Time-Bounded Best-First Search

Carlos Hernández **Roberto Asín**
Depto. de Ingeniería Informática
Universidad Católica de la Santísima Concepción
Concepción, Chile

Jorge A. Baier
Depto. de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Santiago, Chile

Abstract

Time-Bounded A* (TBA*) is a 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. Although the idea underlying TBA* can be generalized to other single-agent deterministic search algorithms, little is known about the impact on performance that would result from using algorithms other than A*. In this paper we propose Time-Bounded Best-First Search (TB-BFS) a generalization of the time-bounded approach to any best-first search algorithm. Furthermore, we propose restarting strategies that allow TB-BFS to solve search problems in dynamic environments. In static environments, we prove that the resulting framework allows agents to always find a solution if such a solution exists, and prove cost bounds for the solutions returned by Time-Bounded Weighted A* (TB-WA*). We evaluate the performance of TB-WA* and Time-Bounded Greedy Best-First Search (TB-GBFS). We show that in pathfinding applications in static domains, TB-WA* and TB-GBFS are not only faster than TBA* but also find significantly better solutions in terms of cost. In the context of videogame pathfinding, TB-WA* and TB-GBFS perform fewer undesired movements than TBA*. Restarting TB-WA* was also evaluated in dynamic pathfinding random maps, where we also observed improved performance compared to restarting TBA*. Our experimental results seem consistent with theoretical bounds.

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 et al. 2011), to highly dynamic robotics (Schmid et al. 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.

Time-Bounded A* (TBA*) (Björnsson, Bulitko, and 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. It terminates when the agent has reached the goal.

TBA* is among the fastest algorithms for real-time-constrained applications. In fact, Hernández et al. (2012) showed it significantly outperforms state-of-the-art real-time heuristic search algorithms such as RTAA* (Koenig and Likhachev 2006) and daRTAA* (Hernández and Baier 2012).

In this paper we extend the time-bounded search approach to a broader class of algorithms. Specifically, we propose Time-Bounded Best-First Search (TB-BFS), which is a family of time-bounded algorithms suitable for static environments. In addition, we propose two restart strategies—eager and lazy restart—that allow TB-BFS to solve search problems in dynamic domains, in which action costs may change during execution. Furthermore, we propose measures of quality inspired by videogame applications that go beyond solution cost and aim at capturing when solutions "look good" to an observer.

We focus on two instances of TB-BFS: Time-Bounded Weighted A* (TB-WA*) and Time-Bounded Greedy Best-First Search (TB-GBFS). Theoretically, we show that TB-BFS is complete, and establish an upper bound on the cost of solutions returned by TB-WA* in static domains. Our cost bound establishes that in some domains solution cost may be reduced *significantly* by increasing w; hence, in contrast to Weighted A*, 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 TBA* in domains in which WA* runs faster than A*. While WA* does not always run faster than A* (see e.g., Wilt and Ruml, 2012), it is known that it does in many domains.

Experimentally, we evaluate the two algorithms on pathfinding benchmarks in static terrain and show that increasing w allows finding significantly better solutions in less time. In terms of our videogame-inspired quality measures TB-GBFS seems to be superior in static domains. In dynamic domains we evaluate TB-WA* on random maps and, yet again, we show that the use of weights greater than one allows improving performance rather significantly. We also show that lazy restart is superior to eager restart.

The rest of the paper is organized as follows. We start off describing the background needed for the rest of the paper. Then we describe TB-BFS for static and dynamic domains

and carry out a theoretical analysis. This is followed by a description of our videogame-inspired measures of quality. Then we describe the experimental results, and finish with a summary and perspectives for future research.

Background

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

Search in Static and Dynamic Environments

A search graph is a tuple $G = (S, A)$, where S is a set of states (vertices), $A \subseteq S \times S$ is a set of edges which represent the actions available to the agent in each state. A path over graph (S, A) is a sequence of states $\pi = s_0 s_1 \cdots s_n$, where $(s_i, s_{i+1}) \in A$, for all $i \in \{0, \ldots, n-1\}$.

A *cost function* c for a search graph (S, A) is such that $c : A \rightarrow \mathbb{R}^+ \cup \{\infty\}$; i.e., it associates an action with a positive cost. The *cost of a path* $\pi = s_0 s_1 \cdots s_n$ is $\sum_{i=0}^{n-1} c(s_i, s_{i+1})$, i.e. the sum of the costs of each edge considered in the path. Given c, we say that t is a *successor* of s if (s, t) is an edge in A with finite cost; moreover, for every $s \in S$ we define $Succ_c(s) = \{t \mid (s, t) \in A, c(s, t) \neq \infty\}$. Thus, our framework allows two alternatives for representing that state t is not a successor of state s: either by saying $(s, t) \notin A$ or by defining $c(s, t) = \infty$. As we see later, this will help us with the definition of dynamic environments.

In this paper we assume the search graph is *undirected*, which informally means that every action is reversible. This restriction is true in many interesting search problems, but we need it here because TBA* and hence TB-BFS may have the agent undo previously performed actions—a process known as *physical backtracking*.

