Real-time Heuristic Search for Pathfinding in Video Games

Vadim Bulitko* bulitko@ualberta.ca
Yngvi Björnsson† yngvi@ru.is
Nathan R. Sturtevant‡ sturtevant@cs.du.edu
Ramon Lawrence§ ramon.lawrence@ubc.ca

July 7, 2010

Abstract

Game pathfinding is a challenging problem due to a limited amount of per-frame CPU time commonly shared among many simultaneously pathfinding agents. The challenge is rising with each new generation of games due to progressively larger and more complex environments and larger numbers of agents pathfinding in them. Algorithms based on A* tend to scale poorly as they must compute a complete, possibly abstract, path for each agent before the agent can move. Real-time heuristic search algorithms satisfy a constant bound on the amount of planning per move, independent of problem size. These algorithms are thus a promising approach to large scale multi-agent pathfinding in video games. However, until recently, real-time heuristic search algorithms universally exhibited a visually unappealing "scrubbing" behavior by repeatedly revisiting map locations. This had prevented their adoption by video game developers. In this chapter we review three modern search algorithms which address the "scrubbing" problem in different ways. Each algorithm presentation is complete with an empirical evaluation on game maps.

1 Introduction and Related Work

Heuristic search is a core area of Artificial Intelligence (AI) research and its algorithms have been widely used in planning, game-playing and agent control. In this chapter we are interested in *real-time* heuristic search algorithms that satisfy a constant upper bound on the amount of planning per action, independent of problem size. This property is important in a number of applications including autonomous robots and agents in video games. A common problem in video games is searching for a path between two locations. In most games, agents are expected to act quickly in response to player's commands and other agents' actions. As a result, many game companies impose a constant time limit on the amount of path planning per move[1] (e.g., one millisecond for *all* simultaneously moving agents).

While in practice this time limit can be satisfied by limiting problem size *a priori*, a scientifically more interesting approach is to impose a constant per-action time limit *independent* of the problem size. Doing so severely limits the range of applicable heuristic search algorithms. For instance, static search algorithms such as A* [15], IDA* [24] and PRA* [39, 38], re-planning algorithms such as D* [37], anytime algorithms such as ARA* [27] and anytime re-planning algorithms such as AD* [26] cannot guarantee a constant bound on planning time per action. This is because all of them produce a complete,

*Department of Computing Science, University of Alberta, Edmonton, Alberta T6G 2E8, Canada
†School of Computer Science, Reykjavik University, Menntavegi 1, IS-101 Reykjavik, Iceland
‡Department of Computer Science, University of Denver, Denver, Colorado 80208, USA
§University of British Columbia Okanagan, 3333 University Way, Kelowna, British Columbia, V1V 1V7, Canada
[1]Henceforth we will use the terms *action* and *move* synonymously.

possibly abstract, solution before the first action can be taken. As the problem increases in size, their planning time will inevitably increase, exceeding any *a priori* finite upper bound.

Real-time search addresses the problem in a fundamentally different way. Instead of computing a complete, possibly abstract, solution before the first action is taken, real-time search algorithms compute (or plan) only a few first actions for the agent to take. This is usually done by conducting a lookahead search of a fixed depth (also known as "search horizon", "search depth" or "lookahead depth") around the agent's current state and using a heuristic (i.e., an estimate of the remaining travel cost) to select the next few actions. The actions are then taken and the planning-execution cycle repeats [25]. Since the goal state is not seen in most such local searches, the agent runs the risks of heading into a dead end or, more generally, selecting suboptimal actions. To address this problem, most real-time heuristic search algorithms update (or learn) their heuristic function over time.

The learning process has precluded real-time heuristic search agents from being widely deployed for pathfinding in video games. The problem is that such agents tend to "scrub" (i.e., repeatedly revisit) the state space due to the need to fill in heuristic depressions [19]. As a result, solution quality can be quite low and, visually, the scrubbing behavior is perceived as irrational.

Since the seminal work on LRTA* [25], researchers have attempted to speed up the learning process. Most of the resulting algorithms can be described by the following four attributes:

The *local search space* is the set of states whose heuristic costs are accessed in the planning stage. The two common choices are full-width limited-depth lookahead [25, 33, 35, 34, 14, 16, 17, 36, 31] and A*-shaped lookahead [21, 23]. Additional choices are decision-theoretic based shaping [32] and dynamic lookahead depth-selection [7, 29]. Finally, searching in a smaller, abstracted state has been used as well [13].

The *local learning space* is the set of states whose heuristic values are updated. Common choices are: the current state only [25, 33, 35, 34, 14, 7], all states within the local search space [21, 23] and previously visited states and their neighbors [16, 17, 36, 31].

A *learning rule* is used to update the heuristic costs of the states in the learning space. The common choices are mini-min [25, 35, 34, 16, 17, 36, 31], its weighted versions [33], max of mins [7], modified Dijkstra's algorithm [21], and updates with respect to the shortest path from the current state to the best-looking state on the frontier of the local search space [23]. Additionally, several algorithms learn more than one heuristic function [32, 14, 33].

The *control strategy* decides on the move following the planning and learning phases. Commonly used strategies include: the first move of an optimal path to the most promising frontier state [25, 14, 16, 17], the entire path [7], and backtracking moves [35, 34, 7, 36].

Given the multitude of proposed algorithms, unification efforts have been undertaken. In particular, [10] suggested a framework, called Learning Real Time Search (LRTS), to combine and extend LRTA* [25], weighted LRTA* [33], SLA* [35], SLA*T [34], and to a large extent, γ-Trap [7].

A breakthrough in performance came with D LRTA* [12] which, for the first time in real-time heuristic search, used automatically selected local subgoals instead of the global goal. The subgoal selection mechanism has later been refined in kNN LRTA*, which we review in this chapter.

In this chapter we review the following three modern real-time heuristic search algorithms:

kNN LRTA* [8, 9] employs a nearest-neighbour algorithm over a database of solved cases. It introduced the idea of compressing a solution path into a series of subgoals so that each can be "easily" reached from the previous one. In doing so, it uses hill-climbing as a proxy for the notion of "easy reachability by LRTA*".

If precomputing a database of solved cases and compressing them into subgoals is not feasible then one can use the following two modern real-time heuristic search algorithms.

TBA* [2] is a time-bounded variant of the classic A*. Unlike A* that plans a complete path before committing to the first action, TBA* interrupts its planning periodically to act. Because initially a complete path to the goal is unknown, the agent instead moves towards the most promising state on the open list, backtracking its steps as necessary. This interleaving of planning and acting is done in such a way

that both real-time behavior and completeness are ensured. Among the attractions of this algorithm are its simplicity and broad applicability as well as the fact that reasonable solution quality and real-time performance is achieved without the need for precomputations or state-space abstractions.

RIBS [40] takes a different approach to learning real-time search. Instead of learning a heuristic estimate of the distance from an arbitrary state to the goal as most algorithms have traditionally done, RIBS learns accurate distances from the start state. This approach has just recently been explored, and more work is required to deploy this algorithm in commercial games. But, the study of RIBS has lead to critical insights in the performance of real-time algorithms and approaches that are likely to be successful.

The rest of the chapter is organized as follows. In Section 2 we formulate the problem. Section 3 presents three classic algorithms that serve as the core to kNN LRTA*, TBA* and RIBS which are reviewed in Sections 5, 6 and 7, respectively. Finally, we discuss applications beyond pathfinding in Section 8 and conclude the chapter.

2 Problem Formulation

We define a heuristic search problem as an undirected graph containing a finite set of states (vertices) and weighted edges, with a single state designated as the *goal state*. At every time step, a search agent has a single *current state*, a vertex in the search graph, and takes an *action* (or makes a *move*) by traversing an out-edge of the current state. By traversing an edge between states s_1 and s_2 the agent changes its current state from s_1 to s_2. We say that a state is *visited* by the agent if and only if it is the agent's current state at some point of time. As it is usual in the field of real-time heuristic search, we assume that path planning happens *between* the moves (i.e., the agent does not think while traversing an edge). The "plan a move" - "travel an edge" loop continues until the agent arrives at its goal state, thereby solving the problem.

Each edge has a positive cost associated with it. The total cost of edges traversed by an agent from its start state until it arrives at the goal state is called the *solution cost*. We require algorithms to be *complete* (i.e., produce a path from start to goal in a finite amount of time if such a path exists). In order to guarantee completeness for real-time heuristic search we make the assumption of safe explorability of our search problems. Specifically, all edge costs are finite and for any states s_1, s_2, s_3, if there is a path between s_1 and s_2 and there is a path between s_1 and s_3 then there is also a path between s_2 and s_3.

