a 2.00GHz QuadCore Intel Xeon machine running Linux. Time is measured in milliseconds.

2.5.1. Weighted Lookahead

We start by discussing results obtained by LSS-LRTwA* and Shimbo and Ishida's approach. To that end, we refer the results presented by us in an earlier publication (Rivera, Baier, & Hernández, 2013). We do not extend such an evaluation since results shown therein (1) can be considered poor and thus we did not think an extended evaluation of such algorithms would be ofq1 value to this extended article, and furthermore (2) results obtained by wLSS-LRTA* are significantly improve upon those obtained by LSS-LRTwA* and Shimbo and Ishida's approach. LSS-LRTwA* and Shimbo and Ishida's approach were evaluated for three weight values $\{1, 2, 4\}$ and six values for the lookahead parameter $\{1, 2, 4, 8, 16, 32, 64\}$. Both algorithms were implemented in a similar way, using the same, standard binary heap for the *Open* list, and breaking ties among states with the same f-value in favor of larger g-values. Figure 2.3 shows a plot of solution cost versus lookahead parameter. We concluded that as w increases, the solution cost obtained by LSS-LRTwA* also increases. Larger differences are observed when the lookahead parameter increases. On the other hand, for Shimbo and Ishida's approach the solution cost increases more significantly when w is increased (e.g., more than twice as expensive when $w = 2$). In summary, the performance of both algorithms degrade as w increases but Shimbo and Ishida's approach is the one the exhibits a greater performance degradation.

[2] `http://www.movingai.com/`. For Dragon Age we used the maps brc202d, orz702d, orz900d, ost000a, ost000t and ost100d of size 481×530, 939×718, 656×1491, 969×487, 971×487, and 1025×1024 cells respectively. For StarCraft, we used the maps ArcticStation, Enigma, Inferno JungleSiege, Ramparts and WheelofWar of size 768×768, 768×768, 768×768, 768×768, 512×512 and 768×768 cells respectively. For Maze maps, we used maze512-8-0, maze512-4-0, maze512-2-0 of size 512×512.

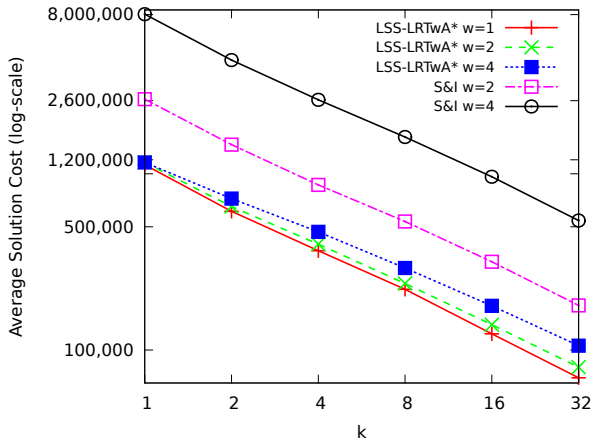
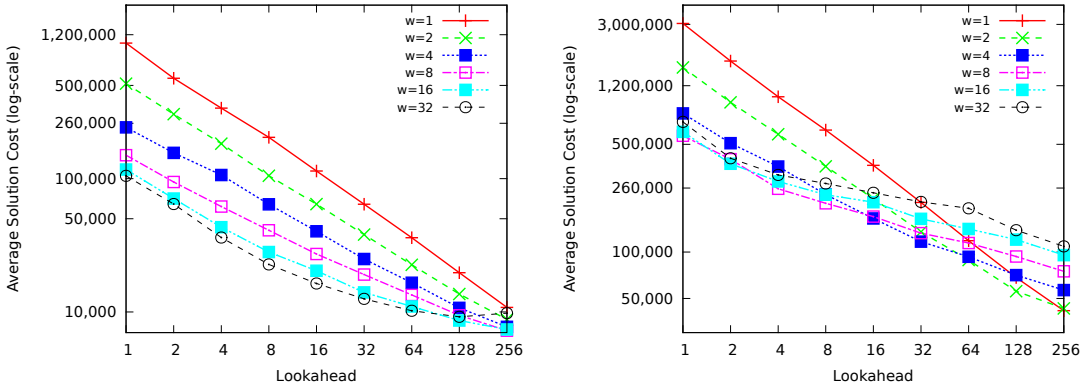


FIGURE 2.3. LSS-LRTwA* and Shimbo and Ishida's Approach.