We focus on search problems in two general settings: *static* and *dynamic* environments. A search problem in a static environment is a tuple $P = (S, A, c, s_{start}, s_{goal})$, where $G = (S, A)$ is a search graph, c a cost function and $s_{start}, s_{goal} \in S$ are, respectively, the initial and the goal state. The problem is to compute a path $\pi = s_0 s_1 \cdots s_n$ such that $s_0 = s_{start}$, $s_n = s_{goal}$, and such that $s_{i+1} \in Succ_c(s_i)$, for all $i \in \{0, \ldots, n-1\}$.

For dynamic environments we assume that after the agent moves, the costs of the arcs may change. Given a tuple, $P = (S, A, \gamma, s_{start}, s_{goal})$ where S, A, s_{start}, and s_{goal} are as above and $\gamma = c_0 c_1 \cdots$ is an infinite sequence of cost functions over (S, A), the problem is to compute a path $\pi = s_0 s_1 \cdots s_n$ such that $s_0 = s_{start}$, $s_n = s_{goal}$, and such that $s_{i+1} \in Succ_{c_i}(s_i)$, for all $i \in \{0, \ldots, n-1\}$. Observe that given the way we define $Succ_{c_i}$, states can be disconnected or reconnected to the search space as the agent executes actions.

Best-First Search

Best-First Search (Pearl 1984) captures a family of search algorithms for static environments which associate a priority $p(s)$ with every state s. The priority is computed using an evaluation function, f, that is such that a *lower* value to states that are viewed as closer to the goal.

The algorithm starts off by initializing the priority of all nodes in 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 $f(t)$. If $f(t)$ is lower than $p(t)$, then t is added to $Open$ and $p(t)$ is set to $f(t)$. The algorithm repeats this process until s_{goal} is in $Open$ with the lowest priority.

An instance of Best-First Search is *Weighted A** (WA*) (Pohl 1970). Its evaluation function is defined as $f(s) = g(s) + wh(s)$, where $g(s)$ is the cost of a path from s_{start} to s, h is a user-given *heuristic function* such that $h(s)$ estimates the cost of a path from s to s_{goal}, and w is a real number greater than or equal to 1. Note that the way $g(s)$ is computed here depends on the particular path discovered to s. Throughout execution state s may be re-discovered multiple times and hence receive different g-values.

Function h is *admissible* when $h(s)$ does not overestimate the cost of an optimal path from s to s_{goal}, for all $s \in S$. If h is admissible WA* is known to find a solution whose cost cannot exceed wc^*, where c^* is the cost of a shortest path from s_{start} to s_{goal}. As such, WA* may return increasingly worse solutions as w is increased. The advantage of increasing w is that execution is also *faster*. When $w = 1$, WA* is equivalent to A* (Hart, Nilsson, and Raphael 1968).

Another instance of Best-First Search is *Greedy Best-First Search* (GBFS). Here f is equal to the user-given heuristic function h. When w is very large GBFS is similar to WA* but not equivalent; indeed, in both algorithms search is mainly driven by h but in WA* $g(n)$ winds up acting as a tiebreaker.

Real-Time Search

In Real-Time search the objective is to solve a search problem with an additional *real-time constraint*: constant—and usually little—computational time is given to the agent before each action is performed. As such, to solve a search problem search must be interleaved with execution. The objective of the search phase is to provide the agent with a move that hopefully leads it to the goal.

An example of a Real-Time Search algorithm is Learning Real-Time A* (LRTA*; Korf, 1990). In static environments the algorithm is as follows.

1. set s to s_{start}.
2. set y to $\arg\min_{t \in Succ_c(s)} c(s, t) + h(t)$; i.e., the most promising neighbor.
3. If $h(s) < c(s, y) + h(y)$, set $h(s)$ to $c(s, y) + h(y)$.
4. execute the action given by arc (s, y) and set s to y.
5. terminate if at the goal state; otherwise go back to step 2.

Step 3 is called the learning step; it makes h more informed and is essential to avoid infinite loops.

Under reasonable conditions, LRTA* and many other variants (e.g., LSS-LRTA*; Koenig and Sun, 2009) are guaranteed to lead the agent to s_{goal} when there is a path from s_{start} to s_{goal}. However when there is no solution to the problem, many of these algorithms iterate forever. As we

see later Time-Bounded algorithms have the additional advantage that they will prove a solution does not exist, and always terminate.

Time-Bounded Best-First Search

Time-Bounded A* (Björnsson, Bulitko, and Sturtevant 2009) is a real-time search algorithm based on A*. Intuitively, TBA* can be understood as an algorithm that runs an A* search rooted at s_{start} whose objective is to reach s_{goal}. Unlike A*, TBA* expands a constant number of states between the execution of each action. More specifically, TBA* alternates a search phase with a movement phase until the goal is reached. If the goal state has not been reached, in each search phase it expands k states, and then builds 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 movement phase, 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, it performs a *physical backtrack*, moving the agent to its previous position. Physical backtracking is a mechanism that guarantees that the agent will eventually reach a state in variable $path$. As soon as such a state is reached the agent will possibly start moving towards the state believed to be closest to the goal.

