Journal of Artificial Intelligence Research 50 (2014) 235-264 Submitted 12/13; published 06/14

Reconnection with the Ideal Tree: A New Approach to Real-Time Search

Nicolás Rivera NICOLAS.RIVERA@KCL.AC.UK
King's College London
London, WC2R 2LS, UK

León Illanes LILLANES@UC.CL
Jorge A. Baier JABAIER@ING.PUC.CL
Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Vicuña Mackenna 4860, Santiago, Chile

Carlos Hernández CHERNAN@UCSC.CL
Departamento de Ingeniería Informática
Universidad Católica de la Santísima Concepción
Caupolicán 491, Concepción, Chile

Abstract

Many applications, ranging from video games to dynamic robotics, require solving single-agent, deterministic search problems in partially known environments under very tight time constraints. Real-Time Heuristic Search (RTHS) algorithms are specifically designed for those applications. As a subroutine, most of them invoke a standard, but bounded, search algorithm that searches for the goal. In this paper we present FRIT, a simple approach for single-agent deterministic search problems under tight constraints and partially known environments that unlike traditional RTHS does not search for the goal but rather searches for a path that connects the current state with a so-called *ideal tree* $\mathcal{T}$. When the agent observes that an arc in the tree cannot be traversed in the actual environment, it removes such an arc from $\mathcal{T}$ and then carries out a reconnection search whose objective is to find a path between the current state and any node in $\mathcal{T}$. The reconnection search is done using an algorithm that is passed as a parameter to FRIT. If such a parameter is an RTHS algorithm, then the resulting algorithm can be an RTHS algorithm. We show, in addition, that FRIT may be fed with a (bounded) complete blind-search algorithm. We evaluate our approach over grid pathfinding benchmarks including game maps and mazes. Our results show that FRIT, used with RTAA*, a standard RTHS algorithm, outperforms RTAA* significantly; by one order of magnitude under tight time constraints. In addition, FRIT(daRTAA*) substantially outperforms daRTAA*, a state-of-the-art RTHS algorithm, usually obtaining solutions 50% cheaper on average when performing the same search effort. Finally, FRIT(BFS), i.e., FRIT using breadth-first-search, obtains best-quality solutions when time is limited compared to Adaptive A* and Repeated A*. Finally we show that Bug2, a pathfinding-specific navigation algorithm, outperforms FRIT(BFS) when planning time is extremely limited, but when given more time, the situation reverses.

1. Introduction

Real-Time Heuristic Search (Korf, 1990) is an approach to solving single-agent search problems when a limit is imposed on the amount of computation that can be used for deliberation. It is used for solving problems in which agents have to start moving before a complete search algorithm can solve the problem and is especially suitable for problems in which the environment is only partially known in advance.

An application of real-time heuristic search algorithms is goal-directed navigation in video games (Bulitko, Björnsson, Sturtevant, & Lawrence, 2011) in which computer characters are expected to find their way in partially known terrain. Game-developing companies impose a constant time limit on the amount of computation per move close to one millisecond for all simultaneously moving characters (Bulitko et al., 2011). As such, real-time search algorithms are applicable since they provide the main loop with quick moves that allow implementing continuous character moves.

Most standard real-time heuristic search algorithms—e.g., LRTA* (Korf, 1990) or LSS-LRTA* (Koenig & Sun, 2009)—are not algorithms of choice for videogame developers, since they will require characters to re-visit many states in order to escape so-called heuristic depressions, producing back-and-forth movements, also referred to as *scrubbing* (Bulitko et al., 2011). The underlying reason for this behavior is that the heuristic used to guide search must be updated—in a process usually referred to as *heuristic learning*—whenever new obstacles are found. To exit so-called heuristic depressions, the agent may need to revisit a group of states many times (Ishida, 1992).

By exploiting preprocessing (e.g., Bulitko, Björnsson, Lustrek, Schaeffer, & Sigmundarson, 2007; Bulitko, Björnsson, & Lawrence, 2010; Hernández & Baier, 2011), one can produce Real-Time Heuristic Search algorithms that will control the agent in a way that is sensible to the human observer. Give a map of the terrain, these algorithms compute information offline that can later be utilized online by a Real-Time Search algorithm to find paths very quickly.

Unfortunately, preprocessing is not applicable in all settings. For example if one wants to implement an agent which has *no knowledge* of the terrain, there is no map that is available prior to search and hence no preprocessing can be carried out. On the other hand, when knowledge about the terrain is only partial (i.e., the agent may know the location of some of the obstacles but not all of them), using a plain Real-Time Heuristic Search along with partial information about the map obtained from preprocessing (i.e., a perfect heuristic computed for the partially known map) may still result in the same performance issues described above.

In this paper we present FRIT, a real-time search algorithm that does not necessarily rely on heuristic learning to control the agent, and that produces high-quality solutions in partially known environments. While easily motivated by game applications, our algorithm is designed for general search problems. An agent controlled by our algorithm always follows the branch of a tree containing a family of solutions. We call such a tree the *ideal tree* because the paths it contains are solutions in the world that is currently known to the agent, but such solutions may not be legal in the actual world. As the agent moves through the states in the ideal tree it will usually encounter states that are not accessible and which block a solution in the ideal tree. When this happens, a secondary algorithm—which is given as a parameter—is used to perform a search and reconnect the current state with another state known to be in the ideal tree. After reconnection succeeds the agent is again on a state of an updated ideal tree, and it can continue following a branch of the tree.

We discuss two different ways in which the algorithm given as a parameter to FRIT can be useful for real-time scenarios. The first alternative is to feed FRIT with a real-time search algorithm. This produces a standard real-time search algorithm, but not a so-called *agent-centered* search algorithm (Koenig, 2001) since to verify that a node is connected to the ideal tree it may need to consider states that are far away from its current position.

The second option to make FRIT amenable for real-time scenarios consists of feeding FRIT with a bounded complete search algorithm; i.e., a complete search algorithm which in each iteration expands a bounded number of states. After performing such a limited number of expansions, the search algorithm may have not found a reconnecting path. In this case the agent may or may not perform an action depending on domain-specific considerations. In our implementation we chose to perform no move at all.