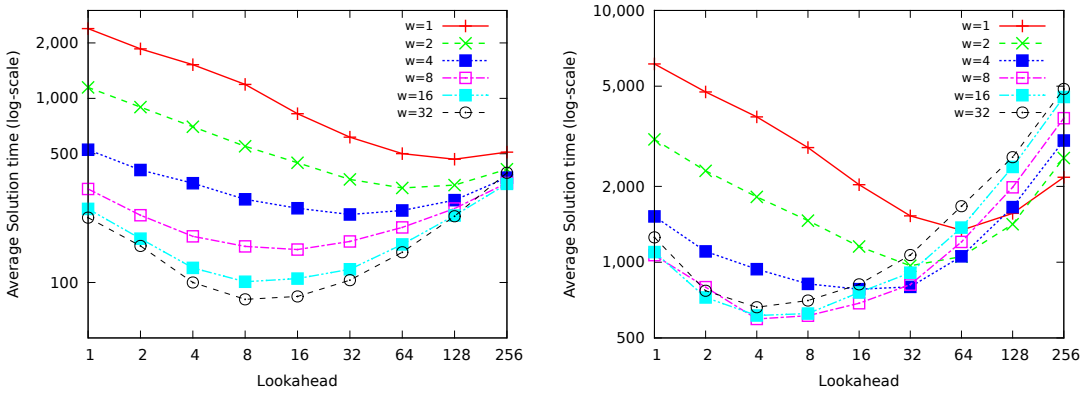


(a) Game Maps (b) Mazes

FIGURE 2.4. Solution cost versus lookahead parameter (k) obtained by wLSS-LRTA* in game maps and mazes.

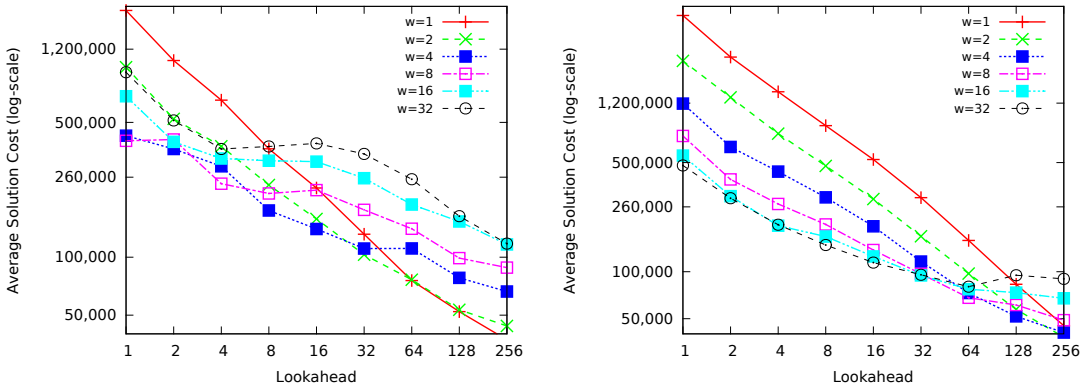
2.5.2. Weighted Update

We evaluated wLSS-LRTA* with six weight values $\{1, 2, 4, 8, 16, 32\}$ and nine lookahead values $\{1, 2, 4, 8, 16, 32, 64, 128, 256\}$. We report the solution cost and the total time per test case obtained by the algorithm for different weight and lookahead values. Regarding the time per search episode, it is known that the time per search episode increases when the lookahead increases (Hernández & Meseguer, 2007; Koenig & Sun, 2009). On the other hand, when different weights are used for a fixed lookahead value, the time per search episode does not increase.



(a) Game Maps (b) Mazes

FIGURE 2.5. Total Time versus lookahead parameter (k) obtained by wLSS-LRTA* in game maps and mazes.



(a) 2-pixel-wide corridors (b) 8-pixel-wide corridors

FIGURE 2.6. Solution cost versus lookahead parameter (k) obtained by wLSS-LRTA* mazes of different width.

