Journal of Artificial Intelligence Research 56 (2016) 547-571 Submitted 01/16; published 08/16

Research Note

Time-Bounded Best-First Search for Reversible and Non-reversible Search Graphs

Carlos Hernández CARLOS.HERNANDEZ.U@UNAB.CL
Departamento de Ciencias de la Ingeniería,
Universidad Andrés Bello,
Santiago, Chile

Jorge A. Baier JABAIER@ING.PUC.CL
Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Santiago, Chile

Roberto Asín RASIN@UCSC.CL
Departamento de Ingeniería Informática
Universidad Católica de la Santísima Concepción
Concepción, Chile

Abstract

Time-Bounded A* is a real-time, single-agent, deterministic search algorithm that expands states of a graph in the same order as A* does, but that unlike A* interleaves search and action execution. Known to outperform state-of-the-art real-time search algorithms based on Korf's Learning Real-Time A* (LRTA*) in some benchmarks, it has not been studied in detail and is sometimes not considered as a "true" real-time search algorithm since it fails in non-reversible problems even it the goal is still reachable from the current state. In this paper we propose and study Time-Bounded Best-First Search (TB(BFS)) a straightforward generalization of the time-bounded approach to any best-first search algorithm. Furthermore, we propose Restarting Time-Bounded Weighted A* (TB$_R$ (WA*)), an algorithm that deals more adequately with non-reversible search graphs, eliminating "backtracking moves" and incorporating search restarts and heuristic learning. In non-reversible problems we prove that TB(BFS) terminates and we deduce cost bounds for the solutions returned by Time-Bounded Weighted A* (TB(WA*)), an instance of TB(BFS). Furthermore, we prove TB$_R$ (WA*), under reasonable conditions, terminates. We evaluate TB(WA) in both grid pathfinding and the 15-puzzle. In addition, we evaluate TB$_R$ (WA*) on the racetrack problem. We compare our algorithms to LSS-LRTWA*, a variant of LRTA* that can exploit lookahead search and a weighted heuristic. A general observation is that the performance of both TB(WA*) and TB$_R$ (WA*) improves as the weight parameter is increased. In addition, our time-bounded algorithms almost always outperform LSS-LRTWA* by a significant margin.

1. Introduction

In many search applications, time is a very scarce resource. Examples range from video game path finding, where a handful of milliseconds are given to the search algorithm controlling automated characters (Bulitko, Björnsson, Sturtevant, & Lawrence, 2011), to highly dynamic robotics (Schmid, Tomic, Ruess, Hirschmüller, & Suppa, 2013). In those settings, it is usually assumed that a standard search algorithm will not be able to compute a complete solution before an action is required, and thus execution and search must be interleaved.

©2016 AI Access Foundation. All rights reserved.

Time-Bounded A* (Björnsson, Bulitko, & Sturtevant, 2009) is an algorithm suitable for searching under tight time constraints. In a nutshell, given a parameter k, it runs a standard A* search towards the goal rooted in the initial state, but after k expansions are completed, a move is performed and then search, if still needed, is resumed. The move is computed as follows. If the agent is in the path π found by A* from the root node to the best node b in the search frontier then the agent is moved towards b following the path π. Otherwise, it performs "backtracking move", returning the agent to its previous state. The algorithm always terminates with the agent at the goal state, if the problem has a solution.

Time-Bounded A* is an algorithm that is relevant to the real-time search community. It is significantly superior to well-known real-time heuristic search algorithms in some applications. Indeed Hernández, Baier, Uras, and Koenig (2012) showed it significantly outperforms state-of-the-art real-time heuristic search algorithms such as RTAA* (Koenig & Likhachev, 2006) and daRTAA* (Hernández & Baier, 2012) in pathfinding.

Being a relatively new algorithm, Time-Bounded A* has not been studied deeply in the literature. One of the reasons for this is perhaps its inability to adequately deal with non-reversible problems. Indeed, in non-reversible problems any real-time search algorithm will fail as soon the algorithm has led the agent to a dead-end state; i.e., one from which the goal is unreachable. Time-Bounded A*, however, has an additional failure condition: it will always fail as soon as a backtrack move is required over an unreversible action. Thus the class of problems it cannot solve is more limited compared to other real-time search algorithms, like, for example, the well-known LRTA* (Korf, 1990). For this reason, Time-Bounded A* is sometimes excluded from experimental comparisons with real-time search algorithms (see e.g. Burns, Ruml, & Do, 2013, p. 725).

In this paper we extend the time-bounded search approach in two directions. As already noted by their authors (Björnsson et al., 2009), the time-bounded approach is not limited just to A*. A first contribution of this paper is a study of what are the implications of using other search algorithms instead of A*. Specifically, we generalize Time-Bounded A* to Time-Bounded Best-First Search. In general, if $\mathcal{A}$ is an instance of Best-First Search, we call TB($\mathcal{A}$) the algorithm that results from applying the time-bounded approach to $\mathcal{A}$. A second contribution of this paper is an extension to the time-bounded search approach that allows the algorithm to deal more adequately with non-reversible problems. The algorithm we propose here, Restarting Time-Bounded Weighted A*—which we call TB$_R$(WA*)—, can be seen as lying in the middle ground between time-bounded algorithms and learning-based real-time search algorithms like Korf's Learning Real-Time A* (LRTA*) (1990). In fact, TB$_R$ (WA*) restarts search from the current state when a backtracking move is not available and *updates* the heuristic function.

We carry out a theoretical analysis of both Time-Bounded Weighted A* (TB(WA*)), an instance of TB(BFS), and of TB$_R$ (WA*). For TB(WA*) we establish upper and lower bounds for the solution cost. Our cost bound establishes that, in some domains, the solution cost may be reduced *significantly* by increasing w without increasing search time; hence, in contrast to what is well-known about Weighted A* when solving offline search problems, we might obtain *better* solutions by increasing the weight. This result is important since it suggests that TB(WA*) (with $w > 1$) should be preferred to TB(A*) in domains in which WA* runs faster than A*. While WA* does not always run faster than A* (see e.g., Wilt & Ruml, 2012), it is known that it does in many situations.

Experimentally, we evaluate TB(WA*) on pathfinding benchmarks and in the 15-puzzle, and TB$_R$ (WA*) on the racetrack problem. In all three benchmarks we observe performance improvement as w is increased. In addition, we observe TB(WA*) is significantly superior to both TB(A*) and LSS-LRTWA* (Rivera, Baier, & Hernández, 2015), a real-time search algorithm that can use weighted heuristics.

This paper extends work that appears in conference proceedings (Hernández, Asín, & Baier, 2014), by including an empirical analysis on new benchmarks (Counter Strike Maps, the racetrack, and the 15-puzzle), by extending pathfinding experiments with 16-neighbor connectivity, by providing a lower bound for the cost of the solution returned by TB(WA*) (Theorem 2, below), and by introducing, analyzing, and evaluating TB$_R$ (WA*).

The rest of the paper is organized as follows. We start by describing the background needed for the rest of the paper. Then we describe TB(BFS) and TB$_R$ (BFS), including a formal analysis of their properties. Then we describe the experimental results, and finish with a summary and perspectives for future research.

2. Background

Below we describe the background for the rest of the paper.

2.1 Search in Reversible and Non-reversible Environments

A search graph is a tuple $G = (S, A)$, where S is a finite set of states, $A \subseteq S \times S$ is a set of edges which represent the actions available to the agent in each state. A path over graph (S, A) from s to t is a sequence of states $\pi = s_0 s_1 \cdots s_n$, where $(s_i, s_{i+1}) \in A$, for all $i \in \{0, \ldots, n-1\}$, $s_0 = s$, and $s_n = t$. We say that t is a *successor* of s if (s, t) is an edge in A. Moreover, for every $s \in S$ we define $Succ(s) = \{t \mid (s, t) \in A\}$.

A *cost function* c for a search graph (S, A) is such that $c : A \to \mathbb{R}^+$; i.e., it associates an action with a positive cost. The *cost of a path* $\pi = s_0 s_1 \cdots s_n$ is $c(\pi) = \sum_{i=0}^{n-1} c(s_i, s_{i+1})$, i.e. the sum of the costs of each edge considered in the path. A cost-optimal path from s to t is one that has lowest cost among all paths between s and t; we denote this cost by $c^*(s, t)$. In addition, we denote by $c^*_T(s, t)$ the cost of a cost-optimal path between s and t that visits states only in T, that is, a cost-optimal path $\pi = s_1 s_2 \ldots s_n$ such that $s = s_1$, $s_n = t$, and $s_i \in T$, for all $i \in \{2, \ldots, n-1\}$.

A search problem is a tuple $(S, A, c, s_{start}, s_{goal})$ where $G = (S, A)$ is a search graph, s_{start} and s_{goal} are states in S, and c is a cost function for G. A search graph $G = (S, A)$ is *reversible* if A is symmetric; that is, whenever $(s, t) \in A$ then $(t, s) \in A$. A search problem is reversible if and only if its search graph is reversible. Consequently, problem is non-reversible if its search graph contains an action (s, t) but does not contain an action (t, s).

A solution to a search problem is a path from s_{start} to s_{goal}.

2.2 Best-First Search

Best-First Search (BFS) (Pearl, 1984) encompasses a family of search algorithms for static environments which associate an evaluation function $f(s)$ with every state s. The priority is such that $f(s) < f(t)$ when s is viewed as a more promising node than t. BFS starts off by initializing the priority of all states in the search space to infinity, except for s_{start}, for which the priority is set to $f(s_{start})$. A priority queue *Open* is initialized as containing s_{start}. In each iteration, the algorithm extracts from *Open* the state with lowest priority, s. For each successor t of s it computes the evaluation $f_s(t)$, considering the path that has been found to t from s. If $f_s(t)$ is lower than $f(t)$, then t

is added to $Open$ and $f(t)$ is set to $f_s(t)$. The algorithm repeats this process until s_{goal} is in $Open$ with the lowest priority.