Formally, all algorithms discussed in this chapter are applicable to any such heuristic search problem. To keep the presentation focused and intuitive we use a particular type of heuristic search problems, video-game pathfinding in grid worlds, for the rest of the chapter. In video-game map settings, states are vacant square grid cells. Each cell is connected to four cardinally (i.e., west, north, east, south) and four diagonally neighboring cells. Outbound edges of a vertex are moves available in the corresponding cell and in the rest of the chapter we will use the terms *action* and *move* interchangeably. The edge costs are *defined* as 1 for cardinal moves and 1.4 for diagonal moves.[2]

An agent plans its next action by considering states in a local search space surrounding its current position. A *heuristic function* (or simply *heuristic*) estimates the (remaining) travel cost between a state and the goal. It is used by the agent to rank available actions and select the most promising one. Furthermore, we consider only *admissible* and *consistent* heuristic functions which do not overestimate the actual remaining cost to the goal and whose difference in values for any two states does not exceed the cost of an optimal path between these states. In this chapter we use *octile distance* – the minimum cumulative edge cost between two vertices ignoring map obstacles – as our heuristic. This heuristic is admissible and consistent. An agent can modify its heuristic function in any state to avoid getting stuck in local minima of the heuristic function, as well as to improve its action selection with experience.

We evaluate the algorithms presented in this chapter with respect to several performance measures.

[2] We use 1.4 instead of the Euclidean $\sqrt{2}$ to avoid errors in floating point computations.

First, we measure mean *planning time* per action in terms of both the number of states expanded[3] as well as the CPU time. Also note that while total planning time per problem is important for non-real-time search, it is irrelevant in video game pathfinding as we do not compute an entire path outright.

The second performance measure of our study is *sub-optimality* defined as the ratio of the solution cost found by the agent to the minimum solution cost minus one and times 100%. To illustrate, suboptimality of 0% indicates an optimal path and suboptimality of 50% indicates a path 1.5 times as costly as the optimal path. We also measure the precomputation time for kNN LRTA* as well as the memory requirements of all three algorithms.

3 The Core Algorithms

TBA*, RIBS and kNN LRTA* presented later in this chapter build on three classic heuristic search algorithms: A* [15], IDA* [24] and LRTA* [25]. We briefly review these algorithms and discuss their drawbacks for real-time heuristic search below.

3.1 A*

The classic A* algorithm [15] is a fundamental algorithm for pathfinding. Given a start state s and a goal state g, it finds a least-cost path between the two states. It is a best-first search algorithm, and uses a distance-plus-cost-estimate function to determine which state to expand next. The cost function, denoted $f(n)$, consists of two parts: $f(n) = g(s,n) + h(n,g)$ where $g(s,n)$ is the distance of the shortest path found so far between the start state s and state n, and $h(n,g)$ is the heuristic estimate of the distance cost of traveling from state n to the goal g. The algorithm uses two containers to keep track of its search progress: the *open list* storing states that have been encountered but not expanded yet, and the *closed list* storing states already expanded. The algorithm iteratively picks the state from the open list with the lowest f-cost, expands the state, and places its children on the open list. To determine whether a child state goes into the open list, it cannot already be on the closed list or on the open list with a lower cost. The state just expanded is moved to the closed list. The role of the closed list is both to avoid state re-expansions and to reconstruct the solution path once the goal is found. This continues until the goal state is removed from the open list, in which case the solution path is reconstructed from the closed list.

The algorithm is complete, finds an optimal solution when used with an admissible heuristic, and never re-expands states given a consistent heuristic.

3.2 Iterative Deepening A* (IDA*)

Early researchers noticed that A* could not solve large problems because it would run out of memory. IDA* [24] was thus developed as an alternate algorithm that could find optimal solutions, like A*, but that would only require memory usage linear in the cost of the solution. Most combinatorial puzzles, which were the original focus of IDA*, have state spaces exponential in the solution cost, and so are a natural fit for IDA*. Henceforth, we will call such problems *exponential domains*.

One way to understand how IDA* works is to contrast it to how A* works. Given a consistent heuristic, the lowest f-cost of any state in A*'s open list will monotonically increase during search. Imagine that we grouped states according to their cost when expanded by A*. For instance, all the states with cost 12 might be expanded first, followed by the states with cost 14, and so on. We demonstrate this in Figure 1, showing contours that delineate states of each successive cost.

IDA* will first expand these groups of states in the same order as A* (modulo tie-breaking among states with equal f-cost) but will subsequently revisit states in subsequent iterations of the algorithm. It does this because it does not maintain an open list. Instead, it performs multiple depth-first searches, with each search bounded by the best f-cost which has yet to be explored. All the states of a particular cost are

[3] A state is called expanded if all of its immediate children are generated.

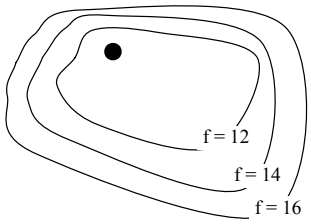


Figure 1: IDA* search contours: IDA* performs multiple depth-first searches within each successive cost frontier found during search.

explored before the next iteration begins anew. In exponential domains such as common combinatorial puzzles, the largest number of states will be expanded in the last iteration, amortizing away the cost of the previous iterations. Because it can be expensive to maintain an open list, IDA* can be faster than A* in practice.

IDA* works best in exponential domains where the state space does not contain many cycles. It might, therefore, seem that IDA* is not well suited to grid-based worlds. These domains are usually *polynomial*, as the number of states in a map grows as a polynomial function of length and/or width of the map. Additionally, there are many cycles on grid-based maps. Surprisingly, IDA* can indeed be adapted to perform real-time heuristic search in such domains, as we show below.

3.3 Learning Real-Time A* (LRTA*)

The core of most real-time heuristic search algorithms is an algorithm called Learning Real-Time A* (LRTA*) [25]. It is shown in Figure 2 and operates as follows. As long as the goal state $s_{\text{global goal}}$ is not reached, the algorithm interleaves planning and execution in lines 4 through 7. In our generalized version we added a new step at line 3 for selecting goal s_{goal} individually at each execution step (the original algorithm uses $s_{\text{global goal}}$ at all times). In line 4, a cost-limited breadth-first search with duplicate detection is used to find frontier states with cost up to $g_{\max}$ away from the current state s. For each frontier state $\hat{s}$, its value is the sum of the cost of a shortest path from s to $\hat{s}$, denoted by $g(s,\hat{s})$, and the estimated cost of a shortest path from $\hat{s}$ to s_{goal} (i.e., the heuristic cost $h(\hat{s}, s_{\text{goal}})$). The state that minimizes the sum is identified as s' in line 5. Ties are broken in favour of higher g costs[4]. Remaining ties are broken in a fixed order. The heuristic value of the current state s is updated in line 6 (we keep separate heuristic tables for the different goals and we never decrease heuristics). Finally, we take one step towards the most promising frontier state s' in line 7.

LRTA* is a special case of value iteration or real-time dynamic programming [1] and has a problem that has prevented its use in video game pathfinding. Specifically, it updates a single heuristic value per move on the basis of heuristic values of near-by states. This means that when the initial heuristic values are overly optimistic (i.e., too low), LRTA* will frequently revisit these states multiple times, each time making updates of a small magnitude. This behavior is known as "scrubbing" and appears highly irrational to an observer. Unlike some combinatorial puzzles (e.g., the sliding tile puzzle), deep heuristic depressions are common in pathfinding due to dead ends and corners.

There are two fundamental approaches to address problems related to heuristic inaccuracies. First, one can use a more accurate heuristic. Second, one can increase the depth of the lookahead (i.e., by increasing the $g_{\max}$ parameter in LRTA*) to compensate for heuristic inaccuracies. Deeper lookaheads have been generally found beneficial in real-time heuristic search [25], though lookahead pathologies (i.e., detrimental effects of deeper lookahead on solution optimality) have been observed as well [11, 6, 28, 29].

[4] In the rest of the chapter we use the terms *cost* and *value* interchangeably whenever we refer to f and g functions on states.