A pseudo-code for Time-Bounded Best-First Search (TB-BFS) is presented in Algorithm 1. 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 is like TBA* but a generic Best-First Search which expands at most k states per call is run instead of A*. The current position of the agent is kept in $s_{current}$. Routine *Initialize-Search*()—which is only called once— initializes the priority of each state. Variable $goalFound$ is set initially to false and made true as soon as a solution is found by *Best-First-Search*().

Algorithm 1 is equivalent to TBA* when the evaluation function is defined as $f(s) = g(s) + h(s)$, where h is a user-given heuristic and $g(s_{start}) = 0$, and $g(s) = g(s') + c(s', s)$ when s was expanded from s'. We call the algorithm resulting from using $f(s) = g(s) + wh(s)$ *Time-Bounded WA** (TB-WA*). Finally, we call *Time-Bounded GBFS* (TB-GBFS) the algorithm that results when $f(s) = h(s)$ is used.

Algorithm 1: Time-Bounded Best-First Search

```
1  procedure InitializeSearch()
2      s_root ← s_current
3      Open ← ∅
4      foreach s ∈ S do
5          p(s) = ∞
6      p(s_root) ← f(s_root)
7      Insert s_root in Open
8      goalFound ← false
9  function Best-First-Search()
10     expansions ← 0
11     while Open ≠ ∅ and expansions < k and
       p(s_goal) > min_{t∈Open} p(t) do
12         Let s be the state with minimum priority in Open
13         Remove s from Open
14         foreach t ∈ Succ_c(s) do
15             Compute f(t) given this newly found path.
16             if f(t) < p(t) then
17                 p(t) ← f(t)
18                 parent(t) ← s
19                 Insert t in Open
20         expansions ← expansions + 1
21     if Open = ∅ then return false
22     Let s_best be the state with minimum priority in Open.
23     if s_best = s_goal then goalFound ← true
24     path ← path from s_root to s_best
25     return true
26 function MoveToGoal()
27     s_current ← s_start
28     InitializeSearch()
29     while s_current ≠ s_goal do
30         if goalFound = false and Best-First-Search() = false then
31             return false;
32         if s_current is on path then
33             s_current ← state after s_current on path
34         else
35             s_current ← parent(s_current);
36         Execute movement to s_current
37     return true
38 procedure Main
39     if MoveToGoal() = true then
40         print("the agent is now at the goal state")
41     else
42         print("no solution")
```

Properties

Now we analyze a few interesting properties of the algorithms we have just proposed. First, just like TBA*, TB-BFS always terminates and finds a solution if one exists. This is an important property since many real-time algorithms (e.g., LRTA*) enter an infinite loop on unsolvable problems. Second, we prove an upper 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 problem P if a solution to P exists. Otherwise, it will eventually print "no solution".*

Proof: Straightforward from the fact that Best-First Search returns a solution iff one exists. ■

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 TBA* but to our knowledge Lemma 1 and Theorem 2 had not been proven before for TBA*.

In the results below, we assume that $P = (S, A, c, s_{start}, s_{goal})$ is a static search problem, that TBA-WA* is run with a parameter $w \geq 1$ and an admissible heuristic. Furthermore, we assume c^* is the cost of a minimum-cost path from s_{start} to s_{goal}, that $c^+ = \max_{(u,v)\in A} c(u, v)$, and that $N(w)$ is the number of expansions needed by WA* to solve P.

Lemma 1 *The cost incurred by an agent controlled by TB-WA* before goalFound becomes true does not exceed $\lfloor \frac{N(w)-1}{k} \rfloor c^+$.*

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 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 TBA* and stated in Theorem 2 by Hernández et al. (2012).

Lemma 2 *The cost incurred by an agent controlled by TB-WA* after $goalFound$ has become true cannot exceed $2wc^*$.*

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 run of WA* and therefore is such that $c(\pi) < wc^*$. 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^*. Thus the agent may not incur in a cost higher than $2wc^*$ to reach the goal. ■

Now we obtain an upper bound on solution cost for TB-WA* which follows straightforwardly from the two previous lemmas.

Theorem 2 *The solution cost obtained by TB-WA* is at most $\lfloor \frac{N(w)-1}{k} \rfloor c^+ + 2wc^*$.*

An interesting observation is that empirical observations show that in many domains, when w is increased, $N(w)$ decreases exponentially. Thus, when increasing w the first term of the sum may decrease exponentially while the second may increase only linearly. 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. On the other hand, note that the first term of the bound is less important if k is large or if $N(w)$ does not decrease significantly by increasing w. Thus, we expect less benefits from running TB-WA* over TBA* if k is large or when search is carried out in problems in which WA* does not expand significantly fewer nodes than A* in offline mode, which is the case in some domains (Wilt and Ruml 2012).

Dynamic Environments via Restarting

TB-BFS is not amenable for dynamic environments. Indeed since the cost function changes, the path being followed by the agent may contain two successive states that are not successors anymore. Furthermore, when an arc cost decreases, there may be a new lowest-cost path, cheaper than the one being followed so far. When the quality of the solution matters, following the currently available path may not be a good decision.

A straightforward approach to dynamic environments is what we call here *eager restarting*, which is to restart search each time the cost function has changed. This way we make sure the path followed by the agent is always obstacle-free. In addition this allows us to establish bounds on the cost incurred by the agent when it is moving forward on a path that contains the goal (in analogy to Lemma 2).