A pseudo code is presented in Algorithm 1. The f-value of state s is usually implemented as an attribute of s, and the $Open$ list is implemented as a priority list. Furthermore, we assume the cost $f_s(t)$ computed in Line 13 is a function of the path to t via s. Thus $f_s(t)$ can only take a finite number of values during an execution of BFS, because it depends on the (finite) number of simple paths that connect the initial state with s.

Algorithm 1: Best-First Search

```
1  $s_{root} \leftarrow s_{current}$
2  $Open \leftarrow \emptyset$
3  foreach $s \in S$ do
4      $f(s) \leftarrow \infty$
5  $f(s_{root}) \leftarrow$ evaluation for $s_{root}$
6  Insert $s_{root}$ in $Open$
7  while $Open \neq \emptyset$ do
8      Let $s$ be the state with minimum $f$-value in $Open$
9      if $s = s_{goal}$ then
10         return $s$
11     Remove $s$ from $Open$
12     foreach $t \in Succ(s)$ do
13         $f_s(t) \leftarrow$ evaluation function for $t$ considering that $t$ is discovered from $s$
14         if $f_s(t) < f(t)$ then
15             $f(t) \leftarrow f_s(t)$
16             $parent(t) \leftarrow s$
17             Insert $t$ in $Open$
18 return "no solution"
```

An instance of Best-First Search is *Weighted A** (WA*) (Pohl, 1970). WA* computes the evaluation function in terms of two other functions, g and h. The g-value corresponds to the cost of the lowest-cost path found so far towards s, and it is implemented as an attribute of s. WA*'s evaluation function is defined as $f(s) = g(s) + wh(s)$, where $g(s)$ is the cost of the lowest-cost path found from s_{start} to s. In addition, h is a non-negative, user-given *heuristic function* such that $h(s)$ estimates the cost of a path from s to s_{goal}. Finally, w is a real number greater than or equal to 1.

The pseudo-code for WA* can be obtained from Algorithm 1 by storing the g-value as an attribute of the state, while the h value is computed by an external function. The resulting pseudo-code appears in Algorithm 2.

A heuristic function h is *admissible* if and only if $h(s) \leq c^*(s, s_{goal})$, for all $s \in S$. Function h is *consistent* if $h(s_{goal}) = 0$, and $h(s) \leq c(s,t) + h(t)$ for every edge (s,t) of the search graph. Consistency implies that if π is a path from s to t then $h(s) \leq c(\pi) + h(t)$, which, in turn, implies admissibility.

BFS's *closed list*—denoted henceforth by $Closed$—is defined as the set of states that are not in $Open$ and that are such that $g(s)$ is not infinity.[1] In other words, it contains the states for which a path is known but that are not being considered for re-expansion.

1. BFS initially sets $f(s)$ to infinity for every s that is not the start node. In WA* this translates to setting $g(s)$ to infinity for all s except from s_{start}.

Algorithm 2: Weighted A*

```
1  $s_{root} \leftarrow s_{current}$
2  $Open \leftarrow \emptyset$
3  foreach $s \in S$ do
4      $g(s) \leftarrow \infty$
5      $f(s) \leftarrow \infty$
6  $g(s_{root}) \leftarrow 0$
7  $f(s_{root}) \leftarrow wh(s_{root})$
8  Insert $s_{root}$ in $Open$
9  while $Open \neq \emptyset$ do
10     Let $s$ be the state with minimum $f$-value in $Open$
11     if $s = s_{goal}$ then
12         return $s$
13     Remove $s$ from $Open$
14     foreach $t \in Succ(s)$ do
15         $g_{s,t} = \min\{g(t), g(s) + c(s,t)\}$
16         if $g_{s,t} < g(t)$ then
17             $g(t) \leftarrow g_{s,t}$
18             $f(t) \leftarrow g(t) + wh(t)$
19             $parent(t) \leftarrow s$
20             Insert $t$ in $Open$
21 return "no solution"
```

If h is admissible, WA* is known to find a solution whose cost cannot exceed $wc^*(s_{start}, s_{goal})$. As such, WA* may return increasingly worse solutions as w is increased. The advantage of increasing w is that search time is usually *decreased* because fewer states are expanded. When $w = 1$, WA* is equivalent to A* (Hart, Nilsson, & Raphael, 1968). Another interesting result generalizes a well-known property of consistent heuristics of the A* algorithm. It is formally stated as follows:

Lemma 1 (Ebendt & Drechsler, 2009) *At every moment during the execution of Weighted A* from state s_{root}, if h is consistent, upon expansion of a state s (Line 14 of Algorithm 2), it holds that $g(s) \leq wc^*(s_{root}, s)$.*

Another instance of Best-First Search is *Greedy Best-First Search* (GBFS). Here f is equal to the user-given heuristic function h. When WA* is used with a sufficiently large value of w, both WA* and GBFS rank nodes in a similar way. Indeed, let f_{GBFS} and $f_{\text{WA*}}$ denote, respectively, the f function for GBFS and WA*. If w is such that it exceeds the g-value of every node ever generated and two nodes s_1 and s_2 have been generated with the same g-value by both algorithms such that $f_{\text{GBFS}}(s_1) = h(s_1) > h(s_2) = f_{\text{GBFS}}(s_2)$, then it will hold that $f_{\text{WA*}}(s_1) > f_{\text{WA*}}(s_2)$. However, even if w is sufficiently large, the reverse is not always true since $f_{\text{WA*}}(s_1) > f_{\text{WA*}}(s_2)$ can hold true when $h(s_1) = h(s_2)$, because the g-value in $f_{\text{WA*}}$ acts in practice as a tie breaker.

2.3 Real-Time Heuristic Search

In real-time search the objective is to solve a search problem subject to an additional *real-time constraint*. Under this constraint, a constant amount of time (independent of problem size) is given to the search algorithm, by the end of which it is expected to perform one or more actions in a sequence. Such a constant is very small in relation to the time that would be required by an offline search algorithm to solve search problem. If after performing actions the agent has not reached the goal, the process repeats. Each iteration of the algorithm can be understood as two consecutive episodes: (1) a search episode, in which a path is computed, and (2) an execution episode, in which the actions in such a path are performed.

Rather than receiving a time limit in seconds, most real-time search algorithms receive a parameter, say k, and guarantee that the computational time taken by the search episode is bounded by a non-decreasing function of k. An example of a real-time search algorithm is Local Search-Space, Learning Real-Time A* (LSS-LRTA*; Algorithm 3) (Koenig & Sun, 2009). It receives a search problem P and a parameter k. In its search episode, it runs a bounded execution of A* rooted at the current state which expands at most k states. Following, it updates the heuristic values of those states in the closed list of the A* run. This update, usually referred to as *learning step*, makes h more informed, and guarantees that the following holds for every s in A*'s closed list:

$$h(s) = \min_{t \in Open} \{c^*_{Closed}(s,t) + h(t)\}. \tag{1}$$

The execution episode performs the actions that appear in the path found by A* from the current state towards the state which has lowest f-value in the open list. In reversible search spaces if h

Algorithm 3: LSS-LRTA*

```
Input: A search problem $P$ and a natural number $k$
1 $s \leftarrow s_{start}$
2 while $s$ is not a goal state do
3     run A* from $s$ until $k$ states are expanded or a goal node the best state in $Open$
4     $best \leftarrow$ state in A*'s closed list with lowest $f$-value
5     for each $s \in Closed$ do
6         update the $h$-value of $s$ such that Equation 1 holds
7     move along the path found by A* between $s$ and $best$
8     $s \leftarrow best$,
```

is initially consistent it can be shown that LSS-LRTA* terminates when the search problem has a solution (Koenig & Sun, 2009). If the search space is non-reversible, however, termination cannot be guaranteed. As we see later, time-bounded algorithms (without restarts) can prove a solution does not exist as well. This property does not hold for algorithms whose search expands nodes whose distance from the current state is bounded, like LSS-LRTA*.

2.4 Comparing Two Real-Time Search Algorithms

One way frequently used in the literature to compare two real-time search algorithms A and B is by comparing the cost of the returned paths when the algorithms are configured is such a way that their search episodes have approximately the same duration. Assume that a real-time search algorithm requires n search episodes to solve a search problem and that its runtime is T. Then we say that the *average time per search episode* for that run is T/n.

To evaluate the relative performance of two algorithms A and B we use a set of benchmark problems $\mathcal{P}$ and a set of algorithm parameters. For each parameter of algorithm A, we obtain and record the average solution cost for all problems in P and the average time per episode. We do likewise with B and then plot the average solution cost versus the average time per episode for each algorithm. If the curve for algorithm A is always on top of the curve for algorithm B we can clearly state that B is superior to A, because B returns better-quality solutions for a comparable search time per episode.

Another approach that has been used to compare real-time search algorithms is the *Game Time Model* (Hernández et al., 2012). In this model, time is partitioned into uniform time intervals. An agent can execute one movement during each time interval, and search and movements are done in parallel. The objective is to move the agent from its start location to its goal location in as few time intervals as possible. The game time model is motivated by video games. Video games often partition time into game cycles, each of which is only a couple of milliseconds long (Bulitko et al., 2011). When using the Game Time Model, the implementation of the real-time search algorithm is modified to stop search as soon as T units of time—where T is a parameter—have passed.