LRTA*($s_{\text{start}}, s_{\text{global goal}}, g_{\max}$)

```
1  s ← s_start
2  while s ≠ s_global goal do
3      select a subgoal s_goal
4      generate successor states of s up to g_max cost, generating a frontier
5      find a frontier state s' with the lowest g(s,s') + h(s',s_goal)
6      update h(s,s_goal) to g(s,s') + h(s',s_goal)
7      change s one step towards s'
8  end while
```

Figure 2: LRTA* algorithm extended with dynamic subgoal selection.

kNN LRTA* takes the former approach and effectively improves the heuristic quality by computing it with respect to a near-by subgoal as opposed to a distant global goal. This is done in an automated fashion as presented below.

4 The Three Modern Algorithms

The three real-time search algorithms discussed in this chapter use A*, IDA* or LRTA* as their core and enhance them in a number of ways. We review them below. The space constraints preclude us from presenting technical details. Thus we will focus on the key ideas, the underlying intuition and support them with highlights of empirical evaluation. We refer the reader to the original publications for additional details [9, 2, 40].

5 k Nearest Neighbors LRTA* (kNN LRTA*)

If an agent is expected to solve a number of problems on the same search graph then it can make sense to analyze the graph and precompute certain information before attempting to solve the first problem. In the following, we describe one such type of precomputation used in kNN LRTA*.

5.1 kNN LRTA*: Off-line Subgoal Precomputation

It has been observed in the literature that common heuristic functions are not uniformly inaccurate [30]. Namely, they tend to be more accurate closer to the goal state and less accurate farther away. The intuition for this fact is as follows: heuristic functions usually ignore certain constraints of the search space. For instance, the Manhattan distance heuristic in a sliding tile puzzle would be perfectly accurate if the tiles could pass through each other. Likewise, the octile distance on a map ignores obstacles. The closer a state is to a goal the fewer constraints a heuristic function is likely to ignore and, as a result, the more accurate (i.e., closer to the optimal solution cost) the heuristic is likely to be.

We can use these observations to select subgoals dynamically. The idea is straightforward: if being far from the goal leads to grossly inaccurate heuristic values, let us move the goal closer to the agent, thereby improving heuristic accuracy. We can do this by computing the heuristic function with respect to an intermediate, and thus nearby, goal as opposed to a distant global goal — the final destination of an agent. Since an intermediate goal is closer than the global goal, the heuristic values of states around an agent will likely be more accurate. Once the agent gets to an intermediate goal, the next intermediate goal is selected so that the agent makes progress towards its actual global goal. Such dynamic subgoal selection can be carried out by using a precomputed subgoal database as described below.

Intuitively, if an LRTA*-controlled agent is in the state s going to the state s_{goal} then the best subgoal is the state $s_{\text{ideal subgoal}}$ that resides on an optimal path between s and s_{goal} and can be reached by LRTA* along an optimal path with no state revisitation. Given that there can be multiple optimal paths between two states, it is unclear how to computationally efficiently detect the LRTA* agent's deviation from an optimal path *immediately after it occurs*.

On the positive side, detecting state revisitation can be done computationally efficiently by running a simple greedy hill-climbing agent.[5] This is based on the fact that if a hill-climbing agent can reach a state b from a state a without encountering a local minimum or a plateau in the heuristic then an LRTA* agent will travel from a to b without state revisitation. Thus, we propose an efficiently computable approximation to $s_{\text{ideal subgoal}}$. Namely, we define the subgoal for a pair of states s and s_{goal} as the state $s_{\text{kNN LRTA* subgoal}}$ farthest along an optimal path between s and s_{goal} that can be reached by a simple hill-climbing agent. In summary, we select subgoals to eliminate any scrubbing but do not guarantee that the LRTA* agent keeps on an optimal path between the subgoals. In practice, however, only a tiny fraction of our subgoals are reached by the hill-climbing agent suboptimally and even then the suboptimality is negligible.

This approximation to the ideal subgoal allows us to effectively compute a series of subgoals for a given pair of start and goal states. Intuitively, we *compress* an optimal path into a series of key states such that each of them can be reached from its predecessor without scrubbing. The compression allows us to save a large amount of memory without much impact on time-per-move. Indeed, hill-climbing from one of the key states to the next requires inspecting only the immediate neighbors of the current state and selecting one of them greedily.

However, it is still infeasible to compute and then compress an optimal path between *every* two distinct states in the original search space. We solve this problem by compressing only a pre-determined fixed number of optimal paths between random states off-line.

5.2 kNN LRTA*: On-line Search

On-line, kNN LRTA*, tasked with going from s to s_{goal}, retrieves the most similar compressed path from its database and uses the associated subgoals. We define (dis-)similarity of a database path to the agent's current situation as the maximum of the heuristic distances between s and the path's beginning and between s_{goal} and the path's end. We use maximum because we would like *both* ends of the path to be heuristically close to the agent's current state and the goal respectively. Indeed, the heuristic distance ignores walls and thus a large heuristic distance to the path's either end tends to make that end hill-climbing unreachable.

Note that high similarity (i.e., both distances being low) does not guarantee that the path will be useful to the kNN LRTA* agent. For instance, the beginning of the path can be heuristically very close to the agent but on the other side of a long wall, making it unreachable without a lot of learning and the associated scrubbing. To address this problem we compliment the fast-to-compute similarity metric with more computationally demanding hill-climbing reachability checks as detailed below.

We illustrate this intuition with a simple example. Figure 3 shows kNN LRTA* operation off-line. On this map, two random start and goal pairs are selected and for each pair an optimal path is computed between the start and goal. Then each path is compressed into a series of subgoals such that each of the subgoals can be reached from the previous one via hill-climbing. The path from S_1 to G_1 is compressed into two subgoals and the other path is compressed into a single subgoal.

Once this database of two records is built, kNN LRTA* can be tasked with solving a problem on-line. In Figure 4 it is tasked with going from the state S to the state G. The database is scanned and similarity between (S,G) and each of the two database records is determined. The records are sorted by

[5] In each state such a simple greedy hill-climbing agent moves to the immediate neighbor with the lowest f-cost. It gives up when all children have their h-cost greater than or equal to the h-cost of the agent's current state.

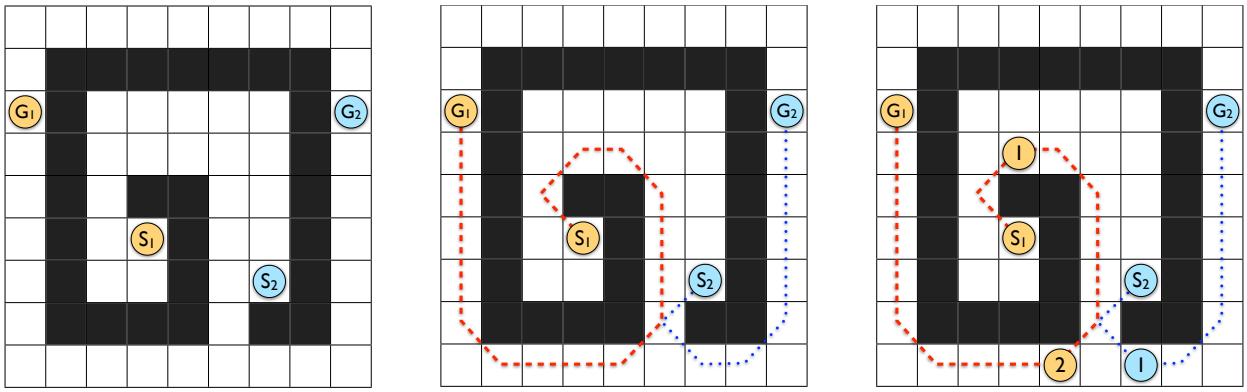


Figure 3: Example of kNN LRTA* off-line operation. **Left:** two subgoals (start,goal) pairs are chosen: (S_1, G_1) and (S_2, G_2). **Center:** optimal paths between them are computed by running A*. **Right:** the two paths are compressed into a total of three subgoals.

their similarity: (S_1, G_1) followed by (S_2, G_2). Then the agent runs hill-climbing reachability checks:[6] from S to S_i and from G_i to G where i runs the database indices in the order of record similarity. In this example, S_1 is found unreachable by hill-climbing from S and thus the record (S_1, G_1) is discarded. The second record passes hill-climbing checks and the agent is tasked with going to its first subgoal (shown as 1 in the figure).