Algorithm 2: Repeated TB-BFS

```
1  function MoveToGoal()
2      s_current ← s_start
3      c ← get initial cost function
4      c_prev ← c
5      InitializeSearch()
6      while s_current ≠ s_goal do
7          if goalFound = false and Best-First-Search() = false then
8              return false;
9          if s_current is on path then
10             s_current ← state after s_current on path
11         else
12             s_current ← parent(s_current);
13         Execute movement to s_current
14         c_prev ← c
15         c ← get new cost function
16         if RestartSearch?() then
17             InitializeSearch()
18     return true
```

A pseudo-code for Restarting TB-BFS is shown in Algorithm 2. Functions not mentioned are inherited from TB-BFS. There are two main differences with TB-BFS: first, after each movement a new cost function is retrieved, and, second, search may be restarted after performing a movement. Eager restarting results from implementing *RestartSearch?*() as returning true if and only if $c_{prev} \neq c$.

Below we propose, however, *lazy restarting* strategies, whereby search is restarted only when needed. Algorithm 3 proposes an implementation for *RestartSearch?*(). If the cost of an arc (u, v) increases it checks whether or not (u, v) is an arc to be followed in the future and if that is the case, it restarts search. If otherwise, its cost decreases then it estimates the shortest path through arc (u, v) as $c' = h(s_{current}, u) + c(u, v) + h(v, s_{goal})$, where $h(s, t)$ is an estimate of the cost of a shortest path between s and t. Search is restarted if wc' is lower than the cost of the remaining path.

Algorithm 3: Lazy Restart Search for WA*

```
1  function RestartSearch?()
2      for each (u, v) such that c_prev(u, v) ≠ c(u, v) do
3          if c(u, v) > c_prev(u, v) then
4              if v is on path then return true
5          else
6              c' ← h(s_current, u) + c(u, v) + h(v, s_goal)
7              c ← cost of the suffix of path after s_current
8              if wc' < c then return true
9      return false
```

The particular way search is restarted allows us to prove the following result, which is analogous to Lemma 2 for static domains.

Theorem 3 *Let $P = (S, A, c_0 c_1 \cdots, s_{start}, s_{goal})$ be a search problem over a dynamic environment, and consider a run of Repeated TB-WA* with parameter $w \geq 1$ and an admissible heuristic. Let c_{path} be the cost of path with respect to cost function c_n, and c_n^* be the cost of a cost-minimal path from $s_{current}$ to s_{goal}. Then $c_{path} \leq wc_n^*$.*

In dynamic domains, it is unfortunately not possible to obtain completeness results like the one in Theorem 1 since an

ever changing cost function may eventually block all paths to the goal.

Quality Measures Beyond Solution Cost

Even though real-time heuristic search algorithms are applicable to videogame pathfinding, they are not used in deployed video games. Bulitko et al. (2011) argue that the main reason for this is due to the fact that these algorithms tend to repeatedly visit certain states of the search space—a behavior they describe as *scrubbing*. Indeed algorithms like LRTA* spend many iterations increasing the h-values of states in heuristic depressions, and certainly exhibit significant scrubbing.

Even though TBA* significantly outperforms LRTA* (Björnsson, Bulitko, and Sturtevant 2009), it may also exhibit significant scrubbing. To see this, imagine a TBA* run with lookahead 1. Each time the path to the best state in $Open$ changes, the agent performs physical backtracking (i.e., scrubbing).

We say that a move is a *back-move* whenever the movement is towards the parent of the current state (i.e., when $s_{current}$ is set to $parent(s_{current})$ prior to executing movement. Later we use back-moves to assess the quality of a solution in addition to cost.

In our experiments below we also measure the number of *non-optimal* moves, which we define as the moves that are not in an optimal path to s_{goal}. It is well-known that solutions returned by A* never contain a non-optimal move whereas those returned by WA* may contain a bounded number of them. This fact does not hold for Time-Bounded versions of these algorithms and therefore we considered valuable to include this metric in our evaluation.

Experimental Results

We evaluated variants of TB-BFS on pathfinding tasks in grid-like static and dynamic terrain. Specifically, we use eight-neighbor grids since they are often preferred in practice (Bulitko et al. 2011; Koenig and Likhachev 2005). Following existing experimental evaluations (Sun, Koenig, and Yeoh 2008; Aine and Likhachev 2013), the agent is aware of cost changes anywhere in the grid. The cost of a horizontal or vertical movement is 1 and cost of a diagonal movement is $\sqrt{2}$. The user-given heuristic are the octile distances. All the experiments were run in a 2.00GHz QuadCore Intel Xeon machine running Linux.

Static Environments

We used all 512×512-cell maps from the video games *World of Warcraft III* (WC3) and *Baldurs Gate* (BG), and all the Room maps (ROOMS) available from N. Sturtevant's pathfinding repository (Sturtevant 2012). In addition, we used larger 1024×1024-cell maps from the *StarCraft* game. Specifically, we used the Cauldron, Expedition, Octopus, PrimevalIsles, and TheFrozenSea map of size 1014×1024 from the video game StarCraft.