3. Time-Bounded Best-First Search

Time-Bounded A* (TB(A*), Björnsson et al., 2009) is a real-time search algorithm based on A*. Intuitively, TB(A*) can be understood as an algorithm that runs an A* search from s_{start} to s_{goal} that alternates a search phase with an execution phase until the goal is reached. In each search phase a bounded number of states are expanded using A*. In the execution phase there are two cases. If the agent is on the path from s_{start} to the best state in $Open$, then a forward movement on that path is performed. Otherwise, the algorithm performs "backtracking moves" in which the agent is moved to the state from where it came from. The search phase does not execute if a path connecting s_{start} and s_{goal} has already been found. The algorithm terminates when the agent has reached the goal.

Our generalization of TB(A*) is Time-Bounded Best-First Search, which simply replaces A* in TB(A*) by a Best-First Search. Its pseudo code is shown in Algorithm 4. The parameters are a search problem $(S, A, c, s_{start}, s_{goal})$, and an integer k which we refer to as the *lookahead parameter*.

TB(BFS) uses a variable $s_{current}$ to store the current state of the agent. Its *MoveToGoal* procedure (called from *Main*) implements the loop that alternates search and execution. At initialization (Lines 27–28) $s_{current}$ is initialized to s_{start}, and, among other things, BFS's $Open$ list is set to contain s_{start} only. If the goal state has not been reached (represented by the fact that variable $goalFound$ is false), a bounded version of BFS is called (Line 31) that expands k states, and then computes a path from s_{start} to the state in $Open$ that minimizes the evaluation function f. The path is built quickly by following parent pointers, and it is stored in variable $path$. In the execution phase (Lines 32–36), if the current position of the agent, $s_{current}$, is on $path$, then the agent performs the action determined by the state immediately following $s_{current}$ on $path$. Otherwise, a backtracking move is implemented by moving the agent to the parent of s in the search tree of BFS, $parent(s_{current})$. The use of backtracking moves is a mechanism that guarantees that the agent will eventually reach a state in variable $path$ because, in the worst case, the agent will eventually reach s_{start}. As soon as such a state is reached the agent will start moving towards the state believed to be closest to the goal.

Algorithm 4 is equivalent to TB(A*) when BFS is replaced by A*. Finally, we call *Time-Bounded Greedy Best-First Search* (TB(GBFS)) the algorithm that results when we use Greedy Best-First Search instead of BFS.

Note that the length of the path cannot in general be bounded by a constant on the size of the problem. To bound the computation of each search episode we can use the same technique described

Algorithm 4: Time-Bounded Best-First Search

```
procedure InitializeSearch()
    $s_{root} \leftarrow s_{current}$
    $Open \leftarrow \emptyset$
    foreach $s \in S$ do
        $f(s) \leftarrow \infty$
    $f(s_{root}) \leftarrow$ evaluation for $s_{root}$
    Insert $s_{root}$ in $Open$
    $goalFound \leftarrow false$
function Bounded-Best-First-Search()
    $expansions \leftarrow 0$
    while $Open \neq \emptyset$ and $expansions < k$ and $f(s_{goal}) > \min_{t \in Open} f(t)$ do
        Let $s$ be the state with minimum $f$-value in $Open$
        Remove $s$ from $Open$
        foreach $t \in Succ(s)$ do
            Compute $f_s(t)$ considering that $t$ is discovered from $s$.
            if $f_s(t) < f(t)$ then
                $f(t) \leftarrow f_s(t)$
                $parent(t) \leftarrow s$
                Insert $t$ in $Open$
        $expansions \leftarrow expansions + 1$
    if $Open = \emptyset$ then return false
    Let $s_{best}$ be the state with minimum priority in $Open$.
    if $s_{best} = s_{goal}$ then $goalFound \leftarrow true$
    $path \leftarrow$ path from $s_{root}$ to $s_{best}$
    return true
function MoveToGoal()
    $s_{current} \leftarrow s_{start}$
    InitializeSearch()
    while $s_{current} \neq s_{goal}$ do
        if $goalFound = false$ then
            if Bounded-Best-First-Search() = false then return false
        if $s_{current}$ is on path then
            $s_{current} \leftarrow$ state after $s_{current}$ on $path$
        else
            $s_{current} \leftarrow parent(s_{current})$;
        Execute movement to $s_{current}$
    return true
procedure Main
    if MoveToGoal() = true then
        print("the agent is now at the goal state")
    else
        print("no solution")
```

by Björnsson et al. (2009), whereby an additional counter (analogous to k is used to measure the effort for path extraction). This is omitted from the pseudocode for clarity.

3.1 Properties

Now we analyze a few interesting properties of the algorithms we have just proposed. First, just like TB(A*), TB(BFS) always terminates and finds a solution if one exists. This is an important property since many real-time heuristic search algorithms (e.g., LSS-LRTA*) enter an infinite loop on unsolvable problems. Second, we prove an upper and a lower bound on the cost of solutions returned by TB(WA*). This bound is interesting since it suggests that by increasing w one might obtain *better* solutions rather than worse.

Theorem 1 *TB(BFS) will move an agent to the goal state given a reversible search problem P if a solution to P exists. Otherwise, it will eventually print "no solution".*

Proof: Follows from the fact that Best-First Search eventually finds a path towards the goal. This is because of the fact that the search space is finite and that each state can only be inserted into $Open$ a finite number of times. In addition, all moves carried out by the algorithm (including moving from s to $parent(s)$) are executable in a reversible search space. ■

It is important to note that the reason why TB(BFS) will eventually print "no solution" in an unsolvable problem is dependent on the fact that an Open list is used. LSS-LRTA* cannot always detect unsolvable problems because search can only expand a locality around the current state. This is a characteristic of *agent-centered* search algorithms (Koenig, 2001), a class of algorithms that TB(BFS) is not a member of.

The following two lemmas are intermediate results that allow us to prove an upper bound on the cost of solutions obtained with TB(WA*). The results below apply to TB(A*) but to our knowledge Lemma 2 and Theorem 2 had not been proven before for TB(A*).

In the results below, we assume that $P = (S, A, c, s_{start}, s_{goal})$ is a reversible search problem, that TB(WA*) is run with a parameter $w \geq 1$ and that h is an admissible heuristic. Furthermore, we assume that $c^+ = \max_{(u,v)\in A} c(u, v)$, that $c^- = \min_{(u,v)\in A} c(u, v)$, and that $N(w)$ is the number of expansions needed by WA* to solve P. Finally, we assume $k \ll N(w)$ which is a reasonable assumption given that we are in a real-time setting.

Lemma 2 *The cost of the moves incurred by an agent controlled by TB(WA*) before goalFound becomes true is bounded from below by $\lfloor \frac{N(w)-1}{k} \rfloor c^-$ and bounded from above by $\lfloor \frac{N(w)-1}{k} \rfloor c^+$.*

Proof: $N(w) - 1$ states are expanded before $goalFound$ becomes true. If k states are expanded per call to the search procedure, then clearly $\lfloor \frac{N(w)-1}{k} \rfloor$ is the number of calls for which *Best-First-Search* terminates without setting $goalFound$ to true. Each move costs at least c^- and at most c^+, from where the result follows. ■

Now we focus on the cost that is incurred after a complete path is found. The following Lemma is related to a property enjoyed by TB(A*) and stated in Theorem 2 by Hernández et al. (2012).

Lemma 3 *The cost of the moves incurred by an agent controlled by TB(WA*) after goalFound has become true cannot exceed $2wc^*(s_{start}, s_{goal})$.*

Proof: Assume *goalFound* has just become true. Let π be the path that starts in s_{start}, ends in $s_{current}$ and that is defined by following the *parent* pointers back to s_{start}. Path π is the prefix of a path to the lowest f-value state in a previous run of WA* and therefore, by Lemma 1, is such that $c(\pi) < wc^*(s_{start}, s_{goal})$. Now the worst case in terms of number of movements necessary to reach the goal is that $path$ and π coincide only in s_{start}. In this case, the agent has to backtrack all the way back to s_{start}. Once s_{start} is reached, the agent has to move to the goal through a path of cost at most $wc^*(s_{start}, s_{goal})$. Thus the agent may not incur a cost higher than $2wc^*(s_{start}, s_{goal})$ to reach the goal. ■

Now we obtain a lower bound and an upper bound on the solution cost for TB(WA*) which follows straightforwardly from the two previous lemmas.

Theorem 2 *Let C be the solution cost obtained by TB(WA*). Then,*

$$\lfloor \frac{N(w)-1}{k} \rfloor c^- \leq C \leq \lfloor \frac{N(w)-1}{k} \rfloor c^+ + 2wc^*(s_{start}, s_{goal}).$$

Proof: We put together the inequalities implied by Lemmas 2 and 3. ■

A first observation about this result is that it has been shown empirically that in some domains, when w is increased, $N(w)$ may decrease substantially. Gaschnig (1977), for example, reports that in the 8-puzzle $N(1)$ is exponential in the depth d of the solution whereas $N(w)$, for a large w is subexponential in d. In other domains like grid pathfinding, it is well known that using high values for w results in substantial reductions in expanded nodes (see e.g., Likhachev, Gordon, & Thrun, 2003). Thus, when increasing w, both the lower bound and the first term of the upper bound may decrease substantially. The second term of the upper bound, $2wc^*(s_{start}, s_{goal})$, when increasing w, may increase only linearly with w. This suggests that there are situations in which *better*- rather than worse-quality solutions may be found when w is increased. As we see later, this is confirmed by our experimental evaluation.

A second observation about the bounds is that the factor $\lfloor (N(w)-1)/k \rfloor$ decreases as k increases. This suggests that when k is large (i.e., close to $N(w)$), increasing w may actually lead to decreased performance.

Putting both observations together, Theorem 2 suggests that TB(WA*) will produce better solutions than TBA* when k is relatively small in problems in which WA* expands fewer nodes than A* in offline mode. Problems in which WA* does not expand fewer nodes than A* exist (Wilt & Ruml, 2012).

