TBA*: Time-Bounded A* [*]

Yngvi Björnsson
School of Computer Science
Reykjavik University
yngvi@ru.is

Vadim Bulitko and **Nathan Sturtevant**
Department of Computing Science
University of Alberta
{bulitko,nathanst}@cs.ualberta.ca

Abstract

Real-time heuristic search algorithms are used for planning by agents in situations where a constant-bounded amount of deliberation time is required for each action regardless of the problem size. Such algorithms interleave their planning and execution to ensure real-time response. Furthermore, to guarantee completeness, they typically store improved heuristic estimates for previously expanded states. Although subsequent planning steps can benefit from updated heuristic estimates, many of the same states are expanded over and over again. Here we propose a variant of the A* algorithm, Time-Bounded A* (TBA*), that guarantees real-time response. In the domain of path-finding on video-game maps TBA* expands an order of magnitude fewer states than traditional real-time search algorithms, while finding paths of comparable quality. It reaches the same level of performance as recent state-of-the-art real-time search algorithms but, unlike these, requires neither state-space abstractions nor pre-computed pattern databases.

1 Introduction

In this paper we study the problem of real-time search where an agent must repeatedly plan and execute actions within a constant time interval that is independent of the size of the problem being solved. This restriction severely limits the range of applicable heuristic search algorithms. For instance, static search algorithms such as A* [Hart *et al.*, 1968] and IDA* [Korf, 1985], re-planning algorithms such as D* [Stenz, 1995], and anytime re-planning algorithms such as AD* [Likhachev *et al.*, 2005] cannot guarantee a constant bound on planning time per action. LRTA* can, but with potentially low solution quality due to the need to fill in heuristic depressions [Korf, 1990; Ishida, 1992].

A common test-bed application for real-time search is path-finding, both in real-world scenarios and on video-game maps. For the latter, especially, there are strict real-time constraints as agents must react quickly regardless of map size and complexity.

[*]The support of RANNIS, NSERC and iCORE is acknowledged.

A major game developer we collaborate with imposes 1-3 ms planning limit for all simultaneously path-finding units, which is too brief for traditional real-time search algorithms to produce solutions of an acceptable quality. This has led to the development of more advanced real-time heuristic search algorithms that use various abstraction and pre-computation mechanisms for boosting their performance [Bulitko *et al.*, 2007; 2008]. However, this approach can be problematic in a dynamic game-world environment where the map structure changes during play, invalidating the pre-computed information (e.g., a new pathway emerges after trees are cut down or one disappears when a bridge is blown up).

The main contribution of this paper is a new variation of the classical A* algorithm, called Time-Bounded A* (TBA*), that is better suited for real-time environments. Empirical evaluation on video-game maps shows that the new algorithm expands an order of magnitude fewer states than traditional real-time search algorithms, while finding paths of equal quality. For example, it can achieve the same quality solutions as LRTA* in 100 times less computation per action. Alternatively, with the same amount of computation, it finds the goal after 20 times fewer actions. TBA* reaches a level of performance that is only matched by recent state-of-the-art real-time search algorithms that rely on state-space abstractions and/or pre-computed pattern databases for improved performance. However, unlike this, TBA* does not require re-computation of databases when the map changes. We present the most general form of TBA* here, although it can be seen as a paradigm for search that can be extended to different algorithms.

2 Problem Formulation

We define a heuristic search problem as a finite weighted directed graph (called *search graph*) with two states designated as the *start* and *goal*. At every time step, a search agent has a single *current state* (i.e., vertex in the search graph) changed only by taking an action (i.e., traversing an outedge of the current state). Each edge has a positive cost associated with it. A *heuristic function* (or simply *heuristic*) takes a state as input and returns an estimate on the cost to the goal state. A *search problem* is then defined as a search graph, start and goal states and a heuristic. We assume that the graph is safely explorable: the goal state can be reached from any state reachable from the start state.

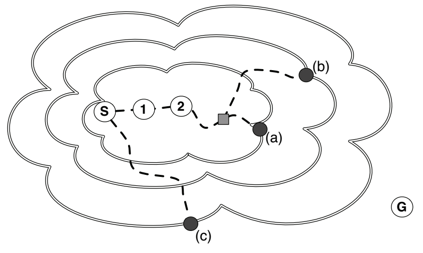

Figure 1: An example of TBA* in action.