We evaluated our algorithm over standard game and maze pathfinding benchmarks using both a blind, breadth-first search algorithm and two different real-time search algorithms for reconnection. Even though our algorithm does not guarantee optimality, solutions returned, in terms of quality and total time, are significantly better than those returned by the state-of-the-art real-time heuristic search algorithms we compared to, when the search effort is comparable. Upon inspection of the route followed by the agent, we observe that when using blind-search algorithms for reconnection they do not contain back-and-forth, "irrational" movements, and that indeed they look similar to solutions returned by so-called bug algorithms (LaValle, 2006; Taylor & LaValle, 2009) developed by the robotics community. As such, it usually detects states that do not need to be visited again—sometimes referred to as dead-ends or redundant states (Sturtevant & Bulitko, 2011; Sharon, Sturtevant, & Felner, 2013)—without implementing a specific mechanism to detect them.

We also compared our algorithm to incremental heuristic search algorithms that can be modified to behave like a real-time search algorithm. We find that, although FRIT does not reach the same solution quality, it can obtain solutions that are significantly better when the time deadline is tight (under 40μ sec).

Our algorithm is extremely easy to implement and, in case there is sufficient time for pre-processing, can utilize techniques already described in the literature, like so-called *compressed path databases* (Botea, 2011), to compute an initial ideal tree. Furthermore, we provide proofs for termination of the algorithm using real-time search and blind-search for reconnection, and provide a bound on the number of moves required to find a solution in arbitrary graphs.

Some of the contributions presented in this article have been published in a conference paper (Rivera, Illanes, Baier, & Hernandez, 2013b). This articles extends the work and includes new material. In particular:

- We describe a method to use our algorithm with a real-time search algorithm passed as a parameter, and evaluate the results obtained when using two different real-time algorithms.
- We provide proofs for the termination of algorithms obtained by using the aforementioned method, and a general proof for convergence applicable to all the algorithms we propose.
- We incorporate a small optimization that affects the InTree[c] function described in Section 3.

- We extend the previous empirical results by including maze benchmarks, which had not been previously considered, and by evaluating on more problem instances. In addition, we compare our algorithm with the Bug2 (Lumelsky & Stepanov, 1987) pathfinding algorithm.

The rest of the paper is organized as follows. In Section 2 we describe the background necessary for the rest of the paper. In Section 3 we describe a simple version of our algorithm that is not real-time. In Section 4 we describe two alternative ways to make the algorithm satisfy the real-time property. In Section 5 we present a theoretical analysis, followed by a description of our experimental evaluation in Section 6. We then describe other related work, and finish with a summary.

2. Background

The search problems we deal with in this paper can be described by a tuple $P = (G, c, s_{start}, g)$, where $G = (S, A)$ is a directed graph that represents the search space. The set S represents the *states* and the arcs in A represent all available actions. State $s_{start} \in S$ is the *initial state* and state $g \in S$ is the *goal state*. We assume that S is finite, that A does not contain elements of the form (s, s), that G is such that g is reachable from all states reachable from s_{start}. In addition, we have a non-negative cost function $c : A \to \mathbb{R}$ which associates a cost with each of the available actions. Naturally, the cost of a path in the graph is the sum of the costs of the arcs in the path. Finally $g \in S$ is the goal state. Note that even though our definition considers a single goal state it can still model problems with multiple goal states since we can always transform a multiple-goal problem into a single-goal problem by adding a new state g to the graph and connecting the goals in the original problem to g with a zero-cost action.

We define the distance function $d_G : S \times S \to \mathbb{R}$ such that $d_G(s, t)$ denotes the cost of a shortest path from s to t in the graph G. A heuristic for a search graph G is a non-negative function $h : S \to \mathbb{R}$ such that $h(s)$ estimates $d_G(s, g)$. We say that h is *admissible* if $h(s) \leq d_G(s, g)$, for all $s \in S$. In addition, we say a heuristic h is *consistent* if for every pair $(s, t) \in A$ it holds that $h(s) \leq c(s, t) + h(t)$, and furthermore that $h(g) = 0$. It is simple to prove that consistency implies admissibility.

2.1 Real-Time Search

Given a search problem $P = (G, c, s_{start}, g)$, the objective of a real-time search algorithm is to move an agent from s_{start} to g, through a low-cost path. The algorithm should satisfy the *real-time property*, which means that the agent is given a bounded amount of time for deliberating, independent of the size of the problem. After deliberation, the agent is expected to move. After such a move, more time is given for deliberation and the loop repeats.

Most Real-Time Heuristic Search algorithms rely on the execution of a bounded but standard state-space search algorithm (e.g., A*, Hart, Nilsson, & Raphael, 1968). In order to apply such an algorithm in partially known environments, they carry out their search in a graph which may not correspond to the graph describing the actual environment. In particular, in pathfinding in grid worlds, it is assumed that the dimensions of the grid are known, and to enable search a *free-space assumption* (Zelinsky, 1992) is made, whereby grid cells are regarded as obstacle-free unless they are known to be blocked.

Below we define a version of the free-space assumption for use with general search problems. We assume a certain search graph G_M is given as input to the agent. Such a graph reflects what the agent knows about the environment, and is kept in memory throughout execution. We assume that this graph satisfies the following generalized version of the free-space assumption: if the actual search graph is $G = (S, A)$, then G_M is a *spanning supergraph* of G, i.e. $G_M = (S, A')$, with $A \subseteq A'$. Note that because G_M is a supergraph of G then $d_{G_M}(s,t) \leq d_G(s,t)$ for all $s, t \in S$, and that if h is admissible for G_M then so it is for G.

While moving through the environment, we assume the agent is capable of observing whether or not some of the arcs in its search graph $G_M = (S, A')$ are present in the actual graph. Specifically, we assume that if the agent is in state s, it is able to sense whether $(s,t) \in A'$ is traversable in the actual graph. If an arc (s,t) is not traversable, then t is inaccessible and hence the agent removes from G_M all arcs that lead to t. Note that this means that if G_M satisfies the free-space assumption initially, it will always satisfy it during execution.

Note the following fact implicit to our definitions: the environment is *static*. This is because G, unlike G_M, never changes. The free-space assumption also implies that the agent cannot discover arcs in the environment that are not present in its search graph G_M.