Finally, it is not hard to see that Theorem 2 can be generalized to other algorithms that provide optimality guarantees. Given two search algorithms $\mathcal{A}$ and $\mathcal{B}$ that provide such bounds and whose relative performance is known, the theorem can be used as a predictor of the relative performance of TB($\mathcal{A}$) versus TB($\mathcal{B}$).

3.2 Non-reversible Search Problems via Restarting

In non-reversible problems, well-known real-time heuristic search algorithms such as LSS-LRTA* will fail when, in the execution episode, a state from which there is no path to the goal is visited. Time-bounded algorithms like TB(BFS) will fail under that very same condition but they will *also* fail as soon as a physical backtrack is required over a non-reversible action. This second condition for failure is the reason why sometimes time-bounded algorithms are discarded for use in non-reversible domains. The objective of this section is to propose a time-bounded algorithm that, when used in non-reversible problems, will not fail due to the latter condition, but only due to the former.

Our modification of TB(WA*) for non-reversible problems comes from incorporating into it the two key characteristics of real-time search algorithms like LSS-LRTA*: search restarts and heuristic updates. Indeed, whenever "physical backtracking" is not available, or, more generally, when some predefined "restart condition" holds, our algorithm restarts search. In addition, to avoid getting trapped in infinite loops, our algorithm updates the heuristic using the same update rule of LSS-LRTA*. We call the resulting algorithm Restarting Time-Bounded Weighted A* (TB$_R$ (WA*)).

Algorithm 5 shows the details of TB$_R$ (WA*). Lines 10–12 are the most relevant difference with the previous algorithm. The algorithm restarts search when the agent is not in $path$ and certain "restart condition", which must become true when there is no action leading from the current state ($s_{current}$) to its parent ($parent(s_{current})$).

Algorithm 5: Restarting Time-Bounded Weighted A*

```
1  function MoveToGoal()
2      $s_{current} \leftarrow s_{start}$
3      InitializeSearch()
4      while $s_{current} \neq s_{goal}$ do
5          if $goalFound = false$ then
6              if Bounded-WA*() = false then return false;
7          if $s_{current}$ is on path then
8              $s_{current} \leftarrow$ state after $s_{current}$ on $path$
9              Execute movement to $s_{current}$
10         else if restart condition holds then
11             Update heuristic function $h$ using LSS-LRTA* update rule (Equation 1)
12             InitializeSearch()
13         else
14             $s_{current} \leftarrow parent(s_{current})$;
15             Execute movement to $s_{current}$
16     return true
17 procedure Main
18     if MoveToGoal() = true then
19         print("the agent is now at the goal state")
20     else
21         print("no solution")
```

Note that prior to restarting the algorithm updates the heuristic just as LSS-LRTA* would. This can be implemented with a version of Dijkstra's algorithm. Note that the number of states that may need to be updated may not be bounded by a constant. If needed, we can compute the update in an incremental manner, across several episodes. We refer the reader to the analysis of Koenig and Sun (2009), and Hernández and Baier (2012) for details about the implementation and proofs of correctness.

3.2.1 TERMINATION OF TB$_R$ (WA*)

TB$_R$ (WA*) can be used in both reversible and non-reversible domains. If the heuristic function h is initially consistent and the search graph is strongly connected, the algorithm terminates.

Theorem 3 *Let P be a search problem with a strongly connected search graph. Then TB$_R$ (WA*), run with a consistent heuristic h, finds a solution for P.*

The proof for Theorem 3 depends on some intermediate results, some of which have proofs that appear elsewhere. The following result establishes that if h is consistent, then it remains consistent after being updated.

Lemma 4 *(Koenig & Sun, 2009) If h is consistent it remains consistent after h is updated with Equation 1.*

Another intermediate results says that h cannot decrease after an update following Equation 1.

Lemma 5 *(Koenig & Sun, 2009) If h is initially consistent then $h(s)$, for every s, cannot decrease if h is updated following Equation 1.*

Another intermediate result says that $h(s)$ finitely converges, which intuitively means that even if we wanted to apply an infinite number of updates to h, then from some point on, h will not change anymore.

Definition 1 (Finite Convergence) *A series of functions $\{f_i\}_{i\geq 0}$ finitely converges to function f if there exists an n such that for every $m \geq n$, it holds that $f_m = f$. In addition, we say that a series of functions $\{f_i\}_{i\geq 0}$ finitely converges if there exists a function f to which it finitely converges.*

Lemma 6 *Let h_0 be a consistent heuristic function and P be a strongly connected graph. Let $\sigma = \{h_i\}_{i\geq 0}$ be such that h_{k+1} is the function that results from (1) assigning h_k to h_{k+1} then (2) updating h_{k+1} using Equation 1, for some set $Closed$ and $Open$ generated by a bounded Weighted A* run rooted at an arbitrary state. Then σ finitely converges.*

Proof: A first observation is that $h_k(s)$ is bounded from above by a positive number for every s and every k. Indeed, because of Lemma 4, h_k is consistent, and thus admissible, for every k. In addition, because the problem has a solution, $h_k(s) \leq c^*(s, s_{goal})$, for every s and every k.

A second observation is that the set of h-values that any state s can take is finite, even if σ is infinite. Formally we prove $\mathcal{H}(s) = \{h_k(s) \mid k \geq 0\}$ is a finite set. Indeed, it is not hard to verify by induction (we leave it as an exercise to the reader) that by using Equation 1, for every $k \geq 0$, it holds that $h_k(s) = c(\pi_k^s) + h_0(s')$ for some, possibly empty path π_k^s originating in s and finishing in s'. Now recall that $h_k(s)$ is bounded from above and observe there are only finitely many paths in the graph whose cost is bounded. We conclude that $\mathcal{H}(s)$ is a finite set, for every s.

Now the proof follows by contradiction, assuming σ does not finitely converge. Because σ is non-decreasing (Lemma 5), the only possibility is that σ increases infinitely often. This implies that there is at least one state s such that $\mathcal{H}(s)$ is infinite: a contradiction. We conclude σ finitely converges. ■

Note that the previous lemma is not saying anything about the function that σ converges to; we do not need to know which function is it for the rest of the proof. The last intermediate result is related to the result by Ebendt and Drechsler (2009) that was stated in Section 2.2 (Lemma 1).

Lemma 7 *At every moment during the execution of Weighted A* from state s_{root}, if h is consistent, for every state s in the open list, it holds that $g(s) \leq wc^*_{Closed}(s_{root}, s)$.*

Proof: Let π be a cost-optimal path from s_{root} to s that visits only states in $Closed$. Let s' be the state that precedes s in π. Because s' is part of an optimal path we have:

$$c^*_{Closed}(s_{root}, s') + c(s', s) = c^*_{Closed}(s_{root}, s). \tag{2}$$

Because s is a successor of s', it holds that:

$$g(s) \leq g(s') + c(s', s). \tag{3}$$

Because of Lemma 1, we have that:

$$g(s') \leq wc^*(s_{root}, s'), \tag{4}$$

Inequalities 3 and 4 imply:

$$g(s) \leq wc^*(s_{root}, s') + c(s', s). \tag{5}$$

Because $w > 0$ and $c^*_{Closed} \geq c^*$:

$$g(s) \leq wc^*_{Closed}(s_{root}, s') + wc(s', s) = w(c^*_{Closed}(s_{root}, s') + c(s', s)). \tag{6}$$

Substituting with Equation 2 we have that:

$$g(s) \leq wc^*_{Closed}(s_{root}, s), \tag{7}$$

which finishes the proof. ■

Now we provide a proof of the main result of this section.

Proof (of Theorem 3) : Let us assume the algorithm does not terminate and thus enters an infinite loop. Note this means the algorithm restarts an infinite number of times (otherwise, Weighted A* would eventually find the goal state, allowing the agent to reach the goal). Assume a moment during this infinite execution after h has converged (we know this by Lemma 6), and let $s_1 s_2 \ldots$ be an infinite sequence of states such that s_i is a state where search was restarted. We now prove that for every i, $h(s_i) > h(s_{i+1})$.

Let O denote the contents of the open list exactly when the algorithm expanded s_{i+1}, and $Closed$ denote the contents of the closed list immediately before the heuristic is updated. From Equation 1, the following holds:

$$h(s_i) = c^*_{Closed}(s_i, s_O) + h(s_O), \quad \text{for some } s_O \in O \tag{8}$$

We can rewrite Equation 8 as:

$$wh(s_i) = wc^*_{Closed}(s_i, s_O) + wh(s_O), \tag{9}$$

Let $g(s_O)$ denote the g-value of s_O exactly when s_{i+1} is preferred for expansion over s_O. Now, we prove that $wc^*_{Closed}(s_i, s_O) \geq g(s_O)$. Indeed, if $s_O \in Closed$ this follows from Lemma 1 and from the fact that $c^*_{Closed} \geq c^*$ and $w \geq 1$. On the other hand, if $s_O \in Open$, then we obtain $wc^*_{Closed}(s_i, s_O) \geq g(s_O)$ from Lemma 7. Now we use this fact to write:

$$wh(s_i) \geq g(s_O) + wh(s_O). \tag{10}$$

Because the algorithm preferred to expand s_{i+1} instead of s_O, then $g(s_O) + wh(s_O) \geq g(s_{i+1}) + wh(s_{i+1})$, and hence:

$$wh(s_i) \geq g(s_{i+1}) + wh(s_{i+1}). \tag{11}$$

Finally, because $w > 0$ and $g(s_{i+1}) > 0$ we obtain $h(s_i) > h(s_{i+1})$.