In video-game map settings, states are defined as vacant square grid cells. Each cell is connected by an undirected edge to adjacent vacant cells. In our empirical testbed, cells have up to eight neighbors: four in the cardinal and four in the diagonal directions, with costs 1 and $\sqrt{2}$, respectively.

Each search problem is solved as follows. A search agent is deployed in the start state and takes actions (i.e., traverses edges) until it reaches the goal state. The cumulative cost of all edges traversed by an agent between the start and the goal state is called the *solution cost*. Its ratio to the shortest path cost is called *solution suboptimality*.

The *real-time property* requires a map-independent fixed upper-bound on the amount of planning by an agent between its actions. To avoid implementation and platform dependency, the amount of planning computation is commonly measured in the number of states *expanded* by an agent. A state is called expanded if all of its neighboring states are considered by an agent. The real-time cut-off is independent of the graph size (assuming a constant-bounded maximum outdegree in the graph). In this paper, we disqualify any algorithms that exceed the cut-off on any action. We evaluate search algorithms based on the number of states expanded and solution suboptimality. These measures are antagonistic insomuch as reducing suboptimality requires increased planning time per action and vice versa [Bulitko *et al.*, 2007].

3 Time-Bounded A*

LRTA*-style heuristic-updating real-time search algorithms described in the introduction satisfy the real-time constraint and are complete (i.e., find a solution for any solvable search problem as defined above). Their downside lies with extensive re-planning. For each action an LRTA*-style agent essentially starts planning from scratch. Although being somewhat more informed because of the information propagated from previous planning steps in the form of an updated heuristic, nonetheless, the agent will re-expand many of the states expanded in the previous planning steps.

In contrast, A* with a consistent heuristic never re-expands a state. However, the first action cannot be taken until an entire solution is planned. As search graphs grow in size, the planning time before the first action will grow, eventually exceeding any fixed cut-off. Consequently, A*-like algorithms violate the real-time property.

We combine both approaches in a new algorithm, Time-Bounded A* (TBA*). Namely, we achieve real-time operation while avoiding many state re-expansions of current real-time search algorithms. The algorithm expands states in an A* fashion, away from the original start state, towards the goal until the goal state is expanded. However, whereas A* plans a complete path before committing to the first action, TBA* interrupts its search after a fixed number of state expansions to act. The path from the most promising state on A* open list (the one to be expanded next) is traced back towards the start state (using A* closed list). The tracing stops early if the traced path passes through the state where the agent is currently situated, in which case the agent's next action is simply to move one step farther along the newly traced path. In the case when the agent is not on the path, the tracing continues all the way back to the start state. The path that the agent was following is rendered obsolete and the agent starts moving towards this new path. There are several ways for accomplishing that; the simplest one is to start backtracking towards the start state until crossing the newly formed path, in the worst case this happens in the start state (a more refined strategy is introduced in a later section). The basic idea is depicted in Figure 1. S is the start and G the goal, the curves represent A* open list after each expansion time-slice, the small solid circles *(a), (b), (c)* are states on the open lists with the lowest f-value. The dashed lines are the shortest paths to them. The first three steps of the agent are: $S \rightarrow 1 \rightarrow 2 \rightarrow 1$. The agent backtracks on the last step because the path to the most promising state on the outermost frontier, labeled *(c)*, did not go through state 2 where the agent was situated at the time.

A key aspect of TBA* is that it retains closed and open lists over its planning steps. Thus, on each planning step it does not start planning from scratch like LRTA* but continues with its open and closed lists from the previous planning step.

3.1 Algorithmic Details

The pseudo-code of TBA* is shown as Algorithm 1 on the next page. The arguments to the algorithm are the $start$ and $goal$ states and the search problem P. The algorithm keeps track of the current location of the agent using the variable loc. After initializing the agent location as well as several boolean variables that keep track of the algorithm's internal state (lines 1-4), the algorithm enters the main loop where it repeatedly interleaves *planning* (lines 6-20) and *execution* (lines 21-31) until the agent reaches the goal.