Many standard real-time search algorithms have the structure of Algorithm 1, which solves the search problem by iterating through a loop that runs four procedures: lookahead, heuristic learning, movement, and observation. The *lookahead* phase (Line 3) runs a time-bounded search algorithm that returns a path that later determines how the agent moves. The *heuristic learning* procedure (Line 4) changes the h-value of some of the states in the search space to make them more informed. Finally, in the *movement and observation* phase (Line 5), the agent moves along a path in the graph previously computed by the lookahead search procedure. While moving, the agent observes the environment, and prunes away from G_M any arc that is perceived to be absent in the actual environment.

Algorithm 1: A Generic Real-Time Search Algorithm

```
  Input: A search graph $G_M$, a heuristic function $h$, a goal state $g$
  Effect: The agent is moved from the initial state to a goal state if a trajectory exists
1 while the agent has not reached the goal state do
2     $s_{curr}$ ← the current state.
3     path ← LookAhead($s_{curr}$, $g$).
4     Update the heuristic function $h$.
5     Move the agent through path. While moving, observe the environment and
      update $G_M$, removing any non traversable arcs. Stop if an arc in path is removed
      or if path has been traversed completely
```

RTAA* (Koenig & Likhachev, 2006) is an instance of Algorithm 1. In its lookahead phase, it runs a bounded A* from s_{curr} towards the goal state, which executes as regular A* does but execution is stopped as soon the node with lowest f-value in *Open* is a goal state or as soon as k nodes have been expanded. The path returned is the one that connects s_{curr} and the best state in *Open* (i.e., the state with lowest f-value in *Open*). On the other hand, heuristic learning is carried out using Algorithm 2, which resets the heuristic of all states expanded by the lookahead according to the f-value of the best state in *Open*. Koenig and Likhachev (2006) prove that Algorithm 2 maintains the consistency of h if h is initially consistent.

Algorithm 2: RTAA*'s heuristic learning.

```
1 procedure Update ()
2     $f^* \leftarrow \min_{s \in Open} g(s) + h(s)$
3     for each $s \in Closed$ do
4         $h(s) \leftarrow f^* - g(s)$
```

LRTA* (Korf, 1990) is also instance of Algorithm 1; indeed, LRTA* is an instance of RTAA* when the k parameter is set to 1. In a nutshell, the most simple version of LRTA* decides where to move to by just looking at the best of s_{curr}'s neighbors, and updates the heuristic of s_{curr} also based on the heuristic of its neighbors.

It is easy to see that both RTAA* and LRTA* satisfy the real-time property since all operations carried out prior to movement take constant time. These algorithms are also *complete*—in the sense that they always find a solution if one exists—when the input heuristic is consistent. To prove completeness, heuristic learning is key. First, because learning guarantees that the state the agent moves to has a lower heuristic value compared to $h(s_{curr})$. Second, because the learning procedure guarantees that the heuristic is always bounded (in the case of RTAA*, and many other algorithms, consistency, and hence admissibility is preserved during execution).

Finally, bounds for the number of execution steps are known for some of these algorithms. LRTA*, for example, can solve any search problem in $(|S|^2 - |S|)/2$ iterations, where $|S|$ is the number nodes in the search graph (Edelkamp & Schrödl, 2011, Ch. 11).

3. Searching via Tree Reconnection

The algorithm we propose below moves an agent towards the goal state in a partially known environment by following the arcs of a so-called *ideal tree* $\mathcal{T}$. Whenever an arc in such a tree cannot be traversed in the actual environment, it carries out a search to reconnect the current state with a node in $\mathcal{T}$. In this section we describe a simple version of our algorithm which still does not satisfy the real-time property. Prior to that, we describe how $\mathcal{T}$ is built initially.

3.1 The Ideal Tree

The ideal tree intuitively corresponds to a family of paths that connect some states of the search space with the goal state. The tree is ideal because some of the arcs in the tree may not exist in the actual search graph. Formally,

Definition 1 (Ideal Tree) *Given a search problem $P = (G, c, s_{start}, g)$, and a graph G_M that satisfies the generalized free-space assumption with respect to G, the* ideal tree $\mathcal{T}$ *over P and G_M is a directed acyclic subgraph of G_M such that:*

1. *the goal state g is in $\mathcal{T}$ and has no parent (i.e., it is the root), and*
2. *if t is a child of s in $\mathcal{T}$, then (t, s) is an arc in G_M.*

Note that arcs in an ideal tree are directed and point from the children to the parent (Figure 1 depicts an ideal tree over a grid world). Properties 1 and 2 of Definition 1 imply that given an ideal tree $\mathcal{T}$ and a node s in G_M it suffices to follow the arcs in $\mathcal{T}$ (which are also in G_M) to reach the goal state g. Property 2 corresponds to the intuition of $\mathcal{T}$ being *ideal*: the arcs in $\mathcal{T}$ may not exist in the actual search graph because they correspond to arcs in G_M but not necessarily in G.

We note that in search problems in which the search graph is defined using a successor generator (as is the case of standard planning problems) it is possible to build an ideal tree by first setting which states will represent the leaves of the tree, and then computing a path to the goal from those states. A way of achieving this is to relax the successor generator (perhaps by removing preconditions), which allows including arcs in $\mathcal{T}$ that are not in the original problem. As such, Property 2 *does not* require the user to provide an inverse of the successor generator in planning problems.

The internal representation of an ideal tree $\mathcal{T}$ is straightforward. For each node $s \in S$ we store a pointer to the parent of s, which we denote by $\mathrm{p}(s)$. Formally $\mathrm{p} : S \cup \{null\} \to S \cup \{null\}$, $\mathrm{p}(null) = null$ and $\mathrm{p}(g) = null$. Notice that this representation can actually be used to describe a forest. Below, we sometimes refer to this forest as $\mathcal{F}$ and use the concept of paths in $\mathcal{F}$, that correspond to paths in some connected component of $\mathcal{F}$ that might or not be $\mathcal{T}$.

At the outset of search, the algorithm we present below starts off with an ideal tree that is also *spanning*, i.e., such that it contains all the states in S. In the general case, a spanning ideal tree can be computed by running Dijkstra's algorithm from the goal node in a graph like G_M but in which all arcs are inverted. Indeed, if $h(s)$ is defined as the distance from g to s in such a graph, an ideal tree can be constructed using the following rules: for every $s \in S \setminus \{g\}$ we define $\mathrm{p}(s) = \arg\min_{u:(s,u)\in A[G_M]} c(s, u) + h(u)$, where $A[G_M]$ are the arcs of G_M.