This implies that the sequence of states $s_1 s_2 \ldots$ has strictly decreasing h-values. But because the state space is finite, it must be the case that $s_i = s_j$, for some i and j with $i \neq j$, which would lead to conclude that $h(s_i) > h(s_i)$, a contradiction. ■

4. Experimental Results

This section presents our experimental results. The objective of our experimental evaluation was to understand the effect of the weight configuration on the performance of both TB(WA*) and TB$_R$ (WA*). To that end, we evaluate TB(WA*) in reversible search problems (grid pathfinding and the 15-puzzle), and TB$_R$ (WA*) in a non-reversible problem (the racetrack). For reference, we compare against LSS-LRTWA* (Rivera et al., 2015), a version of LSS-LRTA* that uses Weighted A* rather than A* in the search phase. We used this algorithm since it is among the few real-time search algorithms that are able to exploit weights during search. LSS-LRTWA* is configured to perform a single action in each execution phase.

We decided not to include results for WLSS-LRTA* (Rivera et al., 2015), another real-time search algorithm that exploits weights, for two reasons. First, our new results are focused on relatively large lookahead values (over 128). With these lookahead values, Rivera et al. (2015), in grid-like terrain, observe improvements but not very significant. Second, we observed that, on the 15-puzzle, WLSS-LRTA* yields worse performance as w is increased.

In Section 4.1 we report results in 8- and 16-neighbor grids in a similar manner as was reported in an earlier publication (Hernández et al., 2014). Section 4.2 reports results for 8- and 16-neighbor grids using the Game Time Model (cf. Section 2.4). Section 4.3 reports results on non-reversible maps in a deterministic version of a setting used to evaluate algorithms for the Stochastic Shortest-Path problem (Bonet & Geffner, 2003). Finally, Subsection 4.4 reports results on the 15-puzzle.

The path-finding tasks of Section 4.1 and Section 4.2 are evaluated using 8-neighbor (Bulitko et al., 2011; Koenig & Likhachev, 2005) and 16-neighbor grids (Aine & Likhachev, 2013) (see Figure 5). The costs of the movements are 1, $\sqrt{2}$, and $\sqrt{5}$ for, respectively, orthogonal, diagonal, chess-knight movements. In our implementation the agent cannot "jump over" obstacles. In addition, a diagonal movement (d, d) (for $d \in \{-1, 1\}$) is illegal in (x, y) if either $(x+d, y)$ or $(x, y+d)$ is an obstacle. For 8-neighbor and 16-neighbor grids we use the octile distance and the Euclidean distance as heuristic values, respectively. All experiments were run on an Intel(R) Core(TM) i7-2600 @ 3.4Ghz machine, with 8Gbytes of RAM running Linux. All algorithms have a common code base and use a standard binary heap for *Open*. Ties in *Open* are broken in favor of larger g-values; we do not have a rule for breaking further ties.

4.1 Results in 8-Neighbor and 16-Neighbor Grid Maps

We evaluated the algorithms considering solution cost and runtime, as measures of solution quality and efficiency, respectively, for several lookahead and weight values.

We used all 512×512 maps from the video game *Baldurs Gate* (BG), all the Room maps (ROOMS), and all maps of different size from the *Starcraft* (SC) available from N. Sturtevant's

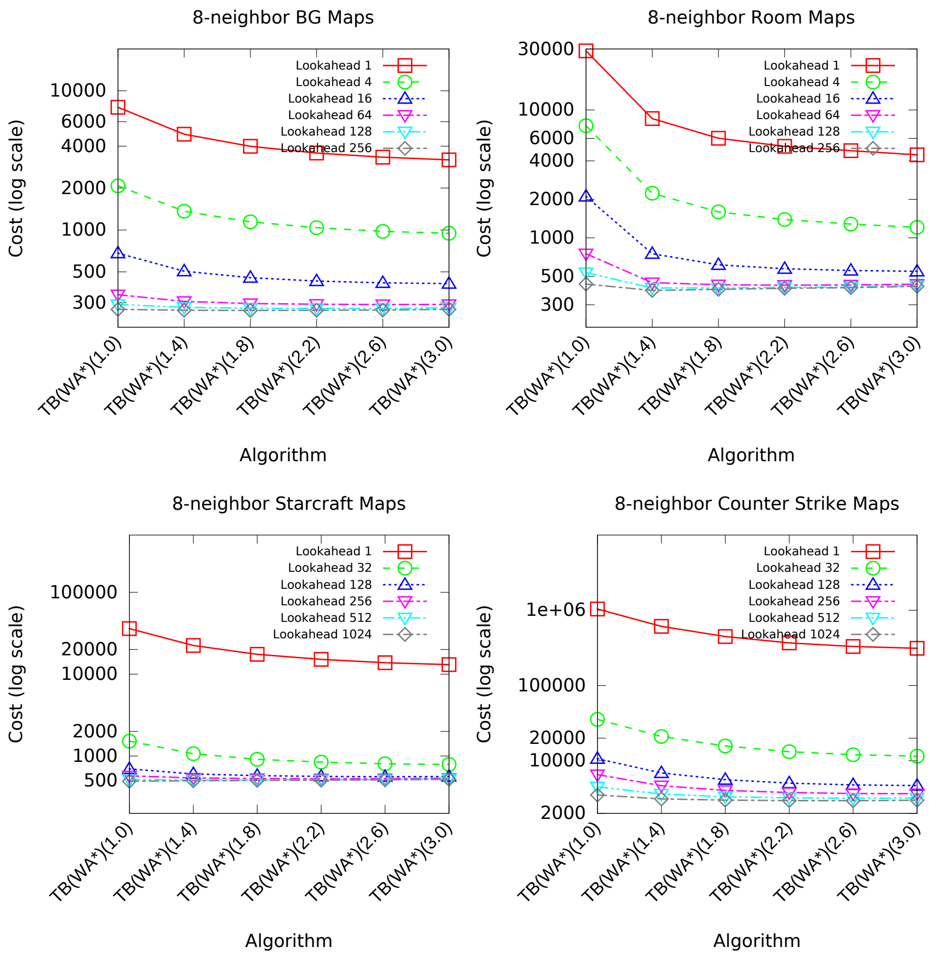

Figure 1: In the 8-neighbor results, solution cost tends to decrease as w or the lookahead parameter is increased.

path-finding repository (Sturtevant, 2012). In addition, we used 7 large maps from *Counter Strike* (CS), whose sizes range between between 4259×4097 and 4096×5462.

We evaluated six lookahead values $(1, 4, 16, 64, 128, 256)$ for the 512×512 maps and six lookahead values $(1, 32, 128, 256, 512, 1024)$ for SC and CS maps. We used six weight values $(1.0, 1.4, 1.8, 2.2, 2.6, 3.0)$. For each map we generated 50 random solvable search problems, resulting in 1800 problems for BG, 2000 problems for ROOMS, 3250 problems for SC, and 350 problems for CS.

Figures 1 and 2 show performance measures for the 8-neighbor grid maps. Note here that the average search time per episode is the same across all algorithms when using the same lookahead parameter. This is because search time per episode is proportional to the lookahead parameter and depends on no other variable (in particular, it does not depend on the weight). Thus fair conclusions can be drawn when comparing two configurations if their lookahead parameter is set to the same value.

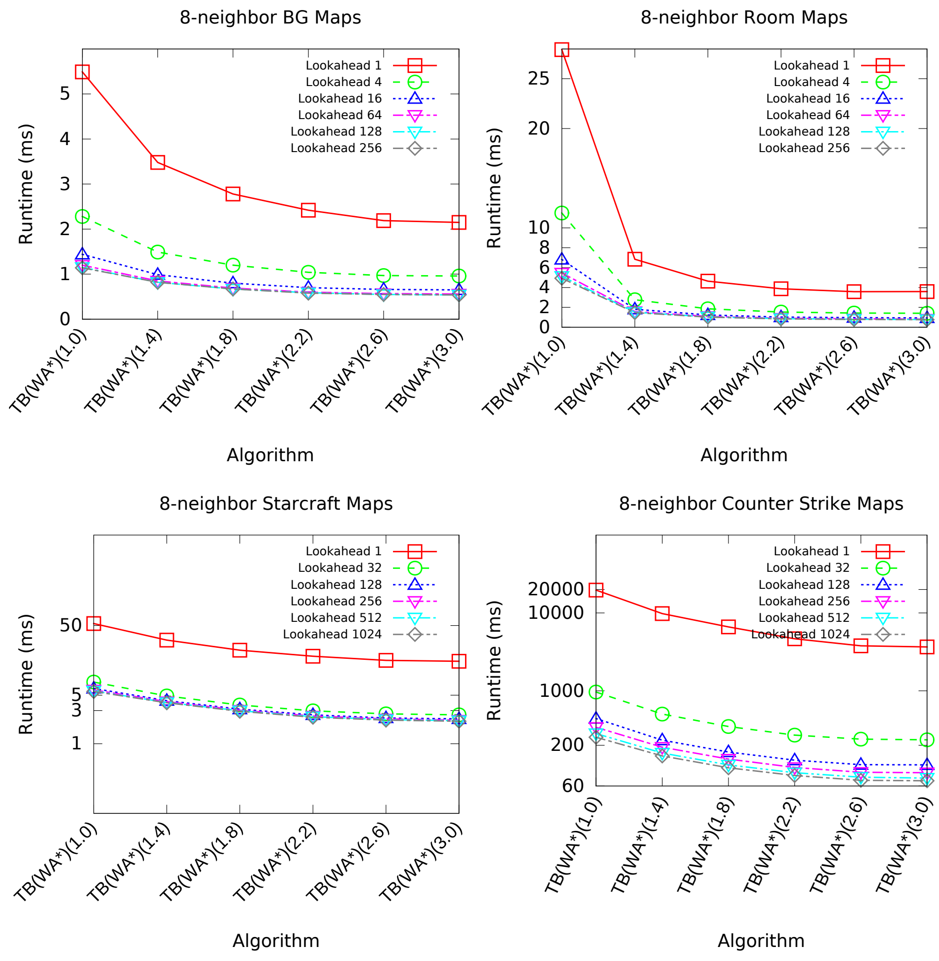

Figure 2: In the 8-neighbor results, search time typically decreases as w or the lookahead parameter is increased.