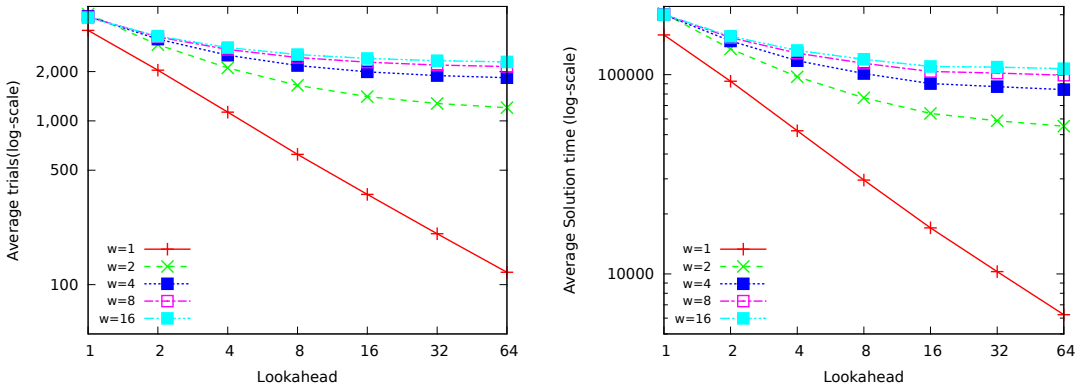
Figures 2.4 and 2.5 show the results for wLSS-LRTA*. The following can be observed in the plots.

- In Game maps we observe that when the weight value increases the solution cost decreases, for all tested lookahead values, except for $w = 32$ and lookahead equal to 256. In Maze maps we observe a similar behavior for smaller lookahead values, but for larger lookahead values the solution cost increases as the value of w is increased. This behavior can be explained by the fact that wLSS-LRTA* may increase the h-values of an entire region of states

that is limited by the walls of a corridor. This in practice produce a blockage in the corridor, which we refer to as an h-blockage. Once the h-blockage has been established, regions outside the blockage become more attractive (due to its lower h-value). In addition, since these maze maps are acyclic, when an h-blockage in a corridor may indeed "cut" all paths to the goal. h-blockages are created more easily when the corridors are narrower. This is shown in Figure 2.6, in which the phenomenon is clearly observed in mazes with 2-pixel-wide corridors while in mazes with 8-pixel-wide corridors curves resemble more those of game maps. Hernández and Baier (Hernández & Baier, 2012) show a similar effect on the performance of depression-avoiding Real-Time Search algorithms, and analyze in more detail how narrow corridors may become blocked during search.

- When the lookahead value increases the solution cost decreases for all weight values tested.
- For all weight values tested the total search time behaves similar to the original LSS-LRTA* algorithm (Koenig & Likhachev, 2006; Koenig & Sun, 2009): first total search time decreases when the lookahead value increases and until a minimum is reached. From then on total search time increases as the lookahead value increases. On the other hand, in Game maps when the weight value increases, total search time decreases. In Maze maps such a behavior is similar to those observed in maps but for larger lookahead values total search time is higher for higher weights. This can be also explained by the h-blockage phenomenon described above.

The results show that the use of weights in wLSS-LRTA* have very positive consequences. The solution cost and the total search time are improved sometimes by orders of magnitude without consuming adding additional per search per episodes.



(a) Number of Trials to Convergence (b) Total Time to Convergence

FIGURE 2.7. Number of trials and total time versus lookahead parameter (k) obtained by wLSS-LRTA* in a game maps to convergence to a w-optimal path

2.5.3. Convergence Evaluation

We evaluated wLSS-LRTA* run to convergence on 200 random problems on one Game map (brc202d). We report the number of trials and the total time to convergence per test case obtained by the algorithm for different weight and lookahead values. Figure 2.7 shows the results. The following can be observed in the plots.

- When the weight value increases the number of trials and the total time increases for all lookahead values tested.
- When the lookahead value increases the number of trials decreases for all weight values tested.
- When the lookahead value increases the total time decreases for all weight values tested.

In practice most solutions returned are optimal. In only two problems the total cost differs from optimal by cost at most 1.

The results show that the use of weights in wLSS-LRTA* may have negative concecuences in the performance. The number of trials and the total search time obtained

by the original algorithm ($w = 1$) are better than the results obtained by the algorithm with w greater than one. Due to this, in the following sections we focus the experimental evaluation in the first trial only.