In some applications like real-time pathfinding in video games, when the environment is partially known a priori it is reasonable to assume that there is sufficient time for preprocessing (Bulitko et al., 2010). In preprocessing time, one could run Dijkstra's algorithm for every possible goal state. If memory is a problem, one could use so-called *compressed path databases* (Botea, 2011), which actually define spanning ideal trees for every possible goal state of a given grid.

Moreover, in gridworld pathfinding in unknown terrain, an ideal tree over an obstacle-free G_M can be quickly constructed using the information given by a standard heuristic. This is because both the Manhattan distance and the octile distance correspond to the value returned by a Dijkstra call from the goal state in 4-connected and 8-connected grids, respectively. In cases in which the grid is completely or partially known initially but there is no time for preprocessing, one can still feed the algorithm with an obstacle-free initial graph in which obstacles are regarded as accessible from neighbor states. Thus, a call to an algorithm like Dijkstra does not need to be made if there is insufficient time.

In the implementation of our algorithm for gridworlds we further exploit the fact that the tree can be built on the fly. Indeed, we do not need to set p(s) for every s before starting the search; instead, we set p(s) when it is needed for the first time. As such, no time is spent initializing an ideal tree before search. More generally, depending on the problem structure, specific implementations can exploit the fact that $\mathcal{T}$ need not be an explicit tree.

3.2 Following and Reconnecting

Our search algorithm, Follow and Reconnect with the Ideal Tree (FRIT, Algorithm 3) receives as input a search graph G_M, an initial state s_{start}, a goal state g, and a graph search algorithm $\mathcal{A}$. G_M is the search graph known to the agent initially, which we assume satisfies the generalized free-space assumption with respect to the actual search graph. $\mathcal{A}$ is the algorithm used for reconnecting with the ideal tree. We require $\mathcal{A}$ to receive the following parameters: an initial state, a search graph, and a goal-checking boolean function, which receives a state as parameter.

Algorithm 3: FRIT: Follow and Reconnect with The Ideal Tree

```
   Input: A search graph $G_M$, an initial state $s_{start}$, a goal state $g$, and a search
          algorithm $\mathcal{A}$
1  Initialization: Let $\mathcal{T}$ be an ideal tree for $G_M$.
2  Set $s$ to $s_{start}$.
3  Set $c$ to 0 and the color of each state in $G_M$ to 0.
4  Set $h_{obstacle}$ to $\infty$.
5  while $s \neq g$ do
6      Observe the environment around $s$.
7      for each newly discovered inaccesible state $o$ do
8          if $h(o) < h_{obstacle}$ then
9              $h_{obstacle} \leftarrow h(o)$.
10         Prune from $\mathcal{T}$ and $G_M$ any arcs that lead to $o$.
11     if p($s$) = null then
12         $c \leftarrow c + 1$
13         Reconnect ($\mathcal{A}, s, G_M$, InTree[$c$]($\cdot$)).
14     Movement: Move the agent from $s$ to p($s$) and set $s$ to the new position of the
       agent.
```

Algorithm 4: Reconnect component of FRIT

```
   Input: A search algorithm $\mathcal{A}$, an initial state $s$, a search graph $G_M$ and a goal
          function $f_{\text{GOAL}}(\cdot)$
1  Let $\sigma$ be the path returned by a call to $\mathcal{A}(s, G_M, f_{\text{GOAL}}(\cdot))$.
2  Assuming $\sigma = s_0 s_1, \ldots s_n$ make p($s_i$) = $s_{i+1}$ for every $i \in \{0, \ldots, n-1\}$.
```

In its initialization (Lines 1–4), it sets up an ideal tree $\mathcal{T}$ over graph G_M. As discussed above, the tree can be retrieved from a database, if pre-processing was carried out. If there is no time for pre-processing but a suitable heuristic is available for G_M, then it computes $\mathcal{T}$ on the fly. In addition it sets the value of the variable c and the color of every state to 0, and sets the variable $h_{obstacle}$ to ∞. Note that if $\mathcal{T}$ is computed on the fly, then state colors can also be initialized on the fly. $h_{obstacle}$ is used to maintain a record of the smallest heuristic value observed in an inaccessible state. The role of state colors and $h_{obstacle}$ will become clear below, when we describe reconnection and the InTree[c] function. After initialization, in the main loop (Lines 6–14), the agent observes the environment and prunes from G_M and from $\mathcal{T}$ those arcs that do not exist in the actual graph. Additionally, it updates $h_{obstacle}$ if needed. If the current state is s and the agent observes that its parent in $\mathcal{T}$ is not reachable in the actual search graph, it sets the parent pointer of s, $\mathrm{p}(s)$, to *null*. Now the agent will move immediately to state $\mathrm{p}(s)$ unless $\mathrm{p}(s) = null$. In the latter case, s is disconnected from the ideal tree $\mathcal{T}$, and a reconnection search is carried out as shown in Algorithm 4. This procedure calls algorithm $\mathcal{A}$. The objective of this search is to reconnect to some state in $\mathcal{T}$: the goal function InTree[c]($\cdot$) returns true when invoked over a state in $\mathcal{T}$ and false otherwise. Once a path is returned, we reconnect the current state with $\mathcal{T}$ through the path found and then move to the parent of s. The main loop of Algorithm 3 finishes when the agent reaches the goal.

3.2.1 The InTree[c] Function

A key component of reconnection search is the InTree[c] function that determines whether or not a state is in $\mathcal{T}$. Our implementation—shown in Algorithm 5—follows the parent pointers of the state being queried and returns true when it reaches the goal state or a state whose h-value is smaller than $h_{obstacle}$. This last condition exploits the fact that the way $\mathcal{T}$ is built (i.e.: the free-space assumption) ensures that all states that are closer to the goal than any observed obstacle must still be in $\mathcal{T}$. This is merely an optimization technique, and removing it will incur in a small performance reduction, but no change in the actions of the agent. In addition, it paints each visited state with a color c, given as a parameter. The algorithm returns false if a state visited does not have a parent or has been painted with c (i.e., it has been visited before by some previous call to InTree[c] while in the same reconnection search).