The planning phase proceeds in two steps: first, a fixed number (N_E) of A* state expansions are done (lines 6-8). Second, a new path to follow, $pathNew$, is generated by backtracing the steps from the most promising state on the open list back to the start state. This is done with A* closed list contained in the variable $lists$ which also stores A* open list thereby allowing us to run A* in a time-sliced fashion. The function $traceBack$ (line 13) backtraces until reaching either the current location of the agent, loc, or the $start$ state. This is also done in a time-sliced manner (i.e., no more than N_T trace steps per action) to ensure real-time performance. Thus, the backtracing process can potentially span several action steps. Each subsequent call to the $traceBack$ routine continues to build the backtrace from the front location of the path passed as an argument and adds the new locations to the front of that path (to start tracing a new path one simply resets the path passed to the routine (lines 10-12). Only when the path has been fully traced back, is it set to become the

Algorithm 1 TBA* ($start$, $goal$, P)

```
1: $solutionFound \leftarrow false$
2: $solutionFoundAndTraced \leftarrow false$
3: $doneTrace \leftarrow true$
4: $loc \leftarrow start$
5: while $loc \neq goal$ do
6:   if not $solutionFound$ then
7:     $solutionFound \leftarrow A^*(lists, start, goal, P, N_E)$
8:   end if
9:   if not $solutionFoundAndTraced$ then
10:    if $doneTrace$ then
11:      $pathNew \leftarrow lists.mostPromisingState()$
12:    end if
13:    $doneTrace \leftarrow traceBack(pathNew, loc, N_T)$
14:    if $doneTrace$ then
15:      $pathFollow \leftarrow pathNew$
16:      if $pathFollow.back() = goal$ then
17:        $solutionFoundAndTraced \leftarrow true$
18:      end if
19:    end if
20:  end if
21:  if $pathFollow.contains(loc)$ then
22:    $loc \leftarrow pathFollow.popFront()$
23:  else
24:    if $loc \neq start$ then
25:      $loc \leftarrow lists.stepBack(loc)$
26:    else
27:      $loc \leftarrow loc\_last$
28:    end if
29:  end if
30:  $loc\_last \leftarrow loc$
31:  move agent to loc
32: end while
```

new path for the agent to follow (line 15); until then the agent continues to follow its current path, $pathFollow$.

In the execution phase the agent does one of two things as follows. If the agent is already on the path to follow it simply moves one step forward along the path, removing its current location from the path (line 22).[1] On the other hand, if the agent is not on the path — for example, if a different new path has become more promising — then the agent simply starts backtracking its steps one at a time (line 25). The agent will sooner or later step onto the path that it is expected to follow, in the worst case this will happen in the $start$ state.

Note that one special case must be handled. Assume a very long new path is being traced back. In general, this causes no problems for the agent as it simply continues to follow its current path until it reaches the end of that path, and if still waiting for the tracing to finish, it simply backtracks towards the $start$ state. It is possible, although unlikely, that the agent reaches the start state before a new path becomes available, thus having no path to follow. However, as the agent must act, it simply moves back to the state it came from (line 27).

3.2 Properties

Real-time property. The number of state expansions and backtraces performed for each action step is bounded. This is sufficient to claim real-time behavior *provided* that the time it takes to expand or backtrace each state is constant-bounded. In TBA* the open and closed lists grow between action steps, so subsequent planning steps work with larger lists. However, as discussed in the next subsection, a careful choice of data-structures still enables (amortized) constant time operation.

Completeness. The algorithm expands states in the same manner as A* and is thus guaranteed to find a path from the start state to the goal provided that one exists. The algorithm does additionally guarantee that the agent will get on this solution path and subsequently follow it to the goal. This is done by having the agent backtrack towards the start state when it has no path to follow; during the backtracking process the agent is guaranteed to walk onto the solution path A* found — in the worst case this will be at the start state. TBA* is thus complete.

Memory complexity. The algorithm uses the same state-expansion strategy as A*, and consequently shares the same memory complexity: in the worst case the open and closed lists will cover the entire state space. Traditional heuristic updating real-time search algorithms face a similar worst-case scenario as they may end up having to store an updated heuristic for every state of the search graph.

3.3 Implementation Details

In this section we cover several important implementation details. First, in the pseudo-code the state-expansion and state-tracing resource bounds are shown as two different parameters, N_E and N_T, respectively. In practice we use only one resource limit, R, to be able to specify the same resource limit to other real-time search algorithms thereby making performance comparisons meaningful. For such algorithms the resource limit R will be used up entirely for state expansions, whereas in TBA* it must be shared between state expansions and backtracing operations. The number of state expansions is defined as:

$$N_E = \lfloor R \times r \rfloor$$

where $r \in [0, 1]$ is the fraction of the resource limit R to use for state expansions. The remaining resources are alloted to N_T backtracing steps as:

$$N_T = (R - N_E) \times c$$

where the multiplication constant c accounts for the relative cost of a state expansion compared to a state backtracing (e.g., a value of 10 indicates that one state expansion takes ten times more time to execute than a backtracing step). In many domains backtracing can be implemented as a much faster operation than a state expansion simply because it usually involves following a single pointer in the closed list as opposed to generating multiple successor states. Another minor enhancement not shown in the pseudo-code is that after A* finds a solution no more expansions are necessary. Thus, we then fully allocate the R limit to backtracing operations.