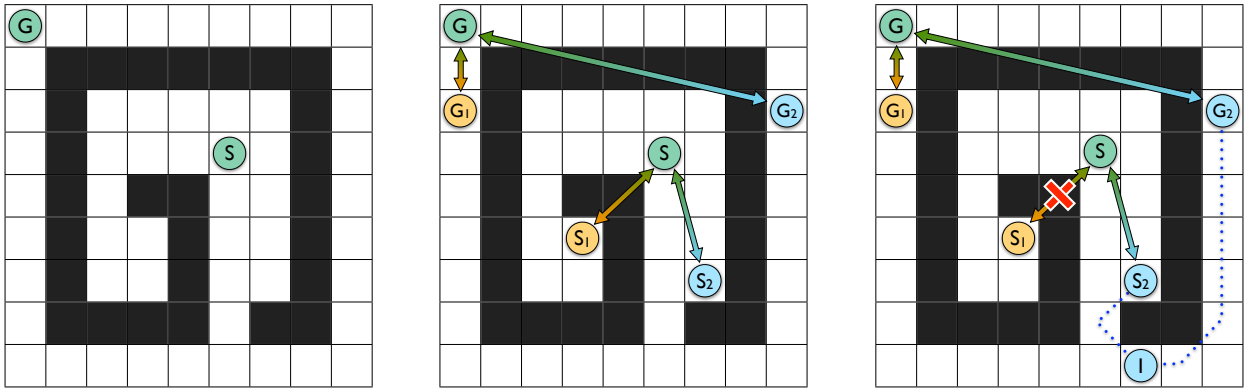


Figure 4: Example of kNN LRTA* on-line operation. **Left:** the agent intends to travel from S to G. **Center:** similarity of (S, G) to (S_1, G_1) and (S_2, G_2) is computed. **Right:** while (S_1, G_1) is more similar to (S, G) than (S_2, G_2), its beginning S_1 is not reachable from S via hill-climbing and hence the record (S_2, G_2) is selected and the agent is tasked with going to subgoal 1.

5.3 kNN LRTA*: Properties

Real-time property. On each move kNN LRTA* invokes LRTA* which expands a constant-bounded

[6] To satisfy the real-time operation constraint, we set an *a priori* constant limit on the number of steps in any hill-climbing check on-line.

Figure 5: The maps used in our empirical evaluation.

number of states. On some moves, kNN LRTA* additionally queries its database to find the appropriate record. Since the database size is independent of the number of states, the query time does not grow with the number of states. The time to sort the records is independent of the total number of states and so are move-limited hill-climbing checks. Therefore, kNN LRTA*'s planning time per move does not grow with the total number of states, satisfying the real-time requirement.

Completeness. Given a problem, the subgoal selection module of kNN LRTA* will either return a database record or instruct LRTA* to go to the global goal. In the latter case, kNN LRTA* is complete because the underlying LRTA* is complete. In the former case, LRTA* is guaranteed to reach the start state of the record due to the way records are picked from the database. LRTA* is then guaranteed to reach the subsequent subgoals due to the completeness of the basic LRTA* and the way the subgoals are constructed.

5.4 kNN LRTA*: Empirical Evaluation

The experiments in this chapter were run on a set of 1000 randomly generated problems across the four maps shown in Figure 5. There were 250 problems on each map and they were constrained to have solution cost of at least 1000. The grid dimensions varied between 4096×4604 and 7261×4096 cells. For each problem we computed an optimal solution cost by running A*. The optimal cost was in the range of $[1003.8, 2999.8]$ with a mean of 1881.76, a median of 1855.2 and a standard deviation of 549.74. We also measured the A* difficulty defined as the ratio of the number of states expanded by A* to the number of edges in the resulting optimal path. For the 1000 problems, the A* difficulty was in the range of $[1, 199.8]$ with a mean of 62.60, a median of 36.47 and a standard deviation of 64.14.

All algorithms compared were implemented in Java using common data structures as much as possible. We used Java version 6 under SUSE Enterprise Linux 10 on a 2.1 GHz AMD Opteron processor with 32 Gbytes of RAM. All timings are reported for single-threaded computations.

We evaluated kNN LRTA* with the following parameters. Database size values were in $\{1000, 5000, 10000, 40000, 60000, 80000\}$ records. On-line, we allowed our hill-climbing test to climb for up to 250 steps before concluding that the destination state is not hill-climbing reachable. This value

was picked after some experimentation and had to be appropriate for the record density on the map. To illustrate, a larger database requires fewer hill-climbing steps to maintain the likelihood of finding a hill-climbing reachable record for a given problem.

We ran reachability checks on the 10 most similar records. LRTA*'s parameter $g_{\max}$ was set to the cost of the most expensive edge (i.e., 1.4) so that LRTA* generated only all immediate neighbors of its current state.

We also contrast kNN LRTA*'s performance to that of TBA*, which was run with the time slices of $\{5, 10, 20, 50, 100, 500, 1000, 2000, 5000\}$ and the cost ratio of expanding a state to backtracing set to 10 (explained in the next section).

5.4.1 Solution Suboptimality and Per-Move Planning Time

We begin the comparisons by looking at average solution suboptimality versus average time per move. Table 1 shows the individual values. kNN LRTA* produces the highest quality solutions, followed by TBA*.

Algorithm	Mean time per move (microseconds)	Solution suboptimality (%)
kNN LRTA*(10000)	7.56	6851.62
kNN LRTA*(40000)	6.88	620.63
kNN LRTA*(60000)	6.40	12.77
kNN LRTA*(80000)	6.55	11.96
TBA*(5)	14.31	1504.54
TBA*(10)	26.34	666.50
TBA*(50)	83.31	131.12
TBA*(100)	117.52	64.66
A*	208.03	0

Table 1: Suboptimality versus time per move.

TBA* cannot reach kNN LRTA* with the database size of 60 and 80 thousand records. Additionally, TBA* is noticeably slower per move as it expands more than one state and allocates some time to backtracing as well. The time per move can be decreased by lowering the value of cutoff but already with the cutoff of 10, TBA* produces unacceptably suboptimal solutions (666.5% suboptimal). As a result, kNN LRTA* dominates TBA* by outperforming it with respect to both measures. This is intuitive as TBA* does not benefit from subgoal precomputation.

For the sake of reference, we also included A* results in the table. A* is not a real-time algorithm and its average time per move tends to increase with the number of states in the map. Additionally, it spends most of it time during the first move when it computes the entire path. Subsequent moves require a trivial computation. In the table, we define A*'s mean time per move as the total planning time for a problem divided by the number of moves in the path A* finds. We average this quantity over all problems. kNN LRTA* is about 30 times faster than A* per move.

5.4.2 Database Precomputation Time

Suboptimality versus database precomputation time is shown in Table 2. Note that while the times are roughly between 10 and 100 hours, they are reported for single-threaded computations. Because database records are independent of each other, the precomputation process scales up linearly with the number of threads. Thus, these times can be decreased by an order of magnitude by simply running the code in parallel on a modern multi-core CPU.

Algorithm	Precomputation time per map (hours)	Solution suboptimality (%)
kNN LRTA*(10000)	13.10	6851.62
kNN LRTA*(40000)	51.89	620.63
kNN LRTA*(60000)	77.30	12.77
kNN LRTA*(80000)	103.09	11.96

Table 2: Suboptimality versus database precomputation time.

5.4.3 Memory Requirements

Memory is at premium in video games, especially on consoles. TBA* space complexity comes from its open and closed list which it builds on-line. kNN LRTA* expands only a single state (the agent's current state) and thus has the closed list of one state and the open list of at most eight states (as any grid cell in our maps has at most eight neighbors). However, it consumes memory as it stores updated heuristic values. Additionally, it stores its subgoal databases. We will first focus on the database size. Then we will cover the total memory consumed on-line: open and closed lists as well as the updated heuristic values.

kNN LRTA* records have two or more states each and the number of records is fixed by the algorithm parameter. Additionally, kNN LRTA* stores start and end states of each record in a kd-tree. We define *relative database size* as the ratio of the total number of states stored in all records to the total number of map grid cells. The empirical results are found in Table 3.

Algorithm	Precomputation time	Records	Relative size	Size (megabytes)
kNN LRTA*(10000)	13.10	10000	0.00308	0.25
kNN LRTA*(40000)	51.89	40000	0.01234	1.00
kNN LRTA*(60000)	77.30	60000	0.01851	1.51
kNN LRTA*(80000)	103.09	80000	0.02468	2.01

Table 3: Database statistics. All values are averages per map. Precomputation time is in hours.