Algorithm 5: InTree[c] function

```
  Input: a vertex $s$
1 while $s \neq g$ do
2     if $h(s) < h_{obstacle}$ then
3         return true
4     Paint $s$ with color $c$.
5     if $\mathrm{p}(s) = null$ or $\mathrm{p}(s)$ has color $c$ then
6         return false
7     $s \leftarrow \mathrm{p}(s)$
8 return true
```

Figure 1 shows an example execution of the algorithm in an a priori unknown grid pathfinding task. As can be observed, the agent moves until a wall is encountered, and then continues bordering the wall until it solves the problem. It is simple to see that, had the vertical been longer, the agent would have traveled beside the wall following a similar down-up pattern.

This example reflects a general behavior of this algorithm in grid worlds: the agent usually moves around obstacles, in a way that resembles bug algorithms (LaValle, 2006; Taylor & LaValle, 2009). This occurs because the agent believes there is a path behind the wall currently known and always tries to move to such a state unless there is another state that allows reconnection and that is found first. A closer look shows that some times the agent does not walk exactly besides the wall but moves very close to it, performing a sort of zig-zag movement. This can occur if the search used does not consider the cost of diagonals. Breadth-First Search (BFS) or Depth-First Search (DFS) may sometimes prefer using two diagonals instead of two edges with cost 1.

To avoid this problem we can use a variant of BFS, that, for a few iterations, generates first the non-diagonal successors and later the diagonal ones. For nodes deeper in the search it uses the standard ordering (e.g., clockwise). Such a version of BFS achieves in practice a behavior very similar to a bug algorithm.[1] This approach was explored in previous work (Rivera et al., 2013b), and the overall improvements were shown to be small. For this paper, we use standard BFS. See Section 6.4 for a more detailed comparison to bug algorithms.

Note that our algorithm does not perform any kind of update to the heuristic h. This contrasts with traditional real-time heuristic search algorithms, which rely on increasing the heuristic value of h to exit the heuristic depressions generated by obstacles. In such a process they may need to revisit the same cell several times.

4. Satisfying the Real-Time Property

FRIT, as presented, does not satisfy the real-time property. There are two reasons for this:

R1. the number of states expanded by a call to the algorithm passed as a parameter, $\mathcal{A}$, depends on the search graph G_M rather than on a constant; and,

R2. during the execution of $\mathcal{A}$, each time $\mathcal{A}$ checks whether or not a state is connected to the ideal tree $\mathcal{T}$, function InTree[c] may visit a number of states dependent on the size of the search graph G_M.

Below we present two natural approaches to making FRIT satisfy the real-time property. The first approach is to use a slightly modified, generic real-time heuristic search algorithm as a parameter to the algorithm. The resulting algorithm is a real-time search algorithm both because it satisfies the real-time property and because the time between movements is bounded by a constant. The second approach limits the amount of reconnection search but does not guarantee that the time between movements is limited by a constant.

1. Videos can be viewed at `http://web.ing.puc.cl/~jabaier/index.php?page=research`.

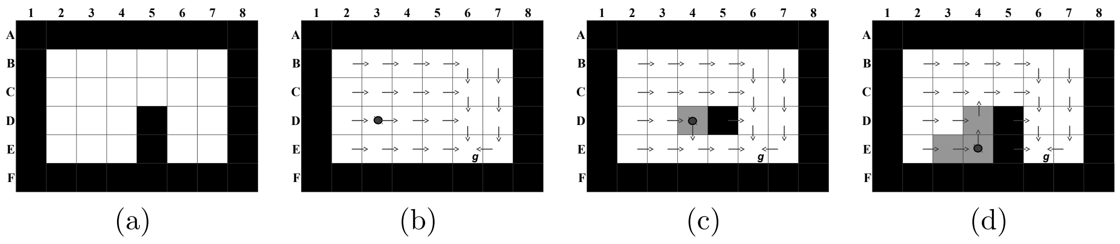


Figure 1: An illustration of some of the steps of an execution over a 4-connected grid pathfinding task, where the initial state is cell D3, and the goal is E6. The search algorithm $\mathcal{A}$ is breadth-first search, which, when expanding a cell, generates the successors in clockwise order starting with the node to the right. The position of the agent is shown with a black dot. (a) shows the true environment, which is not known a priori by the agent. (b) shows the p pointers which define the ideal tree built initially from the Manhattan heuristic. Following the p pointers, the algorithm leads the agent to D4, where a new obstacle is observed. D5 is disconnected from $\mathcal{T}$ and G_M, and a reconnection search is initiated. (c) shows the status of $\mathcal{T}$ after reconnection search expands state D4, finding E4 is in $\mathcal{T}$. The agent is then moved to E4, from where a new reconnection search expands the gray cells shown in (d). The problem is now solved by simply following the p pointers.

4.1 FRIT with Real-Time Heuristic Search Algorithms

A natural way of addressing R1 is by using a real-time search algorithm as parameter to FRIT. It turns out that it is not possible to plug into FRIT a real-time search algorithm directly without modifications. However, the modifications we need to make to Algorithm 1 are simple. We describe them below.

The following two observations justify the changes that need to be made to the pseudocode of the generic real-time search algorithm. First we observe that the objective of the lookahead search procedure of real-time heuristic algorithms like Algorithm 1 is to search towards the goal and thus the heuristic h estimates the distance to the goal. However, FRIT carries out search with the sole objective of *reconnecting* with the ideal tree, which means that both the goal condition and the heuristic have to be changed. Second, one of the main ideas underlying FRIT is to use and maintain the ideal tree $\mathcal{T}$; that is, when the agent has found a reconnecting path, the p function needs to be updated accordingly.

Algorithm 6 shows the pseudocode for the modified generic real-time heuristic search algorithm, which has two main differences with respect to Algorithm 1. First, the goal condition is now given by function $g_{\mathcal{T}}$, which returns true if evaluated with a state that is in $\mathcal{T}$. Second, Line 5 of Algorithm 6 connects the states in the path found by the lookahead search to $\mathcal{T}$. This implies also that the Reconnect procedure described in Algorithm 4 needs to be changed by that described in Algorithm 7.

Now we turn our attention to how we can guide the search towards reconnection using reconnecting heuristics. Before giving a formal definition for these heuristics, we introduce a little notation. Given the graph $G_M = (S, A)$ and the ideal tree $\mathcal{T}$ for G_M over a problem P with goal state g, we denote by $S_{\mathcal{T}}$ the set of states in $\mathcal{T}$. Now we are ready to define reconnecting heuristics formally.