For the smaller maps we evaluated five lookahead values: $1, 4, 16, 64, 256$, whereas for larger maps we used six values: $1, 32, 128, 256, 512, 1024$. For each map we generated 100 random solvable test cases. For TB-WA* we used six weight values: $1.0, 1.4, 1.8, 2.2, 2.6, 3.0$. All algorithms have a common code base and use a standard binary heap for $Open$. In TB-WA*, ties in $Open$ are broken in favor of larger g-values. We evaluated the algorithms considering solution cost and the runtime, as measures of solution quality and efficiency, respectively.

Figure 1 and Figure 2 show performance measures for BG and ROOM. Since the WC3 plots look extremely similar when compared to BG, we omit them due to space restrictions. We observe the following relations hold for all maps regarding solution cost and search time.

Solution Cost For lookahead values up to 64, 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 (≥ 128), the value of w does not affect solution cost significantly. TB-GBFS, on the other hand, obtains the best-quality solutions among all algorithms when the lookahead parameter is small. In fact, in ROOMS TB-GBFS outperforms TBA* by almost an order of magnitude. With greater lookahead values, TB-WA*($w = 3$) is the algorithm obtaining best-quality solutions overall.

Search Time As w is increased, search time decreases significantly for lower lookahead values and decreases moderately for higher lookahead values. TB-GBFS is faster than TBA*, but in most cases TB-WA* is faster than TB-GBFS. In the large maps, even for high lookahead values (> 128) TB-WA*($w = 3$) and TB-GBFS are at least six times faster than TBA*.

Overall, results show that TB-WA* (with $w > 1$) and TB-BFS are superior to TBA* in both cost and search time in the majority of cases, especially when the lookeahead is small (i.e., when time resources are very limited). This is because these algorithms require fewer state expansions to reach the goal compared to TBA*.

Now we focus on the additional quality measures described above. Figure 3 and Figure 4 show the total number of non-optimal moves and the percentage of back-moves in BG and ROOMS relative to non-optimal moves. Again we omit WC from the plots for the same reasons given above. The following relations hold for all three benchmarks.

Non-Optimal Moves We observe the number of non-optimal moves performed by TB-WA* *decreases* as w is increased, except for lookahead 256 in smaller maps and lookahead 512 or higher for larger maps. For high lookahead values, TBA* performs fewer non-optimal moves than TB-WA* ($w > 1$), which is explained by the fact that TBA* finds an optimal solution requiring fewer agent moves when the lookahead parameter is high. TB-GBFS performs fewer non-optimal moves for small lookaheads. With higher lookahead values TB-GBFS yields a higher number of non-optimal moves than other algorithms.

Scrubbing TBA* is the algorithm that on average performs the highest proportion of back-moves compared to TB-WA* ($w > 1$) and TB-GBFS. The tendency is that the

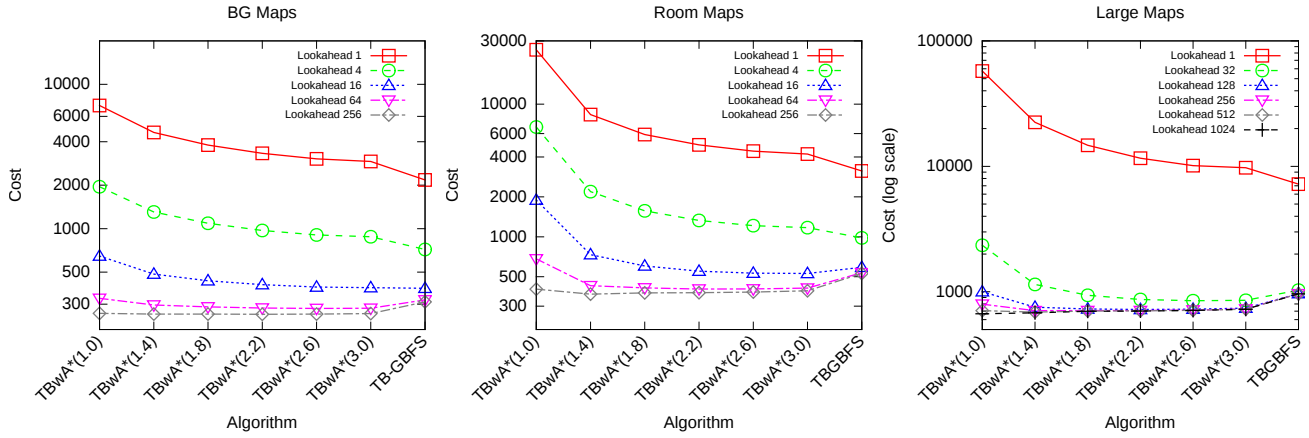

Figure 1: In static domains, solution cost tends to decrease as w or the lookahead parameter is increased.