We will first analyze specifically the amount of memory allocated by the algorithms on-line. When an algorithm solves a particular problem, we record the maximum size of its open and closed lists as well as the total number of states whose heuristic values were updated. We count each updated heuristic value as one state in terms of storage required.[7] Adding these three measures together, we record the amount of *strictly on-line memory* per problem. Averaging the strictly on-line memory over all problems, we list the results in Table 4.

TBA*, as time-sliced A*, does not update heuristic values at all. However, its open and closed lists contribute to the highest memory consumption at 1353.94 Kbytes. This is intuitive as TBA* does not use subgoals and therefore must "fill in" potentially large heuristic depressions with its open and closed lists. Also, notice that the total size of these lists does not change with the cutoff as state expansions are independent of agent's moves in TBA*. A* has identical memory consumption as it expands states in the same way as TBA*. Again, kNN LRTA* dominates TBA* for all cutoff values, using less memory and producing better solutions.

Strictly on-line memory gives an insight into the algorithms but does not present a complete picture. Specifically, kNN LRTA* must load its databases into its on-line memory. Thus we define the *cumulative on-line memory* as the strictly on-line memory plus the size of the database loaded. The values are found in Table 5.

[7] Multiple heuristic updates in the same state do not increase the amount of storage.

Algorithm	Strictly on-line memory (Kbytes)	Solution suboptimality (%)
kNN LRTA*(10000)	8.62	6851.62
kNN LRTA*(40000)	5.04	620.63
kNN LRTA*(60000)	4.23	12.77
kNN LRTA*(80000)	4.22	11.96
TBA*(5)	1353.94	1504.54
TBA*(10)	1353.94	666.50
TBA*(50)	1353.94	83.31
TBA*(100)	1353.94	64.66
A*	1353.94	0

Table 4: Strictly on-line memory versus solution suboptimality.

Algorithm	Cumulative on-line memory (Kbytes)	Solution suboptimality (%)
kNN LRTA*(10000)	265.65	6851.62
kNN LRTA*(40000)	1034.08	620.63
kNN LRTA*(60000)	1547.85	12.77
kNN LRTA*(80000)	2062.20	11.96
TBA*(5)	1353.94	1504.54
TBA*(10)	1353.94	666.50
TBA*(50)	1353.94	83.31
TBA*(100)	1353.94	64.66
A*	1353.94	0

Table 5: Solution suboptimality versus cumulative on-line memory.

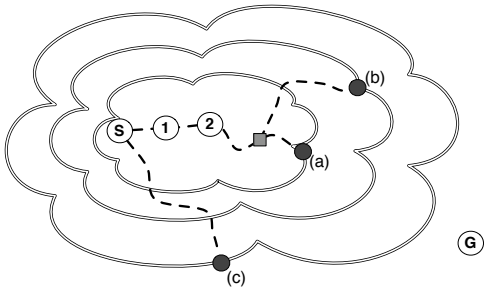


Figure 6: An example of TBA* in action.

TBA* is no longer dominated due to its low memory consumption. The closest comparison is between kNN LRTA* with 60000 records and TBA*. While kNN LRTA* uses 14% more memory than TBA*, it produces solutions of 1.5 to 14.2 times better.

6 Time-Bounded A* (TBA*)

It may not always be feasible to precompute pathfinding information for video game maps. For example, whereas hours of precomputation per map may be acceptable for maps shipping with a game (as the computation is done beforehand at the game studio), the same is unlikely to be the case for user-generated maps. Also, precomputation is much less useful for maps that change frequently during game play (e.g., a bridge or a building is blown-up or a new one built).

Unfortunately, in the absence of precomputed information for guiding the search, LRTA*-like algorithms tend to preform poorly, often revisiting and re-expanding the same states over and over again. In contrast, A* with a consistent heuristic never re-expands a state. However, in A* 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 and, thus, do not scale well. One way of alleviating this problem has been to use A* with hierarchies of state-space abstractions, and search first for an approximate path in a highly abstracted state space and then refine it locally in a less abstract one. While faster, their planning time per move still increases with the number of states, making them non-real-time.

Below we describe a time-bounded version of the A* algorithm, called Time-Bounded A* (TBA*) [2], that achieves true real-time behavior while requiring neither precomputation nor state space abstractions.

6.1 TBA*: Search

The TBA* algorithm expands states in an A* fashion, away from the original start state, towards the goal until the goal state is expanded. However, unlike A* that plans a complete path before committing to the first action, TBA* interrupts its search periodically after a fixed number of state expansions and acts. If the complete path to the goal has not yet been found, the agent instead moves towards the most promising state on the open list. This interleaving of planning and acting operations ensures real-time behavior. A key aspect of TBA* over LRTA*-based algorithms is that it retains closed and open lists over its planning steps. Thus, on each planning step it does not start planning from scratch, but continues with its open and closed lists from the previous planning step.

The basic idea behind TBA* is depicted in Figure 6. 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.

The pseudo-code of TBA* is shown as Algorithm 1. The arguments to the algorithm are the *start* and *goal* states, the search problem *P*, and the per-move search limit *R* (expressed as the number of states to expand on each step). 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 divides up the resource limit as it must be shared between state expansion and backtracing[8] operations (lines 5-6). The constants $r \in [0,1]$ and c stand for the fraction of the resource limit to use for state expansions and the relative cost of a expansion compared to backtracing (e.g., a value of 10 indicates that one state expansion takes ten times more time to execute than a backtracing step), respectively. The algorithm then enters the main loop where it repeatedly interleaves *planning* (lines 8-23) and *execution* (lines 24-35) 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 9-11). 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 16) 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 13-15). Only when the path has been fully traced back, is it set to become the new path for the agent to follow (line 18); 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 26).[9] 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 29). 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 31).

6.2 TBA*: 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, a careful choice of data-structures still enables (amortized) constant-time operation.[10]

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

[8] We use the term *backtracing* for the act of tracing a path backwards in the planning phase. The term *backtracking* is used in its usual sense — physically moving (backwards) along the path in the execution phase.

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

[10] Using the standard heap-based implementation of the open list gives times per move sub-polynomial (logarithmic) in the number of states and, therefore, violates the real-time constraint.

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

```
1: loc ← start
2: solutionFound ← false
3: solutionFoundAndTraced ← false
4: doneTrace ← true
5: N_E = ⌊R × r⌋
6: N_T = (R − N_E) × c
7: while loc ≠ goal do
8:    { PLANNING PHASE }
9:    if not solutionFound then
10:      solutionFound ← A*(lists, start, goal, P, N_E)
11:   end if
12:   if not solutionFoundAndTraced then
13:      if doneTrace then
14:         pathNew ← lists.mostPromisingState()
15:      end if
16:      doneTrace ← traceBack(pathNew, loc, N_T)
17:      if doneTrace then
18:         pathFollow ← pathNew
19:         if pathFollow.back() = goal then
20:            solutionFoundAndTraced ← true
21:         end if
22:      end if
23:   end if
24:   { EXECUTION PHASE }
25:   if pathFollow.contains(loc) then
26:      loc ← pathFollow.popFront()
27:   else
28:      if loc ≠ start then
29:         loc ← lists.stepBack(loc)
30:      else
31:         loc ← loc_last
32:      end if
33:   end if
34:   loc_last ← loc
35:   move agent to loc
36: end while
```

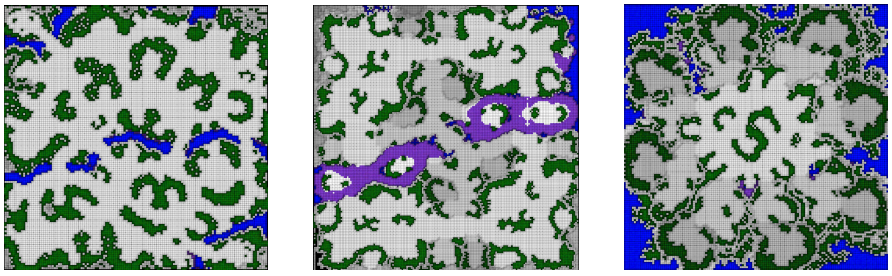


Figure 7: The maps used in the TBA* experiments.

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. One advantage TBA* has over precomputation-based algorithms, is that no memory is allocated for the precomputed data.

6.3 TBA*: Empirical Evaluation

The experiments performed in this section were run using three different maps modeled after game worlds from a popular real-time strategy game (shown in Figure 7). The maps were scaled up to 512×512 cells to increase the problem difficulty [39, 12]. 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 pathfinding problems (3 maps $\times$ 100 searches on each).