2.6. Incorpoating Weighted Update into Other Real-Time Heuristic Search Algorithms

Weighted update, the technique that was just shown in the previous section as yielding best results when incorporated to LSS-LRTA*, is applicable to any algorithm that updates the h-values using the equation:

$$h(s) := \min_{t \in \partial L} d_L(s,t) + h(t), \quad \text{for every } s \in L$$

where L is a set of states. We now briefly describe two algorithms proposed in the literature that use this equation for its learning phase, and describe the empirical results obtained on the same experimental setting we described in the previous section.

2.6.1. LRTA*-LS

LRTA*-LS (Hernández & Meseguer, 2007)—shown in Algorithm 6—is a real-time heuristic search algorithm that differs from LSS-LRTA* mainly in how it builds the region of states for the update. In each iteration, LRTA*-LS builds a *learning space*, denoted by I in Alg. 6. It does so by running a breadth-first search from the current state, which will add a state s to I if $h(s)$ is not justified by any of its successor states outside of I. Just like LSS-LRTA*, LRTA*-LS updates the h-values of states in I to the maximum possible value that preserves consistency (Eq. 2.1; Alg. 4). Finally, in the movement step, like LRTA*, it moves the agent to the best neighbor.

Note that both LSS-LRTA* and LRTA*-LS update equations are exactly the same and as such the same algorithm (Modified Dijkstra's Algorithm) is used to update the region. To implement lookahead update in LRTA*-LS we simply modify the Dijkstra's algorithm to multiply arc costs by w. In addition, we modify the way states are added

Input: A search problem P, a heuristic function h, and a parameter k

while *the agent has not reached a goal state* **do**

  Update: Build a set of states I as follows. Initialize a queue Q as containing the current state. Let $I := \emptyset$. Now, until $|I| = k$ or Q is empty, pop an element s from Q, and if $h(s) < c(s,t) + h(t)$ for every $t \in Succ(s) \setminus I$, then (1) add s to I, and (2) push to Q all successors of s not in I. Finally, update the h-values of every state $s \in I$ such that

$$h(s) := \min_{t \in \partial I} d_I(s,t) + h(t). \tag{2.26}$$

  Lookahead: Let the current state be s. Set *next* to $\arg\min_{t \in Succ(s)} c(s,t) + h(t)$.

  Movement: Move the agent to *next*, observe the environment and update the costs of the search graph when new obstacles are found.

end

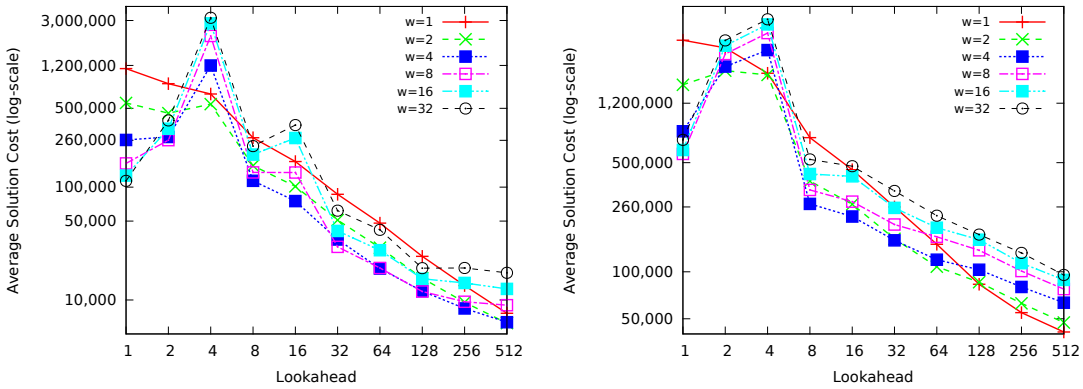
Algorithm 6: LRTA*-LS

to the learning space accordingly, by considering wc instead of c in the inequality used as a condition in the update step. We call the resulting algorithm wLRTA*-LS. Notice that Theorem 2.3 applies directly to wLRTA*-LS.