The choice of data structures for storing the paths ($pathFollow$ and $pathNew$) and A* is crucial for the real-time constraint. For the paths we must be able to answer membership queries in constant time (line 21). This is easily accomplished by storing all locations on each path ad-

[1]It is not necessary to keep the part of the path already traversed since it can be recovered from the closed list.

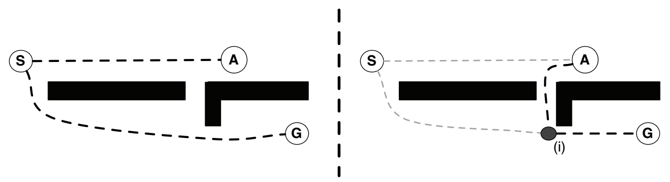

Figure 2: TBA* without and with shortcut enhancement.

ditionally in a hash table.[2] Likewise, the A* closed list is not a problem as it can be kept in a hash table indexed by state identification number. However, a standard implementation of the open list consists of not only a hash table for membership queries but also a priority queue for finding the most promising state. The priority queue is typically implemented using a heap. The insertion and deletion complexity of a heap is $O(\log n)$, where n is the heap size. In TBA* we keep growing the open list between actions, so n increases proportionally to the solution length. For practical purposes a logarithmic growth rate may be acceptable. However, to legitimately claim real-time performance, TBA* open list operations must be performed in (amortized) constant time independent of list size. This can be done by using a hash table to bucket the open list by f-values of its states [Dial, 1969; Björnsson *et al.*, 2005].

Finally, for the very first action we must guarantee that the planned path A* found can be traced back to the start state in that same planning phase for otherwise the agent would have no path to follow. This can be done in several ways, the simplest one is to call A* on this first step with a state-expansion limit of $\min(N_E, N_T)$ instead of N_E. A more sophisticated approach is to have A* monitor the longest path expanded so far, and terminate when either the number of state expansions exceeds N_E or when the number of actions on the longest path expanded equals N_T. The choice between these two strategies is unlikely to have an impact on TBA* overall performance as it affects only the first action.

3.4 Enhancements

One of the main design goals of TBA* was to keep the algorithm as simple as possible, to make it a suitable benchmark algorithm. We have extended it with two optional enhancements, both inspired by visual observations of the algorithm in practice.

The first is a more informed backtracking strategy, as the default strategy sometimes backtracks unnecessarily far. Figure 2 demonstrates a scenario where this happens. There are two corridors leading to the goal (marked with *G*) separated by a wall, but with an opening halfway. Presume that the agent (*A*) is already halfway down the left-side corridor when the optimal path, via the right corridor, is found. An agent using the basic TBA* backtracks all the way to the start location (*S*) before beginning to follow the optimal right-side path, despite there being an obvious shortcut to take through the opening in the wall. This is a contrived example and not likely to happen often in practice on typical game-world maps, however, when occurring it leads to blatantly irrational behavior. We thus extended TBA* to actively look for shortcuts on each backtracking step once an optimal path is found. Specifically, an (expansion-bounded) A* search is performed as follows: the locations on the optimal path between the goal and the intersection point with the agent's current path are inserted onto the open list with an initial g value telling their true distance to the goal. By seeding with g values the search looks for the shortcut that results in the shortest path to the goal, as opposed to simply finding the shortest way to reach the optimal path. The target of the A* search is set to the agent's current location. If the search does not reach the agent within the allotted resource limit, the backtracking step is performed as usual; if the search reaches the agent's location, however, a shorter path to the goal is constructed from the shortcut path found by the A* search and the tail of the optimal path (i.e., in the figure, from A to *(i)* and then from there to G).