In the experiments that follow, TBA* was matched against two recent real-time search algorithms that have been shown particularly effective in pathfinding on video-game maps. They both use state abstraction and precomputation to improve performance. The algorithms and their parameter settings are:

- **PR LRTA*** is Path Refinement Learning Real-Time Search [13]. The algorithm runs LRTA* with a fixed search depth d in an abstract space (abstraction level ℓ in a clique abstraction hierarchy [39]) 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 [12]. 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 states per action).
- **TBA*** is our Time-Bounded TBA*; the resource limit R took on the values $\{10,25,50,75,100,500,1000\}$ but the values of r and c were fixed at 0.9 and 10, respectively.

A previous section of the chapter contrasted the performance of TBA* and kNN LRTA* (thus not included here).

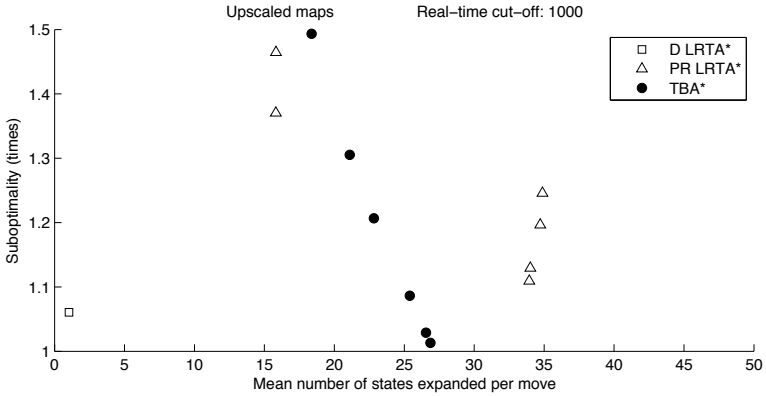


Figure 8: TBA* compared to other real-time algorithms.

Figure 8 presents the results. 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 then the particular parameter setting was excluded from consideration. Also, 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 the aforementioned PR LRTA* and D LRTA* parameter combinations.

We see that TBA* performs on par with these 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 pathfinding 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 versus suboptimality tradeoff than TBA*, is at the expense of extensive precomputation that can take hours for even a single map.

7 Real-time Iterative-deepening Best-first Search (RIBS)

TBA* is a relatively straightforward extension of A* which allows immediate movement by an agent before A* finds a complete path. However, TBA* is not an agent-centric algorithm. That is, the memory accesses performed by TBA* happen at arbitrary places in the map that may not be local to the agent. This can be important if there are cache or memory concerns, where random memory accesses are slow, or if the world is dynamic and might change significantly after planning is completed. One way to look at RIBS is that it is an agent-centric version of TBA*, however there are a few extra pieces that are needed to make RIBS efficient in practice. If an agent-centric approach is not important, TBA* may be a better option.

The basic approach for RIBS is shown in simplified pseudo-code in Figure 9. A global cost limit is used as the current estimate of the cost to the goal. An agent begins at the state $s_{current}$ and is passed the last state visited, which is used to set up parent pointers so the agent can retrace its path if stuck in a dead end (lines 1-4).

Next, the agent computes the f-cost of the successor states, and recursively visits any states which have f-cost less than or equal to the current bound. If all successors are visited without finding the goal,

```
Global: cost_limit ← 0
RIBS(s_current, g-cost, s_parent, s_goal)
 1   if s_current is visited the first time
 2     set parent of s_current to s_parent
 3     store g-cost of s_current
 4   end if
 5   while s_current is not s_goal
 6     mark s_current as visited with current cost_limit
 7     foreach successor succ_i with f-cost ≤ cost_limit and unvisited with current cost_limit
 8       if succ_i never expanded or succ_i's parent is s_current
 9         RIBS(s_current, g-cost + cost(s_current, s_succ_i), s_succ_i, s_goal)
10       end if
11     end foreach
12     if s_parent is not nil { only occurs at s_start }
13       return
14     else
15       increase cost_limit
16     end if
17   end while
```

Figure 9: A simplified version of the RIBS algorithm.

then the agent returns to its parent state. If there is no parent state, then the agent must be in the start state, and there is no path to the goal with the current bound. In this case the bound is increased and the procedure starts over.

This procedure is essentially the same as IDA*, and so it can be proven that, with a consistent heuristic, when an agent expands a state for the first time it will have discovered an optimal cost path to that state. As a corollary, RIBS is guaranteed to identify an optimal path to the goal state by the time it reaches it. Note that it does not mean that it will have followed such an optimal path. Like other real-time heuristic search agents, a RIBS agent tends to follow suboptimal paths in practice.

A simple agent running RIBS would take one action per move. An agent moves forward on line 9 and moves backwards on line 13. The while statement on line 5 really only serves to keep the agent iterating with increasing cost limits at the start state, as for every other state a parent will be defined causing the while loop to exit at line 13.

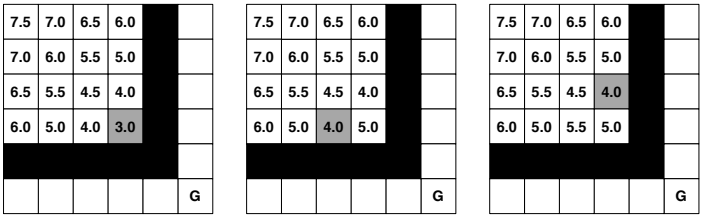
This description of the algorithm is fairly simple to implement, but it is missing a few details, such as some of the code for initializing new states, the procedure for updating the cost limit, and some important pruning details. The first two details are relatively straightforward, so we will only discuss the pruning details here. Also note that RIBS is shown as a recursive algorithm that would run until completion, but it is not hard to break this computation into pieces that could be resumed when the time limit for the current action expires.

7.1 RIBS: Intuition

Learning h-values can be slow because inaccurate heuristics values are used to update other (also inaccurate) heuristic values. This is particularly problematic if a learning agent enters a heuristic depression [20], a localized area in the search space where it has to repeatedly revisit states to raise their heuristic values enough to be able to continue to explore other parts of the search space. The more frequent and deeper the depressions are, the more severely the problem manifests itself.

This is illustrated in Figure 10 with an example of LRTA* behavior on a portion of a map with a local

minima in the corner. To simplify the example, diagonal moves have cost 1.5. States are marked with their initial heuristic values. Consider part (a) where the agent is in the shaded state. Using a lookahead of one, the value of the corner heuristic can be updated from 3 to 5, because a neighbor distance 1 has heuristic cost 4. In part (b) the agent moves to the highlighted state and makes a similar update, raising the h-cost to 5.5, before moving to the state updated in part (c), where that state will be updated to have a heuristic value of 5.5.



(a) Step 1 (b) Step 2 (c) Step 3

Figure 10: Learning in a local minima.

After three updates, considerable learning still remains. This is because the heuristic is being updated locally from neighboring heuristics, which, due to consistency, cannot be considerably larger. Thus, a state must be visited and updated many times before large changes in the heuristic can occur. As this learning begins far from the goal state, heuristic estimates are likely to be inaccurate. For the same reason that heuristic costs (h-costs) tend to be more accurate closer to the goal state, g-costs tend to be more accurate closer to the start state. RIBS takes advantage of this fact and learns g-costs for all states visited by the agent. Since the agent begins in the start state, such a learning process is more efficient than the h-cost learning of LRTA*: more accurate g costs are learned faster.

The second key observation is that (accurate) g-costs can be helpful in escaping heuristic depressions. Not only can they greatly reduce the number of times the agent must revisit states in a heuristic depression, but they are also useful in identifying dead states and redundant paths. Such identifications require accurate costs and, as a result, work much better with g-cost learning than with h-cost learning.

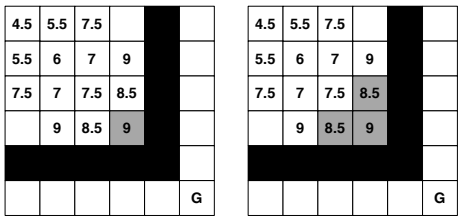
Excluding the start and goal, any state on an optimal path must have neighbors with both higher and lower g-costs. If a state has no neighbors with larger g-costs, then an optimal path to the goal cannot pass through this state. We thus define a *dead state* as follows: Given a start state s and a state n, n is a dead state if n is not the goal state and if for all non-dead neighbors of n, $n_1 \ldots n_i$, $\text{cost}(s,n) \geq \text{cost}(n_i,s)$.