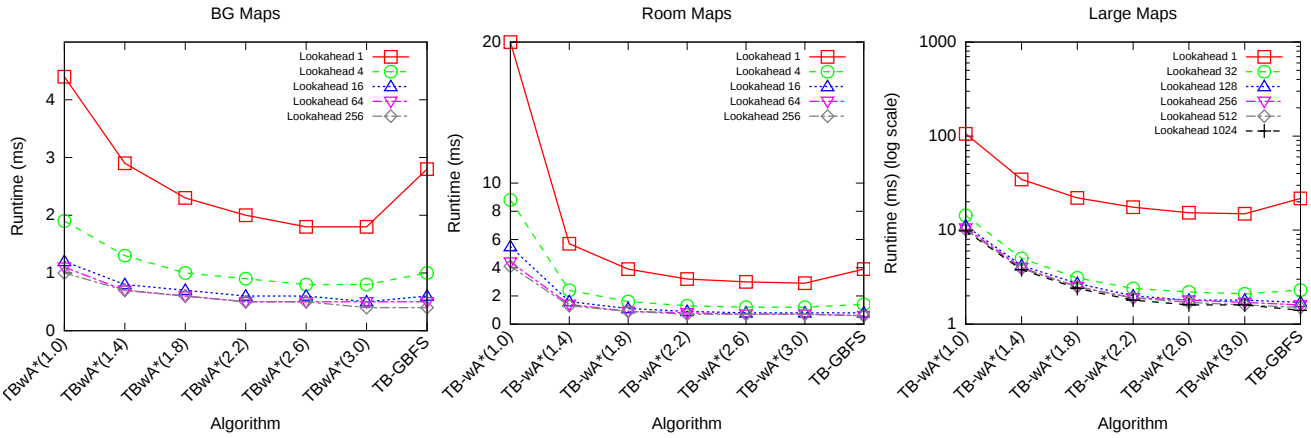

Figure 2: In static domains, search time typically decreases as w or the lookahead parameter is increased.

percentage back-moves decrease as w increases, and TB-GBFS performs the fewest back-moves among all algorithms.

Overall, if one wants to avoid scrubbing TB-GBFS is clearly the algorithm of choice. On the other hand TB-WA*($w = 3$) does significantly less scrubbing than TBA* (but a little more than TB-GBFS) and in most cases performs fewer non-optimal moves than other algorithms. Thus TB-WA*($w = 3$) seems to be the algorithm that achieves the best balance among both quality measures.

Dynamic Environments

Following an existing experimental evaluation by Aine and Likhachev (2013), we evaluated Repeated TB-WA* (henceforth RTB-WA*) with lazy restarting, and six weight values: $1.0, 1.4, 1.8, 2.2, 2.6, 3.0$ in 1000×1000 grids with a 10% obstacles placed randomly. The original map is given to the agent before the first search. After 10 moves made by the agent, $(cr/2)\%$ of the originally unblocked cells become blocked and $(cr/2)\%$ of the originally blocked cells become unblocked. We call cr *change rate*. We tested with three change rates: 1, 5, and 10. For each change rate we

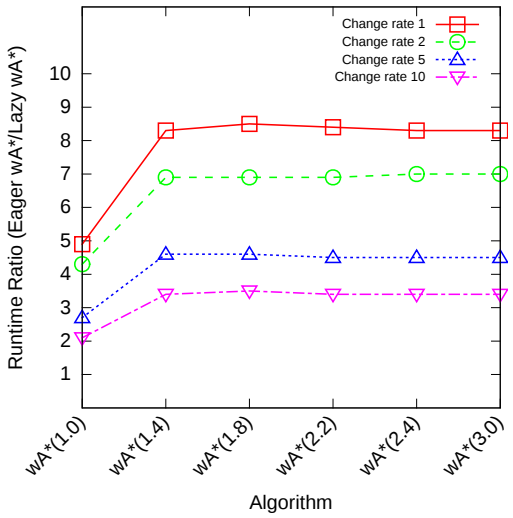

Figure 6: Lazy restart significantly outperforms eager restart in dynamic domains.

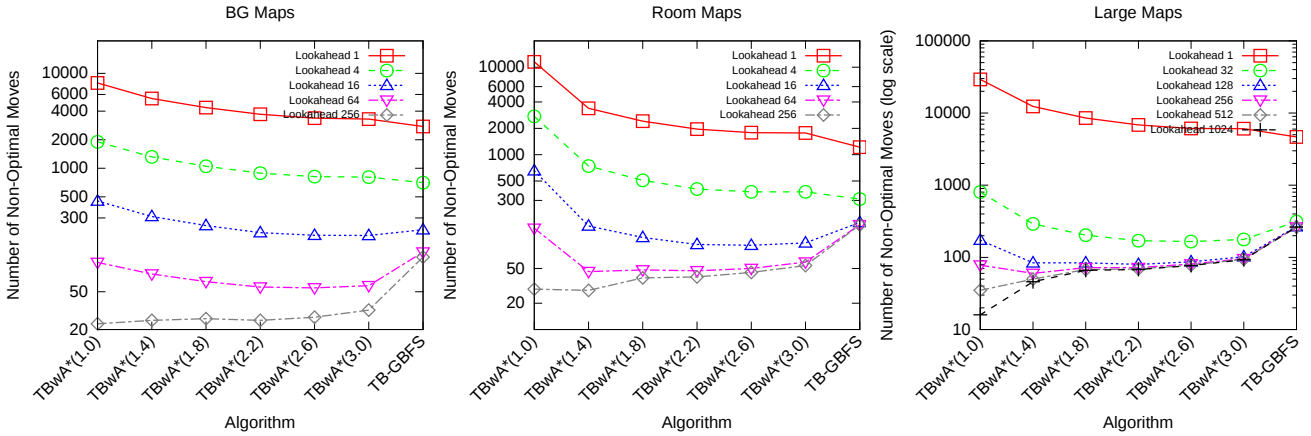

Figure 3: In static domains, as w increases solutions tend to have fewer non-optimal moves, except when the lookahead value is large.