2.6.1.1. Empirical Evaluation of wLRTA*-LS

We use the same experimental setting used in the wLSS-LRTA* evaluation. Figure 2.8 shows the results for wLRTA*-LS. The following can be observed in the plots:

- In Game maps when the weight value increases from 1 to 4 average solution cost decreases (except for lookahead equal to 4 where improves only until lookahead = 2). When the weight increases over 4 the solution cost increases. This is due to the fact that wLRTA*-LS does not move always to the best state in the learning space. Section 2.7 discusses this in more detail.
- When the lookahead value increases the solution cost tends to decrease for almost all weight values tested. There are a few exceptions; for example, when the lookahead parameter is equal to 4. Such a behavior will be discussed in detail in Section 2.7.



(a) Game Maps (b) Mazes

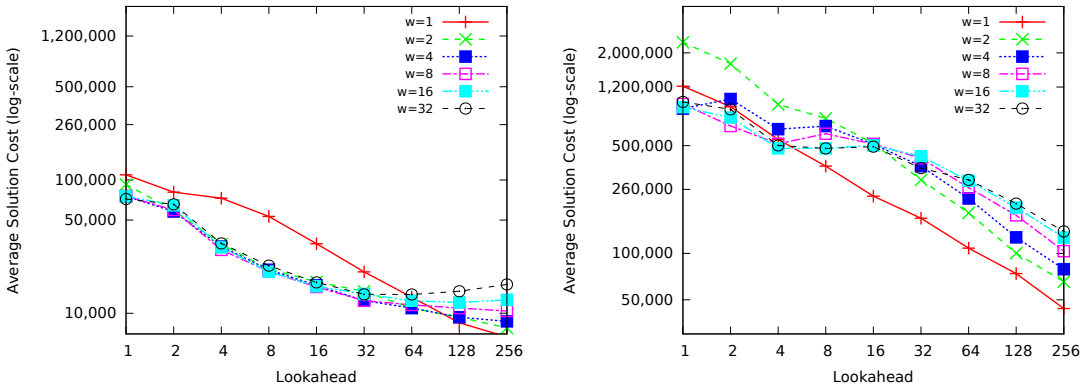
FIGURE 2.8. Solution cost versus lookahead parameter (k) obtained by wLRTA*-LS in game maps and mazes.

- We do not include plots for total time since the total search time behaves similar to LRTA*-LS (Hernández & Meseguer, 2007): the total search time first decreases when the lookahead value increases (except for some small lookahead values where total search time increases, for instance lookahead value equal to 4) and from certain lookahead values the total search time smoothly increases when the lookahead value increases. When the weight value increases the total search time decreases for small weight values, and from greater weight values, the total search time increases.

The results show that the use of weights in wLRTA*-LS could have positive consequences. The solution cost and the total search time are improved for some weight values.

2.6.2. daLSS-LRTA*

Depression-Avoiding LSS-LRTA* (Hernández & Baier, 2012), daLSS-LRTA*, is variant of LSS-LRTA* which, in each movement step, instead of moving the agent to the state of lowest f value in *Open*, it moves it to the state of lowest f-value among those whose h value has changed the least. More specifically, after the A* lookahead



(a) Game Maps (b) Mazes

FIGURE 2.9. Solution cost versus lookahead parameter (k) obtained by $w$daLSS-LRTA* in game maps and mazes.

returns, the state chosen to move to, $next$, is given by:

$$next := \arg\min_{t \in \Gamma} d_{Closed}(s_{current}, t) + h(t),$$

where $\Gamma = \{s \in Open \mid \Delta(s) \leq \Delta(t), \text{for all } t \in Open\}$, and $\Delta(s)$ denotes $h(s) - h_0(s)$, where $h_0(s)$ is the initial h-value of s. daLSS-LRTA* has been shown to have good performance compared to LSS-LRTA* in path-finding tasks (Hernández & Baier, 2012), close to one order of magnitude when time constraints are tight. As such we considered interesting to see whether incorporating weights could improve this algorithm.

As is the case with all other algorithms, $w$daLSS-LRTA* is obtained from daLSS-LRTA* by simply multiplying the costs of the graph by w while running the Modified Dijkstra's algorithm. Notice that Theorem 2.3 does not apply directly to $w$daLSS-LRTA*. We left a formal proof out of the scope of this paper. In our experiment the agent always found the solution and thus we conjeture that $w$daLSS-LRTA* always terminate.