Consider the example in Figure 11, which shows g-cost estimates for the same problem. Upon reaching the corner, the agent can potentially mark each state with the g-costs in the figure, which are upper-bounds on the actual cost to each state. In part (a) of the figure, the agent can see that the highlighted state in the corner is dead, because all neighbors can be reached by shorter paths through other states. After this state is marked dead, in part (b), two more states can be marked dead. Learning that a state is dead only requires visiting a state a single time, unlike learning a heuristic, which may take multiple visits. Fortunately, there is more that can be done if we know the optimal cost to each state.

Consider Figure 12(a). In this case the states in the corners can be marked as dead and ignored once the optimal cost is discovered. Note, however, that even after removing the dead states there are still many paths that lead out through this room, shown in Figure 12(b). However, because there is only a single doorway to the room, these paths are all redundant. Detecting and ignoring states on such redundant paths offers additional saving. This requires two steps. In Figure 12(c) we show two possible optimal paths leading out of the room. We focus on states A and B, shown in detail In Figure 12(d).

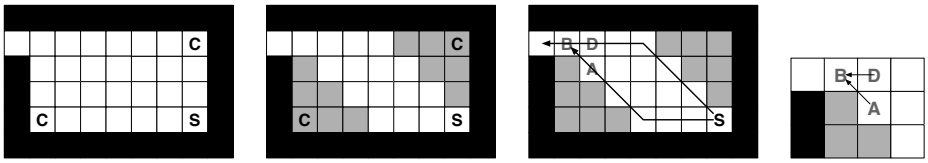
The first step is to mark all parents which are along optimal paths to a state. Each time a state is generated, if the parent is on an optimal path to the state, the parent is added to the list of optimal parents for that state. In the case of state B, states A and D can both reach B with the same cost, and so B maintains this information.

The second step occurs in the next iteration of search. Suppose that A is visited first. Then, at A



(a) Step 1 (b) Step 2

Figure 11: Learning g-costs.



(a) (b) (c) (d)

Figure 12: Marking dead states.

we will notice that there is also an optimal path to B through D. Since there are no other optimal paths through A to a successor of A, A can be marked dead or redundant, and B is marked to have a single optimal parent of D. If D were visited first, D would be marked redundant, and only the path through A would be maintained.

In Figure 13 we show an example of redundant state removal on a fragment of an actual game map. The different types of states are labelled with arrows. Most states have been marked as dead or redundant, meaning that additional exploration focuses just on the successors of a small fringe of previously expanded states.

The effectiveness of dead state and redundant state pruning will depend on the problem being solved. We observe that if previously expanded states can be marked dead and/or redundant at the same rate that new states are expanded, then the performance of RIBS would approach that of TBA*, as TBA* never revisits expanded states. But, RIBS must first mark states as dead and/or redundant in order to stop visiting them, and searches in multiple iterations, so it cannot completely match the performance of TBA*, especially given that it obeys agent-centric constraints.

7.2 RIBS: Properties

Real-time property. The number of state expansions performed for each step of RIBS can be set to any desired constant. Note that RIBS maintains no open or closed lists and thus does not require sophisticated data representations to satisfy the real-time constraint.

Completeness RIBS, being at its core a time-sliced version of IDA*, is complete under the same assumptions as IDA*. Also note that, similar to TBA* and unlike kNN LRTA*, when RIBS finds the goal it will have determined the optimal solution path, although it will not have followed that path en route to the goal.

Memory complexity. RIBS has the same worst case as TBA* where it will consider and store information all states in the state space. Unlike kNN LRTA*, however, it does not require loading or precomputing a database.

7.3 RIBS: Empirical Evaluation in Heuristic Depressions

Of all the algorithms discussed in this chapter, RIBS makes the most restrictive assumptions about what an agent running RIBS is able to perform in the environment. RIBS assumes no up-front knowledge of

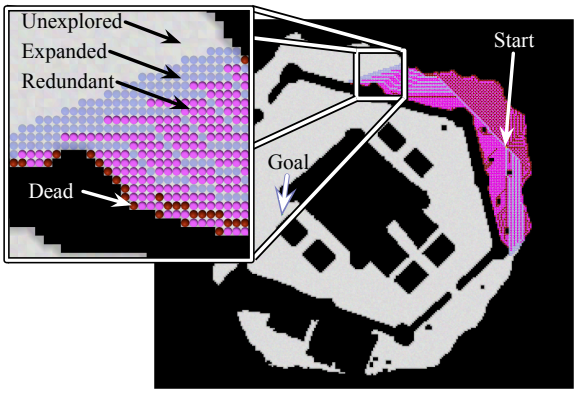


Figure 13: An example map showing the search in-progress with the different types of states.

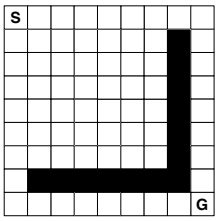


Figure 14: The example map used to compare RIBS and LRTA* performance.

the domain, ruling out any opportunity for precomputation. RIBS also assumes that random access to states far from the agent is not available, ruling out the TBA* approach. TBA* and kNN LRTA* have better performance than RIBS in practice because they do not make such assumptions. As a result, we focus on evaluating RIBS' ability to quickly escape heuristic depressions. We focus on the comparison between g-learning RIBS and h-learning basic LRTA*. This showcases RIBS ability to use its accurately learned g-costs to identify the redundant and dead states as described above.

The basic version of LRTA* which we compare against can be described as follows: The *local search space* only includes the neighbors of the current state. The *local learning space* is only the current state. The *learning rule* is mini-min, and the *control strategy* is to move to the best neighboring state. Although the local search space and learning space are small, increasing their size does not significantly change the results we present here (see [40] for more details).

We experiment on the map in Figure 14, where the agent starts in the upper left corner and must travel to the lower-right corner. The default heuristic leads directly into the corner, from which the agent must then escape. This structure is common in many maps, and so we experiment directly with this example, scaling the size to measure performance.

The results of the comparison are in Figure 15. The x-axis is the number of states in the whole map, while the y-axis is the number of states expanded by each algorithm. Note that both axes use a logarithmic scale. The RIBS line approximates $y = 10x$ as the map gets larger, while the LRTA* approximates $y = 0.14x^{1.5}$. This means that, once the map gets large, RIBS can expand each state 10 times before finding the optimal solution. For LRTA*, however, the number of expansions is polynomial in the size of the map. This explains the "scrubbing" behavior of LRTA* – the number of expansions that it takes for LRTA* to escape a local minima can be far more than the number of states in the local minima. The performance of RIBS and LRTA* crosses when the number of states in the local minima grows to approximately 1500 states, which corresponds to a 40×40 room in a larger map, a size that is not unrealistic.

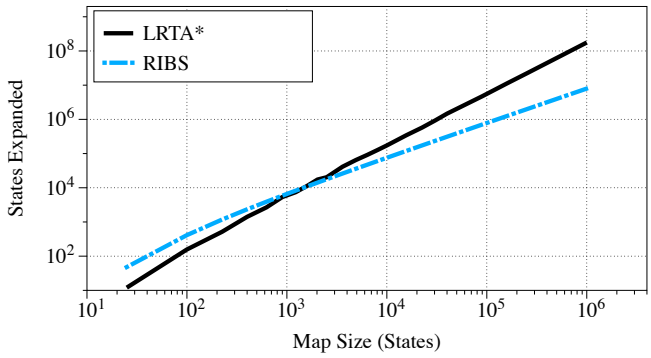


Figure 15: RIBS versus LRTA* performance on the example map.

8 Future Work

We presented and evaluated kNN LRTA*, TBA* and RIBS for grid-based pathfinding. Formally, the algorithms are applicable to arbitrary weighted graphs that satisfy the constraints at the beginning of Section 2. Thus, in principle, they should be applicable to general planning using the ideas from search-based planners ASP [5], the HSP-family [3], FF [18], SHERPA [22] and LDFS [4]. An actual application is a subject of future work.

The methods can also be further improved and fine tuned in our problem domain of pathfinding in video games. Currently one of the main drawbacks of kNN LRTA* is the long precomputation time needed for generating the off-line databases. While the time is affordable on the game company side, most players would want their home-made game maps to be processed in a matter of seconds or minutes. While the computation can be sped up at a linear scale by using multi-core processors this would still come up short. One of the main focus of future work on kNN LRTA* will thus be to shorten the precompuation time, for example, we might be able to get away with much smaller databases if the database records were generated in a manner such that they produce a better coverage of the state space.