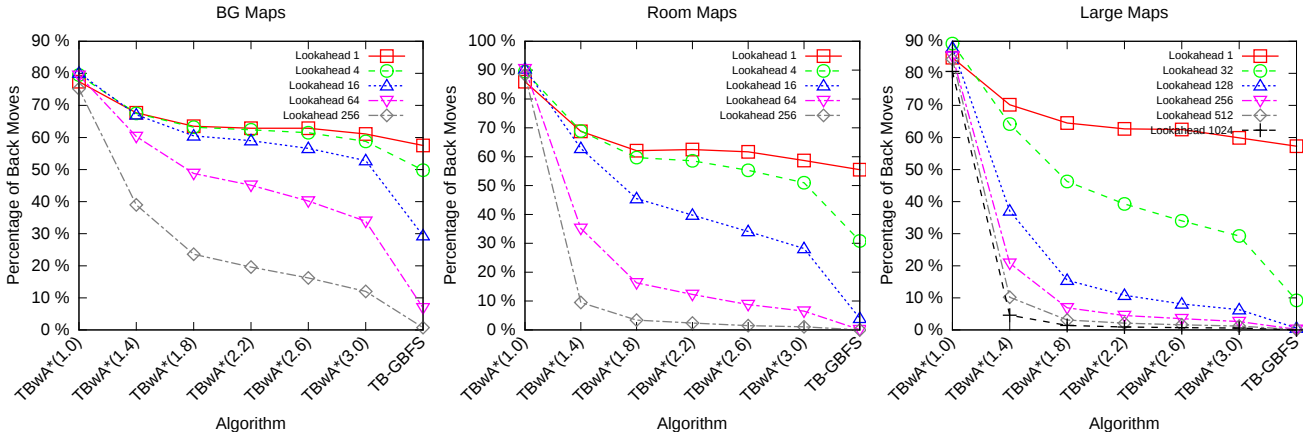

Figure 4: In static domains, as w increases solutions tend to have fewer back-moves.

generated 100 random test cases and computed the average cost and runtime.

First we compare the two restart strategies. For the comparison we use TB-WA* used with lookahead parameter set to infinity. Figure 6 shows the ratio between the runtime obtained by lazy and eager restarting for different change rates and weight values. Lazy restart yields substantially faster executions. Interestingly, eager restarting has been the standard strategy when evaluating incremental search algorithms in dynamic domains (Sun, Koenig, and Yeoh 2008; Aine and Likhachev 2013). This evaluation calls for the use of lazy restarts in future evaluations of those algorithms.

Figure 5 shows the solution cost and the runtime. For both change rates cost and runtime decrease when w increases, for all lookahead values. If $w = 1$, the best solution cost is obtained with lookahead 256 whereas best runtime is obtained with lookahead 4. On the other hand, best cost results are obtained with $w > 1$; costs are similar across all lookahead values and best runtimes are obtained when lookahead is equal to 1. We conclude RTB-WA* with lookahead equal to 1 and $w > 1$ are the best-performing algorithms because the agent makes fewer non-optimal moves and restarts are less frequent.

Conclusions and Perspectives

This paper introduced Time-Bounded Best-First Search (TB-BFS), a generalization of TBA*, suitable for solving search problems under real-time constraints. In addition, it introduced a restarting version of TB-BFS for dynamic environments, and quality measures beyond solution cost aimed at assessing solutions in the context of videogames.

In static domains we proved TB-BFS returns a solution if the problem is solvable, and, unlike many other real-time search algorithms, terminates if the problem has no solution. In addition, we proved a bound on the cost of solutions returned by TB-WA*. Given a weight w, and if the lookahead parameter is not too large, our bound suggests that TB-WA* will be significantly superior to TBA* precisely on search problems in which WA* expands significantly fewer states than A*. On the other hand our bound suggests that TB-WA* may not yield benefits in domains in which WA*—run offline—will not yield any improvements over A*.