Algorithm 6: A Generic Real-Time Search Algorithm for FRIT

```
Input: A search graph $G_M$, a heuristic function $h$, a goal function $g_{\mathcal{T}}(\cdot)$ that
       receives a state as parameter.
Effect: The agent is moved from the initial state to a goal state if a trajectory
        exists. The ideal tree $\mathcal{T}$ is updated.
1 while the agent has not reached a goal state do
2     $s_{curr} \leftarrow$ the current state
3     $path \leftarrow \texttt{LookAhead}(s_{curr}, g_{\mathcal{T}}(\cdot))$
4     Update the heuristic function $h$.
5     Given $path = s_0 s_1 \dots s_n$, update $\mathcal{T}$ so that $\mathrm{p}(s_i) = s_{i+1}$ for every
      $i \in \{0, \dots, n-1\}$.
6     Move the agent through the path. While moving, observe the environment and
      update $G_M$ and $\mathcal{T}$, removing any non traversable arcs and updating $h_{obstacle}$ if
      needed. Stop if the current state has no parent in $\mathcal{T}$.
```

Algorithm 7: Reconnect component for FRIT with a real-time algorithm

```
Input: A real-time search algorithm $\mathcal{A}$, an initial state $s$, a search graph $G_M$ and a
       goal function $f_{\text{GOAL}}(\cdot)$
1 Call $\mathcal{A}(s, G_M, f_{\text{GOAL}})$.
```

Definition 2 (Admissible Reconnecting Heuristic) *Given an ideal tree $\mathcal{T}$ over graph G_M and a subset B of $S_{\mathcal{T}}$, we say function $h : S \to \mathbb{R}_0^+$ is an admissible* reconnecting heuristic with respect to B *iff for every $s \in S$ it holds that $h(s) \leq d_{G_M}(s, s')$, for any $s' \in B$.*

Intuitively, a reconnecting heuristic with respect to B is an admissible heuristic over the graph G_M where the set of goal states is defined as B. As such, when Algorithm 6 is initialized with a reconnecting heuristic, search will be guided towards those connected states.

Depending on how we choose B, we may obtain a different heuristic. At first glance, it may seem sensible to choose B as $S_{\mathcal{T}}$. However, it is not immediately obvious how one would maintain (i.e., learn) such a heuristic efficiently. This is because both $\mathcal{T}$ and $S_{\mathcal{T}}$ change when new obstacles are discovered. Initially $S_{\mathcal{T}}$ contains all states but during execution, some states in S cease to belong to $S_{\mathcal{T}}$ as an arc is removed and other become members after reconnection is completed.

In this paper we propose to use an easy-to-maintain reconnecting heuristic, which, for all s is initialized to zero and then is updated in the standard way. Below, we prove that if the update procedure has standard properties, such an h corresponds to a reconnecting heuristic for the subset $B = V(E)$ of $S_{\mathcal{T}}$, where $V(E)$ is defined as follows:

$$B = V(E) = \{s \in S_{\mathcal{T}} : s \text{ has not been visited by the agent and } s \notin E\}.$$

In addition, E must be set to the set of states whose heuristic value has been potentially updated by the real-time search algorithm. The reason for this is that, by definition, all states in B should have their h-value set to zero and thus we do not want to include in B states that have been potentially modified.

Now we prove that a simple heuristic initialized as 0 for all states and updated in a standard way is indeed a reconnecting heuristic.

Proposition 1 *Let FRIT be modified to initialize h as the null heuristic. Let E be defined as the set of states that the update procedure has potentially updated.*[2] *Furthermore, assume that $\mathcal{A}$ is an instance of Algorithm 6 satisfying:*

P1. $g_{\mathcal{T}}(s)$ returns true iff $s \in S_{\mathcal{T}}$.

P2. Heuristic learning maintains consistency; i.e., if h is consistent prior to learning, then it remains as such after learning.

Then, along the execution of FRIT($\mathcal{A}$), h is a reconnecting heuristic with respect to $B = V(E)$.

Proof: First we observe that initially h is a reconnecting heuristic because it is set to zero for every state. Let s be any state in S and s' be any state in B. We prove that $h(s) \leq d(s, s')$. Indeed, let $\sigma = s_0 s_1 \ldots s_n$, with $s_0 = s$ and $s_n = s$, be a shortest path between s and s'. Since h is consistent, it holds that

$$h(s_i) \leq c(s_i, s_{i+1}) + h(s_{i+1}), \tag{1}$$

for any $i \in \{0, \ldots, n-1\}$. From where we can write

$$h(s) - h(s') = \sum_{i=0}^{n-1} h(s_i) - h(s_{i+1}) \leq \sum_{i=0}^{n-1} c(s_i, s_{i+1}) = d(s, s') \tag{2}$$

Now observe that because $s' \in B$, then the h-value of s' could have not been updated by the algorithm and therefore $h(s') = 0$, which substituted in Inequality 2, proves the desired result. ■

4.1.1 Tie-Breaking

In pathfinding, the standard approach to tie-breaking among states with equal f-values is to select the state with highest g-value. For the reconnection search, we propose a different strategy based on selecting a state based on a user-given heuristic that should guide search towards the final goal state. For example, in our experiments on grids we break ties by selecting the state with smallest octile distance to the goal. Intuitively, among two otherwise equal states, we prefer the one that seems to be closer to the final goal. This seems like a reasonable way to use information that is commonly used by other search algorithms, but unavailable to the reconnection search due to the initial use of the null heuristic.

2. Note that in practice, E is a very natural set of states. For example if RTAA* is used, the set of states that have potentially been updated are those that were expanded by some A* lookahead search.

4.1.2 Making InTree[c] Real-Time

At the beginning of Section 4 we identified R1 and R2 as the two reasons why FRIT does not satisfy the real-time property, and then discussed how to address R1 by using a real-time search algorithm. Now we discuss how to address R2.