The second enhancement addresses apparent indecisiveness of the agent, where it frequently switches between following two paths that alternate looking the best, resulting in it stepping back and forth repeatedly. This behavior can be alleviated by instructing the agent not to switch to a new seemingly better path unless it is significantly more promising than the one it is currently following. We use the following rule of thumb: for an agent to switch to a new path, the cumulative cost of that path (its g-value) must be at least as high as of the one that is currently being followed.

4 Empirical Evaluation

In this section we first define our empirical testbed. We then compare TBA* (without enhancements) to existing classic, contemporary state-of-the-art real-time search algorithms. We then assess the enhancements separately, and, finally, we analyze the performance of a trivial time-sliced A* that moves randomly while a complete path is being computed.

4.1 Setup

Gridworld-based path-finding is one of the most popular domains for testing real-time heuristic search algorithms. Recent literature used video-game maps as opposed to less practically interesting randomly generated gridworlds. We use three different maps modeled after game worlds from a popular real-time strategy game. The maps were scaled up to 512×512 cells to increase the problem difficulty [Sturtevant and Buro, 2005; Bulitko *et al.*, 2008]. One hundred different searches were performed on each map with start and goal locations chosen randomly, although constrained such that the optimal solution cost was between 230 and 320. Each data-point we report below is thus an average of 300 different path-finding problems (3 maps $\times$ 100 searches on each). The heuristic function used by all the algorithms is the octile distance, a natural generalization of the Manhattan distance to diagonal actions (see e.g. [Sturtevant and Buro, 2005]).

4.2 TBA* versus Other Real-Time Algorithms

We compared the performance of TBA* against several well-known real-time algorithms. A brief description of the algorithms is given below as well as their parameter settings:

[2]Hash tables with imperfect hashing guarantee *amortized* real-time operation at best. This is a common "wrinkle" in real-time heuristic search analysis and applies to most existing algorithms.

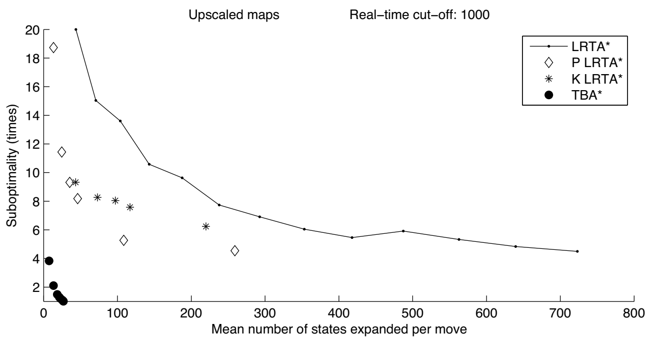

Figure 3: TBA* compared to traditional real-time algorithms.

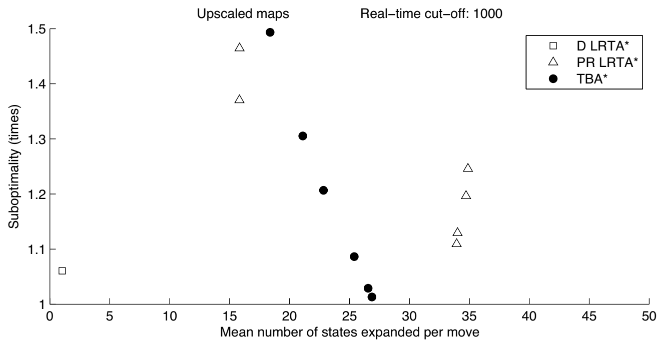

Figure 4: TBA* compared to advanced real-time algorithms.

- **LRTA*** is Learning Real-Time A* [Korf, 1990]. For each action it conducts a breadth-first search of a fixed depth d around the agent's current state. Then the first action towards the best depth d state is taken and the heuristic of the agent's previous state is updated using Korf's mini-min rule. We used $d \in \{4, 5, \ldots, 16\}$.
- **P LRTA*** is Prioritized LRTA* – a variant of LRTA* proposed by Rayner *et al.* (2007). It uses a lookahead of depth 1 for all actions. However, for every state whose heuristic value is updated, all its neighbors are put onto a priority queue, sorted by the magnitude of the update, for deciding the propagation order. The control parameter (queue size) took on $\{10, 20, 30, 40, 100, 250\}$.
- **K LRTA*** is a variant of LRTA* proposed by Koenig (2004). Unlike the original LRTA*, it uses A*-shaped lookahead search space and updates heuristic values for all states within it using Dijkstra's algorithm. The number of states that K LRTA* expands per action took on $\{10, 20, 30, 40, 100, 250, 500, 1000\}$.
- **TBA*** is our Time-Bounded A*; the resource limit R took on $\{10, 25, 50, 75, 100, 500, 1000\}$ but the values of r and c were fixed at 0.9 and 10 respectively.