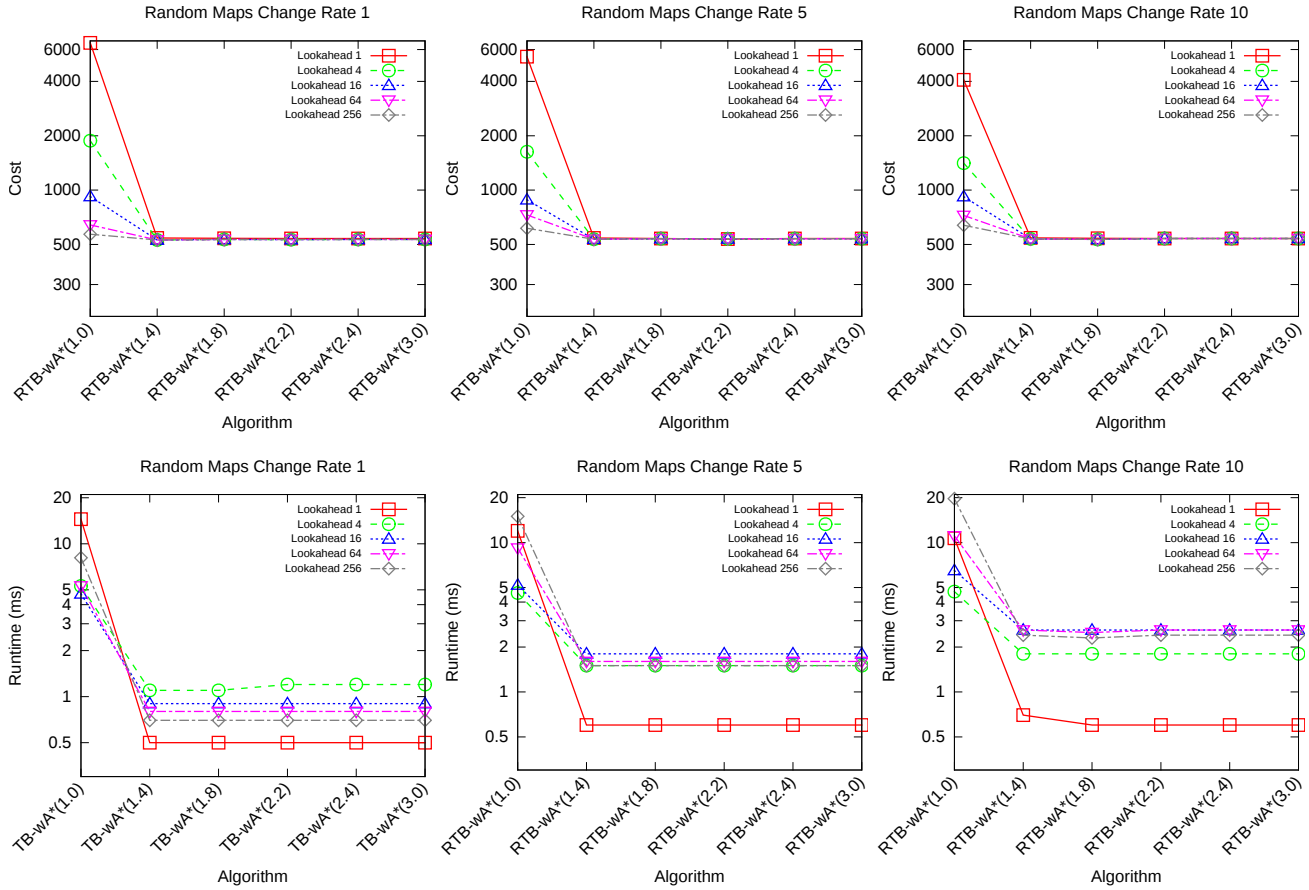

Figure 5: In dynamic domains, solution cost and search time typically decrease as w is increased.

We implemented and analyzed the performance of time-bounded algorithms on pathfinding benchmarks. Our results confirmed some of the conclusions that can be drawn from our theoretical bound. It is well known that in pathfinding, WA* may expand significantly fewer nodes than A*. Consistent with this, in our experiments, Time-Bounded versions of suboptimal algorithms like Weighted A* and Greedy Best-First Search produce significantly better solutions than those obtained by TBA*. Improvements are less noticeable when the lookahead parameter is large, as is also predicted by theory. A conclusion we draw from our experimental analysis is that TB-WA* seems to be the algorithm that achieves the best balance among quality and performance measures. For instance, TB-WA* ($w = 3$) with high lookahead values obtains good solutions very fast, performing almost no scrubbing. TB-WA* seems a good choice for pathfinding in deployed video games. In our evaluation in dynamic domains, we also observed savings in time and cost from using weights and we conclude that lazy restarting is more efficient than eager restarting.

Our findings are consistent with those obtained by Rivera, Baier, and Hernández (2013), who also obtain better solutions by using weighted heuristics in the context of real-time search. Our work adds another piece of evidence that justifies studying the incorporation of weights into other algorithms (e.g., RIBS; Sturtevant, Bulitko, and Björnsson, 2010). Other avenues of research include multiagent pathfinding settings (e.g., Standley, 2010), in which Repeated TB-BFS is applicable.

References

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

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

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

Hart, P. E.; Nilsson, N.; and 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., and Baier, J. A. 2012. Avoiding and es-

caping depressions in real-time heuristic search. *Journal of Artificial Intelligence Research* 43:523–570.

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

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

Koenig, S., and Likhachev, M. 2006. Real-time adaptive A*. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, 281–288.

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

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

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

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

Rivera, N.; Baier, J. A.; and Hernández, C. 2013. Weighted real-time heuristic search. In *Proceedings of the 11th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, 579–586.

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

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI)*.

Sturtevant, N. R.; Bulitko, V.; and 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)*, 333–340.

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

Sun, X.; Koenig, S.; and Yeoh, W. 2008. Generalized adaptive A*. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, 469–476.

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