Unlike A*, no real-time algorithm can guarantee finding an optimal path. This is of a little consequence in video-game pathfinding as long as approximately optimal paths are found. More importantly is that the agents navigate the game world in a rational way, for example that they do not show visually jarring or indecisive behavior by frequently changing their mind as of where to go. Whereas both kNN LRTA* and TBA* are much improved in that respect compared to most other mainstream real-time algorithms, such behavior does still occasionally surface, especially in the latter. Even a single incident of an irrational pathfinding behavior can break the player's immersion. Some preliminary solutions for TBA* are provided in [2] but more effective solutions are required.

RIBS is a promising approach but, given its recency, further empirical evaluation as well as algorithmic improvements are necessary. In particular, variants of RIBS that forgo the eventual identification of optimal paths and, as a result, find better suboptimal solutions, can be explored.

9 Conclusions

In this chapter we considered the problem of real-time heuristic search whose planning time per move does not depend on the number of states. We reviewed three modern algorithms, each with its strengths and weaknesses.

In terms of solution sub-optimality when given equal computing resources (or vice versa, required computing resources for finding equally good solutions), kNN LRTA* shows the best performance. Because pathfinding tends to be a rather computing intensive task in modern games, especially in large game worlds with multiple agents navigating simultaneously, this metric is of an utmost importance. This level of on-line performance comes at the cost of long offline precomputation times (hours per

map). TBA*, although not being quite as effective as kNN LRTA*, still shows a good performance and has the benefit of not requiring any precomputation. It may thus be better poised for environments that dynamically change during game play. TBA* also uses on average somewhat less memory that kNN LRTA*, which can be an important consideration on some gaming platforms (e.g., consoles). Both algorithms thus appear well poised for video game pathfinding.

RIBS is an interesting way of moving TBA* closer to being an agent-centered approach – an important consideration for some problem domains. The algorithm also provides added insights into how real-time search agents can learn heuristics.

Acknowledgements

Parts of this chapter have been previously published as conference and journal articles written by the authors.

This research was supported by grants from the National Science and Engineering Research Council of Canada (NSERC) and the Icelandic Centre for Research (RANNÍS). All research by Nathan Sturtevant included in this chapter was performed at the University of Alberta. We appreciate the help of Josh Sterling, Stephen Hladky and Daniel Huntley.

References

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

[2] Björnsson, Y., Bulitko, V., Sturtevant, N.: TBA*: Time-bounded A*. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 431 – 436. AAAI Press, Pasadena, California (2009)

[3] Bonet, B., Geffner, H.: Planning as heuristic search. Artificial Intelligence **129**(1–2), 5–33 (2001)

[4] Bonet, B., Geffner, H.: Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), pp. 142–151. Cumbria, UK (2006)

[5] Bonet, B., Loerincs, G., Geffner, H.: A fast and robust action selection mechanism for planning. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 714–719. AAAI Press / MIT Press, Providence, Rhode Island (1997)

[6] Bulitko, V.: Lookahead pathologies and meta-level control in real-time heuristic search. In: Proceedings of the 15th Euromicro Conference on Real-Time Systems, pp. 13–16 (2003)

[7] Bulitko, V.: Learning for adaptive real-time search. Tech. Rep. http://arxiv.org/abs/cs.AI/0407016, Computer Science Research Repository (CoRR) (2004)

[8] Bulitko, V., Björnsson, Y.: kNN LRTA*: Simple subgoaling for real-time search. In: Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE), pp. 2–7. AAAI Press, Stanford, California (2009)

[9] Bulitko, V., Björnsson, Y., Lawrence, R.: Case-based subgoaling in real-time heuristic search for video game pathfinding. Journal of Artificial Intelligence Research (JAIR) (2010). (in press).

[10] Bulitko, V., Lee, G.: Learning in real time search: A unifying framework. Journal of Artificial Intelligence Research (JAIR) **25**, 119–157 (2006)

[11] Bulitko, V., Li, L., Greiner, R., Levner, I.: Lookahead pathologies for single agent search. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1531–1533. Acapulco, Mexico (2003)

[12] Bulitko, V., Luštrek, M., Schaeffer, J., Björnsson, Y., Sigmundarson, S.: Dynamic control in real-time heuristic search. Journal of Artificial Intelligence Research (JAIR) **32**, 419 – 452 (2008)

[13] Bulitko, V., Sturtevant, N., Lu, J., Yau, T.: Graph abstraction in real-time heuristic search. Journal of Artificial Intelligence Research (JAIR) **30**, 51–100 (2007)

[14] Furcy, D., Koenig, S.: Speeding up the convergence of real-time search. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 891–897 (2000)

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

[16] Hernández, C., Meseguer, P.: Improving convergence of LRTA*(k). In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Workshop on Planning and Learning in A Priori Unknown or Dynamic Domains, pp. 69–75. Edinburgh, UK (2005)

[17] Hernández, C., Meseguer, P.: LRTA*(k). In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1238–1243. Edinburgh, UK (2005)

[18] Hoffmann, J.: A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In: Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems (ISMIS), pp. 216–227 (2000)

[19] Ishida, T.: Moving target search with intelligence. In: National Conference on Artificial Intelligence (AAAI), pp. 525–532 (1992)

[20] Ishida, T.: Moving target search with intelligence. In: Proceedings of the National Conference on Artificial Intelligence, pp. 525–532 (1992)

[21] Koenig, S.: A comparison of fast search methods for real-time situated agents. In: Proceedings of Int. Joint Conf. on Autonomous Agents and Multiagent Systems, pp. 864 – 871 (2004)

[22] Koenig, S., Furcy, D., Bauer, C.: Heuristic search-based replanning. In: Proceedings of the Int. Conference on Artificial Intelligence Planning and Scheduling, pp. 294–301 (2002)

[23] Koenig, S., Likhachev, M.: Real-time adaptive A*. In: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 281–288 (2006)

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

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

[26] Likhachev, M., Ferguson, D.I., Gordon, G.J., Stentz, A., Thrun, S.: Anytime dynamic A*: An anytime, replanning algorithm. In: ICAPS, pp. 262–271 (2005)

[27] Likhachev, M., Gordon, G.J., Thrun, S.: ARA*: Anytime A* with provable bounds on sub-optimality. In: S. Thrun, L. Saul, B. Schölkopf (eds.) Advances in Neural Information Processing Systems 16. MIT Press, Cambridge, MA (2004)

[28] Luštrek, M.: Pathology in single-agent search. In: Proceedings of Information Society Conference, pp. 345–348. Ljubljana, Slovenia (2005)

[29] Luštrek, M., Bulitko, V.: Lookahead pathology in real-time path-finding. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search, pp. 108–114. Boston, Massachusetts (2006)

[30] Pearl, J.: Heuristics. Addison-Wesley (1984)

[31] Rayner, D.C., Davison, K., Bulitko, V., Anderson, K., Lu, J.: Real-time heuristic search with a priority queue. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 2372–2377. Hyderabad, India (2007)

[32] Russell, S., Wefald, E.: Do the right thing: Studies in limited rationality. MIT Press (1991)

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

[34] Shue, L.Y., Li, S.T., Zamani, R.: An intelligent heuristic algorithm for project scheduling problems. In: Proceedings of the 32nd Annual Meeting of the Decision Sciences Institute. San Francisco (2001)

[35] Shue, L.Y., Zamani, R.: An admissible heuristic search algorithm. In: Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS-93), *LNAI*, vol. 689, pp. 69–75 (1993)

[36] Sigmundarson, S., Björnsson, Y.: Value Back-Propagation vs. Backtracking in Real-Time Search. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search, pp. 136–141. Boston, Massachusetts, USA (2006)

[37] Stenz, A.: The focussed D* algorithm for real-time replanning. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1652–1659 (1995)

[38] Sturtevant, N.: Memory-efficient abstractions for pathfinding. In: Proceedings of the third conference on Artificial Intelligence and Interactive Digital Entertainment, pp. 31–36. Stanford, California (2007)

[39] Sturtevant, N., Buro, M.: Partial pathfinding using map abstraction and refinement. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 1392–1397. Pittsburgh, Pennsylvania (2005)

[40] Sturtevant, N.R., Bulitko, V., Björnsson, Y.: On learning in agent-centered search. In: Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS) (2010)