We observe the following relations hold for all maps regarding solution cost and search time.

Solution Cost For most lookahead values, solution cost *decreases* as w is *increased*. More significant improvements are observed for lower lookahead values. This is not surprising in the light of our cost bound (Theorem 2) . For large lookahead parameters (≥ 256), the value of w does not affect solution cost significantly. When the lookahead parameter increases, fewer search episodes are needed and less physical backtracks (back moves) are needed (Hernández et al., 2014). Back moves strongly influence the performance of the algorithms. In TB(WA*), when w is increased the number of back moves decreases, which explains the improvement in solution quality. For example, in the BG maps, when using lookahead 1, the average reduction of back moves is 1,960.5, when comparing $w = 1$ and $w = 3$, whereas when lookahead is 512 this reduction is only 2.4, when comparing $w = 1$ and $w = 3$.

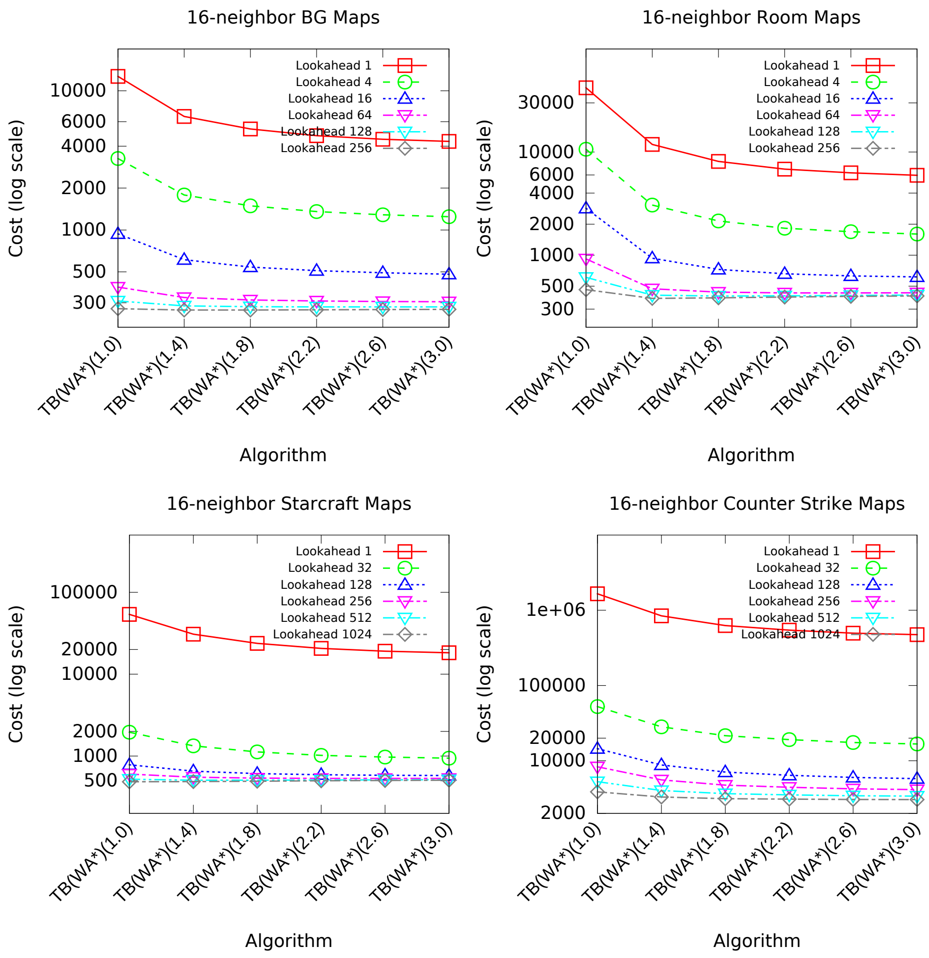

Figure 3: In the 16-neighbor results, solution cost tends to decrease as w or the lookahead parameter is increased.

Search Time As w is increased, search time decreases significantly for lower lookahead values and decreases moderately for higher lookahead values. In ROOMS we observe the largest improvements when w is increased. This behavior in ROOMS is explained because WA* performs very well in this type of map for $w > 1$.

Figures 3 and 4 show performance measures for the 16-neighbor grid maps. We observe the same relations observed in 8-neighbor grid maps regarding solution cost and search time.

4.1.1 8-NEIGHBOR VERSUS 16-NEIGHBOR GRID MAPS

Lower cost solutions are obtained with 8-neighbor grids than with 16-neighbor grids for the lookahead values 1, 4, and 16 in BG. Note that there exist some 16-neighbor movements which are more expensive than any 8-neighbor moves, so for small lookaheads, 16-neighbor solutions may have a similar number of moves, but a worse quality than 8-neighbor solutions. On the other hand, a

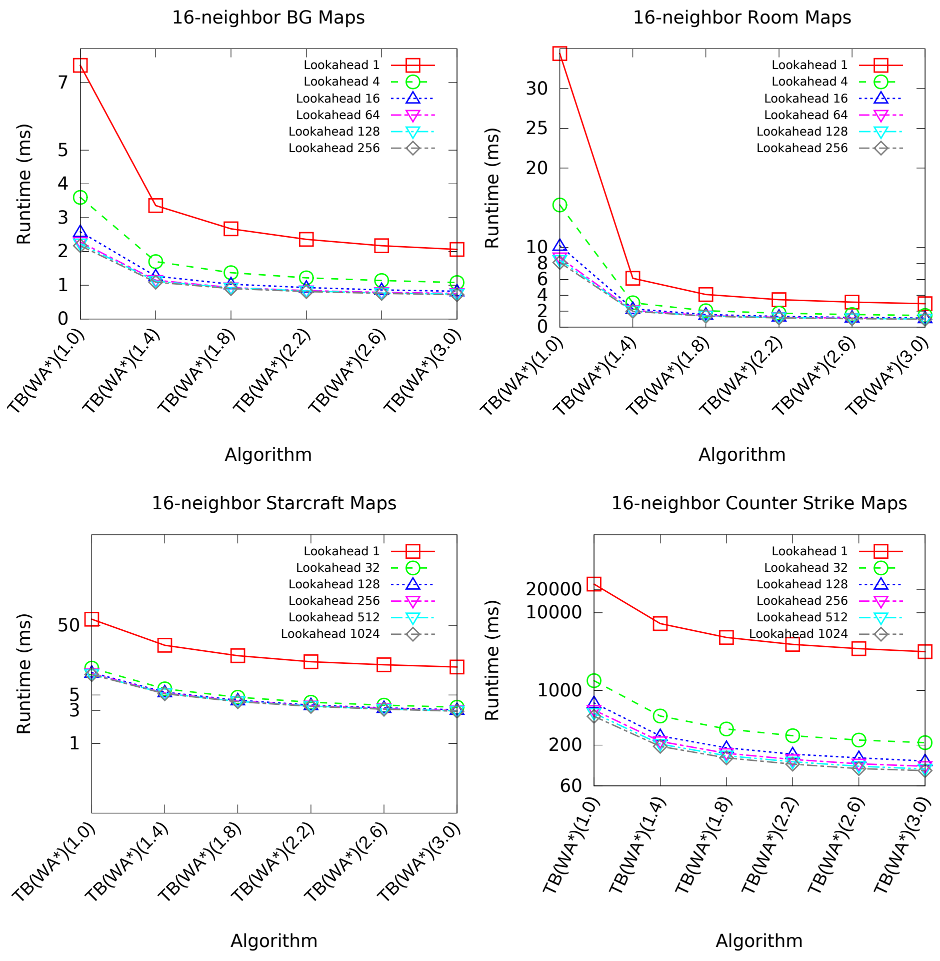

Figure 4: In the 16-neighbor results, search time typically decreases as w or the lookahead parameter is increased.

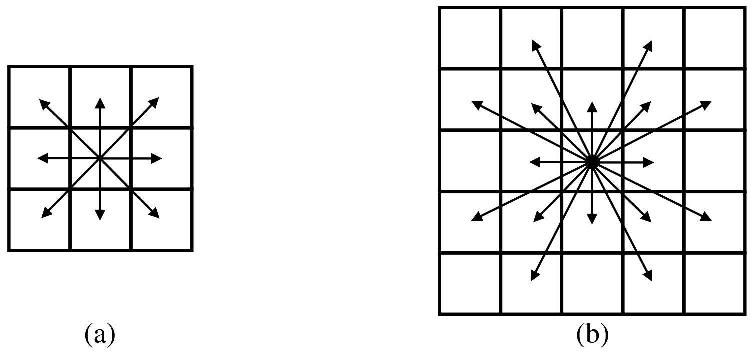

Figure 5: 8-neighborhoods (a) and 16-neighborhoods (b).