In Figure 3 the run-time efficiency of the algorithms is plotted. The x-axis represents the amount of work done in terms of the mean number of states expanded per action, whereas the y-axis shows the quality of the solution found relative to an optimal solution (e.g., a value of four indicates that a solution path four times longer than optimal was found). Each point in the figure represents a run of one algorithm with a fixed parameter setting. The closer a point is to the origin the better performance it represents. Note that we imposed a constraint on the parameterization: if the worst-case number of states expanded per action exceeded a cut-off of 1000 states[3] then the particular parameter setting was excluded from consideration.

The topmost curve in Figure 3 shows the performance of LRTA* (for different lookahead depth values), the diamonds plot the performance of P LRTA* and the asterisks plot K LRTA* performance. The contemporary P LRTA* and K LRTA* easily outperform the classic LRTA*, finding equally good solutions with approximately half the resources. However, the new TBA* algorithm dominates all others, performing more than an order of magnitude better.

Given the easy victory against the classic and contemporary algorithms, we pitted TBA* against two state-of-the-art search algorithms. They both use state abstraction and precomputation to improve performance and have been shown particularly effective in path-finding on video-game maps. The algorithms and their parameter settings are:

- **PR LRTA*** is Path Refinement Learning Real-Time Search [Bulitko *et al.*, 2007]. The algorithm runs LRTA* with a fixed search depth d in an abstract space (abstraction level ℓ in a clique abstraction hierarchy [Sturtevant and Buro, 2005]) and refines the first action using a corridor-constrained A* running on the original ground-level map. The control parameters are as follows: abstraction level $\ell \in \{3, 4, \ldots, 7\}$, LRTA* lookahead depth $d \in \{1, 3, 5, 10, 15\}$ and LRTA* heuristic weight $\gamma \in \{0.2, 0.4, 0.6, 1.0\}$.
- **D LRTA*** is a variant of LRTA* equipped with dynamic search depth and intermediate goal selection [Bulitko *et al.*, 2008]. For each map optimal search depths as well as intermediate goals (or waypoints) were precomputed beforehand and stored in pattern databases. State abstraction was used to reduce the amount of precomputation. We used the abstraction level of 3 (higher levels of abstraction exceeded the real-time computation cut-off threshold of 1000 nodes per action).

Figure 4 presents the results. To focus on the high-performance area close to the center of origin, we limited the axis limits and, as a result, displayed only a subset of PR LRTA* and D LRTA* parameter combinations. In contrast to the traditional real-time search algorithms, TBA* performs on par with these state-of-the-art algorithms. However, unlike these, it requires neither state-space abstractions nor precomputed pattern databases. This has the advantages of making it both much simpler to implement and better poised for application in non-stationary search spaces, a common condition in video-game map path-finding where other agents or newly constructed buildings can block a path. For example, the data point that is provided for D LRTA*, although showing a somewhat better computation vs. suboptimality tradeoff than TBA*, is at the expense of extensive pre-computations that takes hours for even a single map.

[3]This is approximately the number of states that an optimized implementation of real-time search algorithms is allowed to expand for planning each action in video games.

Table 1: Benefits of early acting in TBA*.

R	**10**	**25**	**50**	**75**	**100**	**200**	**500**	**1000**
E	3.83	2.10	1.49	1.31	1.21	1.09	1.03	1.01
L	4.21	2.32	1.64	1.43	1.30	1.15	1.06	1.02

4.3 TBA* Performance Analysis

The TBA* algorithm always follows the most promising path towards the goal. When a new such path emerges the algorithm, in its basic form, causes the agent simply to backtrack its steps until reaching the new path. A major appeal of this strategy is that it is both conceptually simple and easy to implement (i.e., the already existing closed list can be used for the backtracking). Keeping things simple fits well with the goal of this work to introduce a simple, yet powerful, real-time algorithm that can (among other things) serve as a benchmark for real-time search algorithms in domains where memory is not of a primary concern.