2.6.2.1. Empirical Evaluation of $w$daLSS-LRTA*

We use the same experimental setting used in the evaluation of wLSS-LRTA*. Figure 2.8 shows the results for $w$daLSS-LRTA*. The following can be observed in the plots:

- In game maps the use of weights improves performance in up to about 50% depending on the lookahead value used except when the lookahead value is greater than 64. Unlike in the previous algorithms, we do not observe significant benefit or disadvantage of using a weight value greater than 2 over using $w = 2$.
- In maze maps we only observe a minor improvement for lookahead value equal to 1, for most weight values. For higher lookahead values we observe a decrease in performance. This may be explained, again, by the weighted update's tendency to create blockages.
- We do not include total time plots because for all weight values tested the total search time behaves similar to the original algorithm (daLSS-LRTA* weight = 1)(Hernández & Baier, 2012): the total search time first decreases when the lookahead value increases and from a certain lookahead values the total search time increases when the lookahead value increases.

In summary, the results show that the use of weights in $w$daLSS-LRTA* have positive consequences in Game maps only. The solution cost and the total search time are improved for most of the weight and lookahead values tested.

2.7. Discussion

Two of the results shown in the previous section could be seen as rather surprising. First and foremost is the fact that plugging Weighted A*—an algorithm that yields good results in Heuristic Search—into Real-Time Heuristic Search algorithms yields poor results. The second is the anomalous behavior exhibited by LRTA*-LS for some values of the k parameter. We analyze both of these issues below.

2.7.1. Weighted A* in Real-Time Heuristic Search

As seen in the last section, LSS-LRTwA* has very poor performance as w increases. Such finding is interesting, as it shows that the benefits of weighted A* cannot be immediately leveraged into Real-Time Heuristic Search.

Even though we do not have formal proofs that show why this happens we think two factors may play an important role. The first factor comes from a known property of Weighted A*: the solution cost typically increases as w increases. As such, it should not be surprising that worse intermediate solutions are returned by each of the calls in the lookahead step, which could explain why more costly solutions are found. Furthermore, since the number of expanded nodes in each search episode is *constant* (it is equal to the k parameter), using Weighted A* does not yield any time benefits either per lookahead step. Since the solution found is longer, more iterations of the algorithm are needed, which explains the increase in total time.

The poor performance of LSS-LRTwA* may also be explained by the quality of learning, which, at least in some parts of the learning space, is worse than when using LSS-LRTA*. In fact, assume the agent is in state s and that we want to construct a learning space of size k around s. The next theorem states that the region built by expanding k nodes using A* is the one that *maximizes* the increase of the h-value of s. In other words, such a region maximizes learning in s.

Theorem 2.7. *Let k be a natural number, h be a consistent heuristic, and $s \in S$ be such that s_g is at least at k edges away from s. Furthermore, let $\Delta_h(s, L)$ denote the amount by which the h-value of s increases after learning (i.e. $\Delta_h(s, L) = \min_{t \in \partial L}[h(t) + d_L(s, t)] - h(s)$). Consider the following optimization problem:*

$$\max_{s \in L} \Delta_h(s, L), \text{ subject to } s \in L \text{ and } |L| = k.$$

Then the maximum is attained when L is the Closed list of an A search just after Closed reaches size k.*

PROOF. Let L be set of states in the Closed list of an A* started at state s when Closed reaches size k. Notice that Closed actually reaches size k because s_g more than k steps aways and thus s_g cannot be in Closed. Recall that A* with consistent heuristics does not reopen states thus after k iterations of A*'s main loop the Closed list indeed has size k.

We now prove that if we chose a region of states different from L then the amount of learning given by such a region is not greater than the amount obtained given by L. Indeed, let I be such that $I \neq L$, such that $s \in I$, and such that $|I| = k$. Let $t_1 = s, t_2, ..., t_k$ be the order in which the elements were inserted in L by A*. Let t be the element of $\{t_1, .., t_k\}$ with lowest index such that $t \notin I$. By definition, $t \in \partial I$. Let f^* be the f-value of state that would have been expanded in the $(k+1)$-th iteration of the A* run. Note that since h is consistent,

$$f^* = \min_{u \in \partial L} h(u) + d_L(s, u),$$

Also due to the fact that h is consistent, we have that

$$f(t) \leq f^*, \tag{2.27}$$

because the f-values of expanded states cannot decrease through execution using consistent heuristics.