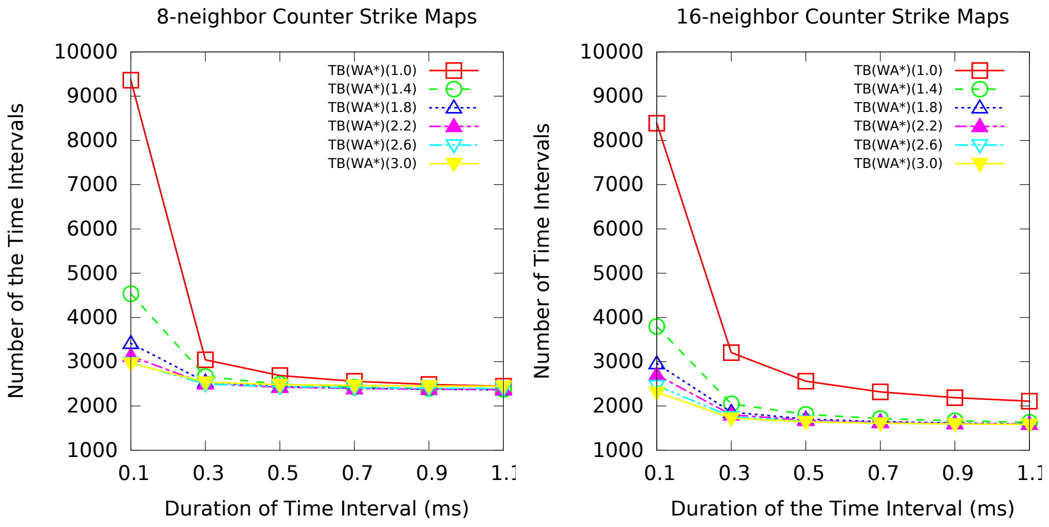

Figure 6: Results in the Game Time Model.

similar quality is observed for other lookahead values. TB(WA*), for almost all values of w and lookahead configurations, on 16-neighbor grids performs fewer moves than on 8-neighbor grids. For example, in SC when $w = 2.6$ and the lookahead parameter is 1024, 8-neighbor grids need a factor of 1.6 more moves than 16-neighbor grids. Note however that some 16-neighbor moves have a higher cost than 8-neighbor moves. Regarding runtime, TB(WA*) in 8-neighbor connectivity runs faster than TB(WA*) in 16-neighbor connectivity. This happens because the expansion of a state with 16-neighbor connectivity takes more time than expanding the same state with 8-neighbor connectivity.

4.2 Results on the Game Time Model

We report results for TB(WA*) using the Game Time Model in the Counter Strike maps for 8- and 16-neighbor grids. We use $0.1, 0.3, 0.5, 0.7, 0.9, 1.1$ milliseconds as the duration of the time intervals. In this setting, the quality of the solution is measured as the number of time intervals required to solve the problem, so the fewer the intervals that are used, the better the solution quality is.

Figure 6 shows average performance. We observe that when the length of the time interval increases, TB(WA*) yields solutions of better quality. On the other hand, as w is increased, TB(WA*) obtains better solutions. This can be observed more clearly when the duration of the intervals is small (e.g., 0.1ms). We also observe that better-quality solutions with 16- rather that with 8-neighbor connectivity. This is because with 16-neighbor connectivity the agent can perform a knight move in a single interval.

4.3 Results on Non-reversible Search Graphs: The Racetrack

In this section we compare TBR(WA*) and LSS-LRTWA* on a deterministic version of the *race-track problem* (Barto, Bradtke, & Singh, 1995; Bonet & Geffner, 2003). In this problem the race-

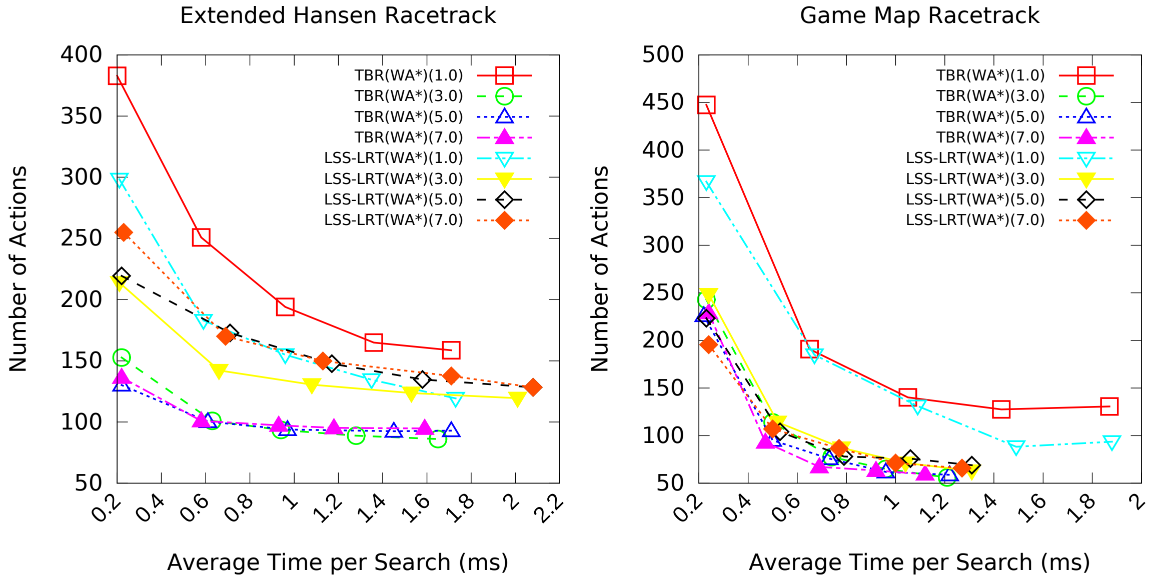

Figure 7: Results on the Racetrack Grids.

track is represented as a grid where some cells are marked as obstacles. Similar to grid pathfinding, the problem is to move an agent from a set of initial positions to any of the cells marked as a final position. Nevertheless, in this problem the agent has an associated velocity, and the set of actions involve accelerating (vertically or horizontally), or performing a no-op action which maintains the current velocity.

A state in the racetrack is a tuple (x, y, v_x, v_y), where (x, y) is the position of the vehicle, and (v_x, v_y) is the velocity vector. The actions are represented as tuples of the form (a_x, a_y), where $a_x, a_y \in \{-1, 0, 1\}$, which correspond to an acceleration vector. Unlike the original version (Barto et al., 1995), in ours actions are deterministic and we have only one initial and one destination cell. Because actions are deterministic, when (a_x, a_y) is performed in (x, y, v_x, v_y), the new state is given by (x', y', v'_x, v'_y), where $v'_x = v_x + a_x$ and $v'_y = v_y + a_y$, and where (x', y') is computed considering the vehicle changes its velocity to (v'_x, v'_y) before moving. When the movement towards (x', y') would lead to crashing into an obstacle, like Bonet and Geffner (2003) do, we leave the vehicle next to such an obstacle with velocity $(0, 0)$.

In our experiments, we used two racetracks. The first—which we refer to as HRT—is a 33×207 grid which corresponds to an extended version of the racetrack used by Hansen and Zilberstein (2001) (which is a 33×69 grid). We also use the game map AR0205SR from Baldur's Gate, whose size is 214x212. Below we refer to such a map by GRT.

We generated 50 random test cases for HRT and GRT that were such that the Manhattan distance between the initial state and goal state was greater than half of the width of the map. The absolute value of each of the components of the velocity vector is restricted to be at most 3. As a heuristic we use the Euclidean distance divided by the maximum speed.

We evaluated TB$_R$ (WA*) and LSS-LRTAWA* with four weight values $(1.0, 3.0, 5.0, 7.0)$. Figure 7 shows a plot of the number of actions versus average time per search episode. For TB$_R$ (WA*) the number of actions corresponds to the sum of the number of moves plus the number of times the

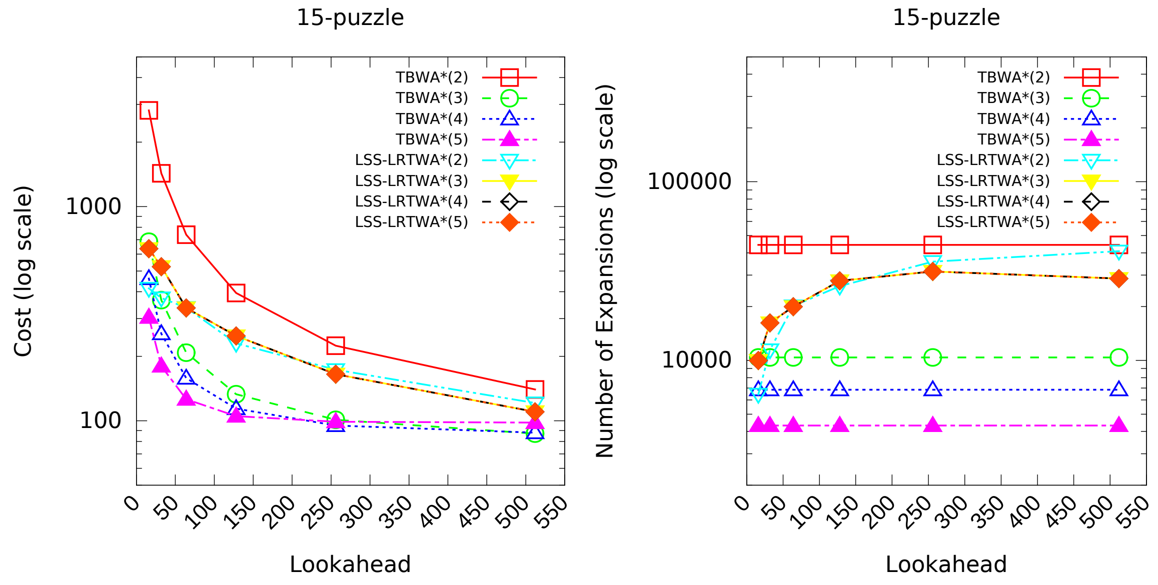

Figure 8: Cost and time comparison between TB-WA and LSS-LRTWA*

vehicle did not move. We do this because TB$_R$ (WA*) does not make any movements when search is restarted.