One can think of an even simpler strategy: instead of following a best path the agent simply moves back and forth between the start state and a randomly chosen neighbor until a complete solution path is found by A*. Although such a strategy would be completely unacceptable in computer games as the agent will appear not to follow user command, it is nonetheless interesting to compare the computational efficiency of such a strategy to the one TBA* uses. Table 1 compares suboptimality of paths generated by "early acting" of TBA* (row E) to "late acting" of a delayed A* (row L) for different amount of planning time allowed per action (R). The "late acting" strategy results in approximately 10% more costly paths than the "early acting" one, although that difference diminishes with higher resource limits (explained by fewer action steps until a solution is found). This result confirms that the agent on average already makes headway towards the goal before a solution path is found.

We also investigated how the two enhancements we introduced earlier affect the search (aforementioned experiments used the basic version of TBA*). Having the agent look out for a shortcut while backtracking resulted in insignificant improvements in average solution suboptimality. Analyzing the data revealed that despite multiple planning slices — $\{700, 70, 7\}$ for R=$\{10, 100, 1000\}$ respectively — in only about 5% of the cases the agent was following an alternative path when the optimal one was found, and in most of these cases no beneficial shortcuts existed. Nonetheless, this improvement is important for video-game path-finding as unnecessarily long backtracking can be visually jarring and even a single incident can break the player's immersion.

Our rule of thumb for path switching (i.e., the second enhancement) not only alleviated the indecisive behavior of the agent, but also returned slightly shorter paths. For the smaller R values (100 or less) the paths were about 2% shorter, whereas for larger R values the stepping back and forth was less of an issue in the first place.

5 Conclusions

The traditional approach to real-time heuristic search is for the agent to plan from scratch at each action step, and update heuristic values to ensure progress. This approach, although both real-time and complete, comes at the cost of extensive re-planning. In this paper we introduced TBA*, an adaptation of the A* algorithm that guarantees a real-time response.

In our empirical evaluation in the domain of path-finding on video-game maps the new algorithm outperformed classic and contemporary real-time algorithms by a large margin. Furthermore, it reached the same level of performance as state-of-the-art real-time search algorithms. However, unlike these, TBA* requires neither state space abstraction nor pre-computed pattern databases. This makes it not only substantially simpler to implement but also better poised for application to non-stationary problems, such as path-finding on dynamically changing maps in video games.

Finally, the idea behind TBA* can be viewed as a general approach to time-slicing heuristic search algorithms, and is not limited to A*.

References

[Björnsson *et al.*, 2005] Y. Björnsson, M. Enzenberger, R. Holte, and J. Schaeffer. Fringe search: Beating A* at pathfinding on computer game maps. In *IEEE Symp. on Comp. Intelligence in Games*, pages 125–132, 2005.

[Bulitko *et al.*, 2007] V. Bulitko, N. Sturtevant, J. Lu, and T. Yau. Graph Abstraction in Real-time Heuristic Search. *JAIR*, 30:51–100, 2007.

[Bulitko *et al.*, 2008] V. Bulitko, M. Luštrek, J. Schaeffer, Y. Björnsson, and S. Sigmundarson. Dynamic Control in Real-Time Heuristic Search. *JAIR*, 32:419–452, 2008.

[Dial, 1969] R. B. Dial. Shortest-path forest with topological ordering. *Commun. ACM*, 12(11):632–633, 1969.

[Hart *et al.*, 1968] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cyber.*, 4(2):100–107, 1968.

[Ishida, 1992] T. Ishida. Moving target search with intelligence. In *AAAI*, pages 525–532, 1992.

[Koenig, 2004] S. Koenig. A comparison of fast search methods for real-time situated agents. In *AAMAS*, pages 864–871, 2004.

[Korf, 1985] R.E. Korf. Depth-first iterative deepening : An optimal admissible tree search. *AIJ*, 27(3):97–109, 1985.

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

[Likhachev *et al.*, 2005] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *ICAPS*, pages 262–271, 2005.

[Rayner *et al.*, 2007] D. C. Rayner, K. Davison, V. Bulitko, K. Anderson, and J. Lu. Real-time heuristic search with a priority queue. In *IJCAI*, pages 2372 – 2377, 2007.

[Stenz, 1995] A. Stenz. The focussed D* algorithm for real-time replanning. In *IJCAI*, pages 1652–1659, 1995.

[Sturtevant and Buro, 2005] N. Sturtevant and M. Buro. Partial pathfinding using map abstraction and refinement. In *AAAI*, pages 1392–1397, 2005.