On the other hand, since $t \in \partial I$,

$$\min_{u \in \partial I}[h(u) + d_I(s, u)] \leq h(t) + d_I(s, t) \tag{2.28}$$

In addition, $d_L(s, t) = d_I(s, t)$ because t was expanded via an optimal path (because h is consistent) and I contains such a path. Then we have $f(t) = h(t) + d_I(s, t)$, which allows us to write, using Inequation (2.28), that:

$$\min_{u \in \partial I}[h(u) + d_I(s, u)] \leq f(t) \tag{2.29}$$

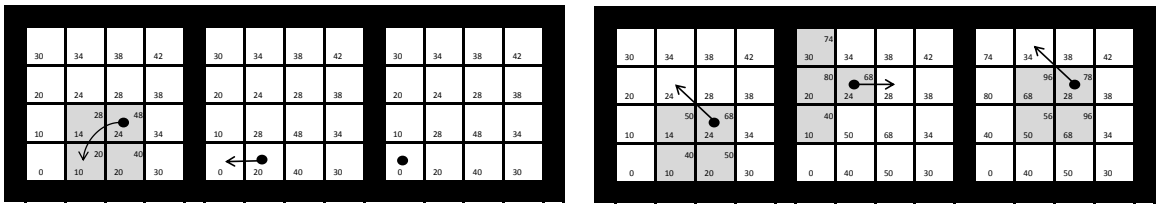


FIGURE 2.10. First three iterations of two runs of wLRTA*-LS with $w = 2$ (left) and $w = 4$ (right) with parameter $k = 4$ in a $\mathbf{4} \times \mathbf{4}$ grid. The grid is 8-connected, horizontal moves have cost 10, and diagonal movements have cost 14. Each cell shows the h-value before the update step in the lower-left corner, and the h-value after update in the upper right corner. The black dot shows the current position of the agent and the arrow shows the next cell chosen by the algorithm. We assume that states are added to the queue Q in clockwise order starting at 6 o'clock. The goal state is the state with heuristic value 0. We observe that when $w = 2$ it takes 2 movements to reach the goal. On the other hand, we observe that when $w = 4$ the agent moves away from the goal. In fact, when $w = 4$ it takes the agent 7 moves to reach goal.

Adding Inequations (2.27) and (2.29), substituting f^* and $f(t)$ and subtracting $h(s)$ we obtain:

$$\Delta_h(s, I) = \min_{u \in \partial I}[h(u) + d_I(s, u)] - h(s) \leq \min_{u \in \partial L}[h(u) + d_L(s, u)] - h(s) = \Delta_h(s, L),$$

which is what we wanted to establish. □

Since Weighted A*, given a bound of k expansions, generates a region *different* from A*, we can infer that in some cases the learning performed in the current state is of inferior quality. This suggests that part of the poor performance of LSS-LRTwA* may be explained by its poor learning, since it is known that stronger learning yields better performance in Real-Time Heuristic Search (see e.g., (Koenig & Likhachev, 2006)).

2.7.2. wLRTA*-LS for Low Values of k

Interestingly the decrease in the performance of wLRTA*-LS for $k = 2$ and $k = 4$, in Game maps, can be explained in very simple terms. Such a bad behavior should be expected whenever k is lower than the branching factor of the search problem. Indeed when that is the case, wLRTA*-LS will update the heuristic value of only *some* of

the neighbors of the current state. Since in the movement phase wLRTA*-LS chooses the position to move to from its immediate *neighbors* it could be the case that the h-values of those neighbors are quite incomparable, because only some of them have been updated using w. In these situations it could be that the algorithm chooses to move away from the goal. Figure 2.10 illustrates how this phenomenon may affect performance in 8-connected grid navigation.

wLSS-LRTA* does not have this problem, because it always chooses to move to the best state in *Open*. Since the h-values of those states is *not* updated, they are comparable. This observation suggests that wLRTA*-LS movement step could me modified in order to move to the state from ∂I that justifies the h-value of the current state. We decided to leave the implementation of such an algorithm out of the scope of this paper. We conjecture that it will not exhibit performance degradation for low values of k.