It is important to note that the time spent updating the heuristic is proportional to the number of states being updated. As such an update by TB$_R$ (WA*) may take more time than an update by LSS-LRTWA* because the *Closed* list may contain more states for the former algorithm. For this reason we use in our comparison the average time per search, which considers both search and update time.

In HRT (Figure 7) we observe that the worst behavior is the one obtained with TB$_R$ (WA*)(1.0). Both algorithms improve performance when increasing w, but TB$_R$ (WA*), used with a weight greater than 1.0, is the algorithm that clearly yields the best performance. In GRT, the worst algorithms are TB$_R$ (WA*)(1.0) and LSS-LRTA(1.0). Here, both algorithms improve when increasing the weight.

Because in this benchmark we used fewer problems than on the game maps, we carried out a 95% confidence analysis on for the cost of the solutions. In the HRT, this showed that costs for our best configuration TB$_R$ (WA*)(5.0) could be 10% away from the true mean, while for LSS-LRTA*(3.0) costs could be 11% away from the true mean. In the GRT, on the other hand, the difference in performance of the two best configurations TB$_R$ (7) and LSS-LRTWA*(7) is not statistically significant.

Finally, our experiments showed that the computational cost of learning phase of TB(WA*) is not higher than that of LSS-LRTA(WA*). Indeed, the number of updates carried out by TB(WA*) is 3.4 times less than the number of updates carried out by LSS-LRTA(WA*) in HRT and 1.6 time less in GRT. This explains the better performance in terms of runtime.

4.4 Results on the 15-Puzzle

We chose the 15-puzzle as a another domain for evaluating the time-bounded algorithms. We build our 15-puzzle implementation extending Richard Korf's implementation available from Carlos Linares's homepage.[2] We present results for TB(WA*), and LSS-LRT(WA*) algorithms. We use the 100 test cases presented by Korf (1993), which uses the Manhattan distance as a heuristic.

In this domain we report the results in a slightly different way. First, we omit results for TB(A*) (TB(WA*) with $w = 1$) because it does not terminate in a reasonable time. This is due to the fact that A* needs too many expansions for solving the hardest test cases. Second, we use the number of expansions instead of runtime as an efficiency measure. In this domain, we found this measure to be more stable since, in general, solving all 100 problems does not take too much time when $w > 1$ (0.3s for $w = 2$; 0.08s for $w = 3$), and thus time is prone to be affected by external factors controlled by the operating system.

Figure 8 shows the performance of TB(WA*) and LSS-LRT(WA*). We use lookahead values in $\{16, 32, 64, 128, 256, 512\}$ and weights in $\{2, 3, 4, 5\}$. We observe the following relations.

Solution Cost The solution cost of TB(WA*) *decreases* as w is *increased* for almost all lookahead values. TB(WA*) obtains better results than LSS-LRTWA* for all lookahead values when $w > 2$. With $w < 2$ the performance of TB(WA*) is worse than the performence of LSS-LRTA*. On the other hand, TB(WA*) with $w = 5$ obtains a solution 2.0 times better on average than the solution obtained by LSS-LRTA* (LSS-LRTWA* with $w = 1$).

Number of Expansions The number of expansions of TB(WA*) *decreases* as w is *increased*. TB(WA*) is more efficient than LSS-LRTWA* for all lookahead values and $w > 2$. The worst performing configuration for TB(WA*) is $w = 1$.

Note that the curve remains flat for several of the configurations. This is because a small number of expansions are needed to solve the problem.

In conclusion, considering solution cost and number of expansions, in 15-puzzle TB(WA*) is the better algorithm. For instance, the average solution cost of TB(WA*) is 1.6 times better on average than the average solution cost of LSS-LRTA*.

We did not compare to the greedy algorithm of (Parberry, 2015), which is real-time but domain-specific, unlike our.

5. Summary and Conclusions

This paper introduced Time-Bounded Best-First Search, a generalization of the real-time search algorithm Time-Bounded A*. In addition, it introduced a restarting version of the time-bounded approach, TB_R (WA*), which unlike TB(BFS), has a better coverage of non-reversible domains.

We carried out a theoretical analysis of both TB(WA*) and TB_R (WA*), including termination results and a cost bound for TB(WA*). Given a weight w, our bound suggests that TB(WA*) will be significantly superior to TB(A*) precisely on search problems in which WA* expands significantly fewer states than A*. In addition, our bound suggests that TB(WA*) may not yield benefits in domains in which WA*, run offline, will not yield any improvements over A*. Our theoretical bounds can be easily adapted to other instances of Best-First Search that offer guarantees on solution

2. `http://scalab.uc3m.es/~clinares/download/source/ida/ida.html`

quality. For TB_R (WA*), we proved termination in strongly connected graphs, even if they contain non-reversible actions. This property is also enjoyed by real-time search algorithms of the LRTA* family but is not enjoyed by TB(BFS).

In our experimental evaluation, that focused on pathfinding, the 15-puzzle, and the racetrack problem, we found both TB(WA*) and TB_R (WA*) to be significantly superior to some real-time search algorithms of the LRTA* family. In addition, we found that performance tends to improve as the weight parameter is increased, without increasing the time per search episode. This finding is interesting because although quality can also be improved by increasing the lookahead parameter, this increases the time spent on each search episode.

It is well known that in many search benchmarks, WA* may expand significantly fewer nodes than A*. Consistent with this, in our experiments, time-bounded versions of suboptimal algorithms like Weighted A* produce significantly better solutions than those obtained by TB(A*). Improvements are less noticeable when the lookahead parameter is large, as is also predicted by theory.

We are not the first to observe performance gains when using weights in a real-time setting. Indeed, our findings are consistent with those of Rivera et al. (2015), who also obtain better solutions by using weighted heuristics. Our work adds another piece of evidence that justifies studying the incorporation of weights into other real-time algorithms (e.g., RIBS and EDA;* Sturtevant, Bulitko, & Björnsson, 2010; Sharon, Felner, & Sturtevant, 2014). Finally, SLA* (Shue & Zamani, 1993) and LRTS (Bulitko & Lee, 2006) are two algorithms that also perform backtracking moves. An investigation of whether or not restarts could provide benefits for those algorithms is left for future work.

Acknowledgements

We thank Vadim Bulitko for providing the Counter Strike maps. This research was partly funded by Fondecyt grant number 1150328.

References

Aine, S., & Likhachev, M. (2013). Truncated incremental search: Faster replanning by exploiting suboptimality. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*, Bellvue, Washington, USA.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*(1-2), 81–138.

Björnsson, Y., Bulitko, V., & Sturtevant, N. R. (2009). TBA*: Time-bounded A*. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 431–436.

Bonet, B., & Geffner, H. (2003). Labeled rtdp: Improving the convergence of real-time dynamic programming.. In *ICAPS*, Vol. 3, pp. 12–21.

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

Bulitko, V., Björnsson, Y., Sturtevant, N., & Lawrence, R. (2011). *Real-time Heuristic Search for Game Pathfinding*. Applied Research in Artificial Intelligence for Computer Games. Springer.

Burns, E., Ruml, W., & Do, M. B. (2013). Heuristic search when time matters. *Journal of Artificial Intelligence Research*, *47*, 697–740.

Ebendt, R., & Drechsler, R. (2009). Weighted A* search - unifying view and application. *Artificial Intelligence*, *173*(14), 1310–1342.

Gaschnig, J. (1977). Exactly how good are heuristics?: Toward a realistic predictive theory of best-first search. In Reddy, R. (Ed.), *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 434–441. William Kaufmann.

Hansen, E. A., & Zilberstein, S. (2001). Lao: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, *129*(1), 35–62.

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., Asín, R., & Baier, J. A. (2014). Time-bounded best-first search. In *Proceedings of the 7th Symposium on Combinatorial Search (SoCS)*.

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., Baier, J. A., Uras, T., & Koenig, S. (2012). TBAA*: Time-Bounded Adaptive A*. In *Proceedings of the 10th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, pp. 997–1006, Valencia, Spain.

Koenig, S. (2001). Agent-centered search. *Artificial Intelligence Magazine*, *22*(4), 109–131.

Koenig, S., & Likhachev, M. (2005). Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, *21*(3), 354–363.

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.

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

Korf, R. E. (1993). Linear-space best-first search. *Artificial Intelligence*, *62*(1), 41–78.

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.

Parberry, I. (2015). Memory-efficient method for fast computation of short 15-puzzle solutions. *IEEE Trans. Comput. Intellig. and AI in Games*, *7*(2), 200–203.

Pearl, J. (1984). *Heuristics: Preintelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

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

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

Schmid, K., Tomic, T., Ruess, F., Hirschmüller, H., & Suppa, M. (2013). Stereo vision based indoor/outdoor navigation for flying robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3955–3962.

Sharon, G., Felner, A., & Sturtevant, N. R. (2014). Exponential deepening a* for real-time agent-centered search. In *Proceedings of the 7th Symposium on Combinatorial Search (SoCS)*, pp. 871–877.

Shue, L., & Zamani, R. (1993). An admissible heuristic search algorithm. In Komorowski, H. J., & Ras, Z. W. (Eds.), *Proceedings of the 7th International Symposium Methodologies for Intelligent Systems (ISMIS)*, Vol. 689 of *LNCS*, pp. 69–75. Springer.

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

Sturtevant, N. R., Bulitko, V., & Björnsson, Y. (2010). On learning in agent-centered search. In *Proceedings of the 9th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, pp. 333–340, Toronto, Ontario.

Wilt, C. M., & Ruml, W. (2012). When does weighted A* fail?. In *Proceedings of the 5th Symposium on Combinatorial Search (SoCS)*, Niagara Falls, Ontario, Canada.