To address R2, we simply make InTree[c] a bounded algorithm. All real-time search algorithms receive a parameter that allows them to bound the computation carried out per search. Assume that Algorithm 6 receives k as parameter. Furthermore, assume without loss of generality that lookahead search is implemented with an algorithm that constantly expands states (such as bounded A*). Then we can always choose implementation-specific constants N_E and N_T, associated respectively to the expansions performed during lookahead and the operation that follows the p pointer in the InTree[c] function. Given that e is the number of expansions performed by lookahead search and f is the number of times the p pointer has been followed in a run of the real-time search algorithm, we modify the stop condition of InTree[c] to return *false* if $N_E \cdot e + N_T \cdot f > k$. Also, we modify lookahead search to stop if the same condition holds true.

Henceforth we call FRIT_{RT} the algorithm that addresses R1 and R2 using a real-time search algorithm and a bounded version of InTree[c]. Note that because the computation per iteration of FRIT_{RT} is bounded, the time between agent moves is bounded, and thus FRIT_{RT} can be considered a standard real-time algorithm, as originally defined by Korf (1990). Note however that the time to reach a state connected to the ideal tree is not bounded, as several calls to the real-time algorithm may be required before reaching such a state.

4.2 FRIT with Bounded Complete Search Algorithms

In the previous section we proposed to use a standard real-time heuristic search algorithm to reconnect with the ideal tree. A potential downside of such an approach is that those algorithms usually find suboptimal solutions which sometimes require re-visiting the same state many times—a behavior usually referred to as "*scrubbing*" (Bulitko et al., 2011). In applications in which the quality of the solution is important, but in which there are still real-time constraints it is possible to make FRIT satisfy the real-time property in a different way.

Imagine for example, that we are in a situation in which FRIT is given a sequence of time frames, each of which is very short. After each time frame FRIT is allowed to return a movement which is performed by the agent. Such a model for real-time behavior has been termed as the *game time model* (Hernández, Baier, Uras, & Koenig, 2012b) since it has a clear application to video games in which the game's main cycle will reserve a fixed and usually short amount of time to plan the next move for each of the automated characters.

To accommodate this behavior in FRIT we can apply the same simple idea already described in Section 4.1.2, but using a *complete* search algorithm for reconnection rather than a real-time search algorithm. As described above this simply involves choosing implementation-specific constants N_E and N_T, associated respectively to the expansions performed by the (now complete) search algorithm for reconnection and the operation that follows the p pointer in the InTree[c] function. As before, given that e is the number of expansions performed by reconnection search and f is the number of times the p pointer we modify the Reconnect algorithm to return an *empty path* as soon as $N_E e + N_T f > k$ and save all local variables used by $\mathcal{A}$ and InTree[c]. Once Reconnect is called again, search is resumed at the same point it was in the previous iteration and e and f are set to 0.

Note that instead of returning an empty path other implementations may choose to move the agent in a fashion that is meaningful for the specific application. We leave a thorough discussion on how to implement such a movement strategy out of the scope of this paper since we believe that such a strategy is usually application-specific. If a movement ought to be carried out after each time frame, the agent could choose to move back-and-forth, or choose any other moving strategy that allows it to follow the reconnection path once it is found. Later, in our experimental evaluation, we choose not to move the agent if computation exceeds the parameter and discuss why this seems a good strategy in the application we chose.

Note that if a non-empty path is returned after each given time frame, then FRIT, modified in the way described above, is also a real-time search algorithm, as originally defined by Korf (1990). Finally, we note that implementing the stop-and-resume mechanism described above is easy for most search algorithms.

5. Theoretical Analysis

The results described in this section prove the termination of the algorithms and present explicit bounds on the number of agent moves performed by FRIT and FRIT_{RT} before reaching the goal. Additionally, we show that both algorithms converge in the second run so that subsequent executions of the algorithm result in identical paths. Our first theorem is correctness of the InTree[c] function.

5.1 Proofs for InTree[c]

To determine whether or not a state s belongs to the ideal tree, our InTree[c] function (Algorithm 5) follows the p pointers until the goal is reached or until some state whose h-value is smaller than $h_{obstacle}$ is reached. Here we prove that InTree[c] is correct in the sense that it returns true iff a state s belongs to the ideal tree. We start by proving the following intermediate result.

Lemma 1 *Let $H = \{s : h(s) < h_{obstacle}\}$. Reconnection search never modifies the parent pointer of a state $s \in H$.*

Proof: Take any state $s \in H$. It is clear that any call to InTree[c](s) will immediately return *true* (Algorithm 5, Lines 2 and 3). This effectively ends the search, and a path that ends in s is selected. This path does not change the parent of s. ■

Note that the property described in the Lemma holds both for FRIT and FRIT_{RT}. The bounded version of InTree[c] used for FRIT_{RT} will always answer *true* when called for a state in H. Indeed, all states in H are part of the reconnection target set B, and are correctly identified as such during execution.

Theorem 1 *When $\mathcal{T}$ is initialized as described in Section 3 and the color c is set to increment for each reconnection search,* InTree[c]*, as described in Algorithm 5, returns true for a state s iff $s \in \mathcal{T}$.*

Proof: Note that besides the exit condition established in Lines 2 and 3, the algorithm is trivially correct. It follows the parent pointers, returns *true* only if it reaches the goal, and returns *false* if it reaches a dead end or a state that has already been checked.

Let H be as defined in Lemma 1. We need to prove that all states $s \in H$ are in $\mathcal{T}$. We know that all such states have their original parent pointers set through the construction of $\mathcal{T}$ described in Section 3. Note that all the paths in the initial Ideal Tree are monotonic; for every state s different from the goal it holds that $h(s) \geq h(\mathrm{p}(s))$. From this, we know that for any state $s \in H$, $\mathrm{p}(s) \in H$ is true. This proves that all ancestors of s are in H, and therefore they represent a path that existed in the initial $\mathcal{T}$ and has not been modified. ■

5.2 Termination and Bound for FRIT$_{\mathrm{RT}}$

Our first result proves termination of the algorithm when it uses a real-time search algorithm as parameter. We provide an explicit bound on the number of agent moves until reaching the goal.