2.8. Conclusions and Future Work

We proposed two approaches that allow exploiting weights in Real-Time Heuristic Search algorithms: weighted lookahead and weighted update. We incorporated weighted update to LSS-LRTA*, a standard Real-Time Heuristic Search algorithm, and showed it does not yield performance improvements. On the other hand, we incorporated weighted update to LSS-LRTA* and showed it may yield superior performance of up to one order of magnitude in some path-finding benchmarks. Performance gains were also observed when incorporating the technique to other algorithms like LRTA*-LS and daLSS-LRTA*, although improvements on the latter algorithm are less impressive.

In addition we thoroughly analyzed some desirable properties of wLSS-LRTA*. In particular, we prove that it terminates when a solution exists. Furthermore we prove wLSS-LRTA* finds w-optimal solutions on convergence, but we also found bounds that can be much tighter, and, indeed, under certain conditions, we found a bound on solution quality that does not depend on w but only on features of the search graph.

Future work includes the incorporation of this techniques to other Real-Time Heristic Search algorithms that do not use the Modified Dijkstra algorithm, like, for example, RTAA* (Koenig & Likhachev, 2006). Another line of research has to do with how to determine good values of w and whether or not good policies for adjusting the weight dynamically can be devised. Regarding convergence behavior, it seems necessary to study whether or not using dynamic weights will produce faster convergence results at the expense of sacrificing solution quality.

References

Bulitko, V. (2004). Learning for adaptive real-time search. *Computing Research Repository*, *cs.AI/0407016*. Available from `http://arxiv.org/abs/cs.AI/0407016`

Bulitko, V., Björnsson, Y., Sturtevant, N., & Lawrence, R. (2010). *Real-time heuristic search for game pathfinding. in book: Applied research in artificial intelligence for computer games*. Springer USA.

Bulitko, V., Björnsson, Y., Sturtevant, N., & Lawrence, R. (2011). Real-time heuristic search for game pathfinding. Springer.

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

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms, second edition*. The MIT Press and McGraw-Hill Book Company.

Edelkamp, S., & Schrödl, S. (2011). *Heuristic search: Theory and applications*. Morgan Kaufmann.

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

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

Hernández, C., & Meseguer, P. (2007). Improving LRTA*(k). In *Proceedings of the 20th International joint Conference on Artificial Intelligence (IJCAI)* (p. 2312-2317).

Koenig, S., & Likhachev, M. (2006). Real-time adaptive A*. In *Proceedings of the 5th International joint Conference on autonomous agents and multi agent systems (AAMAS)* (pp. 281–288).

Koenig, S., & 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.

Koenig, S., Tovey, C. A., & Smirnov, Y. V. (2003). Performance bounds for planning in unknown terrain. *Artificial Intelligence*, *147*(1-2), 253-279.

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

Likhachev, M., Gordon, G. J., & Thrun, S. (2003). ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *Proceedings of the 16th Conference on advances in neural information processing systems (NIPS).* Vancouver, Canada.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, *1*(3), 193-204.

Richter, S., Helmert, M., & Westphal, M. (2008). Landmarks revisited. In *Proceedings of the 23rd aaai Conference on Artificial Intelligence (AAAI)* (p. 975-982). Chicago, IL.

Richter, S., Thayer, J. T., & Ruml, W. (2010). The joy of forgetting: Faster anytime search via restarting. In *Proceedings of the 20th International Conference on automated planning and Scheduling (ICAPS)* (p. 137-144).

Rivera, N., Baier, J. A., & Hernández, C. (2013). Weighted real-time heuristic search. In *Proceedings of the 10th International joint Conference on autonomous agents and multi agent systems (AAMAS).* St. Paul, MN.

Rivera, N., Illanes, L., Baier, J. A., & Hernández, C. (2013). Reconnecting with the ideal tree: An alternative to heuristic learning in real-time search. In *Proceedings of the 6th symposium on combinatorial search (SoCS).*

Shimbo, M., & Ishida, T. (2003). Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, *146*(1), 1-41.

Wilt, C. M., & Ruml, W. (2012). When does weighted a* fail? In *Proceedings of the 5th symposium on combinatorial search (SoCS).*

Zelinsky, A. (1992). A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation*, *8*(6), 707-717.