Theorem 2 *Consider the same conditions of Proposition 1 and let A be a modified real-time search algorithm as described in Algorithm 6, that requires at most $f_A(x)$ agent moves to solve any problem with x states, and such that it never updates the h-value of the goal state. Then an agent controlled by FRIT$_{RT}$ (A) reaches a goal state performing $\mathcal{O}(|S| f_A(|S|))$ moves.*

Proof: Let $\mathcal{M}$ denote the elements in the state space S that are inaccessible from any state in the connected component that contains s_{start}. Furthermore, let $\mathcal{T}$ be the ideal tree computed at initialization. Note that, by Proposition 1, we know that we use a reconnecting heuristic. By definition, this means the heuristic is always admissible for some subset of states in $\mathcal{T}$ that will always contain at least g. Therefore, we know that all reconnections are eventually successful and that each reconnection takes at most $f_A(|S|)$ steps. Notice that the agent moves at most $|S|$ steps in the Ideal Tree before it reaches an inaccessible state. Because reconnection search is only invoked after a new inaccessible state is detected, it can be invoked at most $|\mathcal{M}|$ times. By the definition of $\mathcal{T}$, we know that after $|\mathcal{M}|$ reconnections, the agent must be able to reach the goal by following $\mathcal{T}$. Therefore, the total number of steps is at most $|S| + |\mathcal{M}|(f_A(|S|) + |S|) \in O(|S| f_A(|S|))$. ■

The average length of the paths found by FRIT$_{\mathrm{RT}}$ can be expected to be much lower. Indeed, the number of reconnections is bounded by the number of obstacles that are reachable by some state in G_M, which in many cases is much lower than the total number inaccessible states.

5.3 Termination and Bound for FRIT

The following result provides a bound on the length of the solutions found by FRIT.

Theorem 3 *Given an initial tree G_M that satisfies the generalized free-space assumption, then FRIT solves P in at most $\frac{(|S|+1)^2}{4}$ agent moves.*

Proof: Let $\mathcal{M}$ and $\mathcal{T}$ be as described in the proof of Theorem 2. Note that the goal state g is always part of $\mathcal{T}$, thus $\mathcal{T}$ can never become empty and reconnection will always succeed. As for FRIT_{RT}, reconnection search can be invoked at most $|\mathcal{M}|$ times. Between two consecutive calls to reconnection search, the agent moves in a tree and thus cannot visit any state twice. Hence, the number of states visited between two consecutive reconnection searches is at most $|S| - |\mathcal{M}|$. We conclude that the number of moves until the algorithm terminates is

$$(|\mathcal{M}| + 1)(|S| - |\mathcal{M}|), \tag{3}$$

which is largest when $|\mathcal{M}| = \frac{|S|-1}{2}$. Substituting this value in (3) gives the desired result. ■

Again, the average complexity can be expected to be much lower than this bound.

5.4 Convergence

The following results prove that after termination of either FRIT or FRIT_{RT}, the agent knows a solution to the problem that is possibly shorter than the one just found.

Lemma 2 *Let $\mathcal{F}$ be the forest defined by the* p *pointers. Throughout the execution of either FRIT or $FRIT_{RT}$, there is a path σ in $\mathcal{F}$ that goes from s_{start} to the current position of the agent.*

Proof: The proof is done by induction over the number of steps taken by the agent. Let s represent the current position of the agent. Initially, the proposition is trivial, as $s_{start} = s$. Let s' be the position of the agent after moving. By the induction hypothesis, we know there is a path σ from s_{start} to s. If s' is not in σ, we know that the parent pointers of the states in σ different from s have not been modified, and therefore the path that extends σ by appending s' is valid and satisfies the property. If s' is in σ, we know that the parent pointers of states in σ that appear before s' have not been modified, and therefore there is a valid subpath of σ that goes from s_{start} to s' which satisfies the property. ■

Theorem 4 *Running the algorithm for a second time over the same problem, without reinitializing the ideal tree, results in an execution that never runs reconnection search and finds a potentially better solution than the one found in the first run.*

Proof: The proof is straightforward from Lemma 2. At the end of the execution, there is a path in $\mathcal{F}$, and specifically in $\mathcal{T}$, from s_{start} to g. Note that all states on the path have necessarily been visited during the first execution, which ensures that this new path is at most as long as the one resulting from the first execution. ■

Note that Theorem 4 implies that our algorithm can return a different path in a second trial, which can be viewed as an "optimized solution" that does not contain the loops that the first solution had. The second execution of the algorithm is naturally very fast, because reconnection search is not required.

It is interesting to note that this approach could be used with any other real-time search algorithm. By storing for each visited state the direction in which the agent moved away from it, a path with no loops that goes from s_{start} to g can be immediately extracted as soon as the execution finishes.

6. Empirical Evaluation

The objective of our experimental evaluation was to compare the performance of our algorithm with various state-of-the-art algorithms on the task of pathfinding with real-time constraints. We chose this application since it seems to be the most straightforward application of real-time search algorithms.

We compared three classes of search algorithms. For the first class, we considered state-of-the-art real-time heuristic search algorithms and the corresponding versions of FRIT_{RT} that result when it is fed with these. Specifically, we compare to RTAA* (Koenig & Likhachev, 2006) and daRTAA* (Hernández & Baier, 2012), a variant of RTAA* that may outperform it significantly. In both versions of the algorithm, the tree ideal tree is not built at the outset of search but rather built on-the-fly, using the heuristic function.

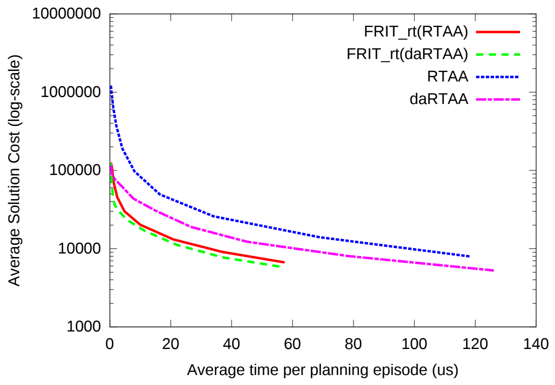
For the second class, we compared FRIT fed with a breadth-first-search algorithm to the incremental heuristic search algorithms Repeated A* (RA*) and Adaptive A* (AA*). We chose them on the one hand because it is fairly obvious how to modify them to satisfy the real-time property following the same approach we follow with FRIT, and on the other hand because they have reasonable performance. Indeed, we do not include D* Lite (Koenig & Likhachev, 2002) since it has been shown that Repeated A* is faster than D* Lite in most instances of the problems we evaluate here (Hernández, Baier, Uras, & Koenig, 2012a). Other incremental search algorithms are not included since it is not the focus of this paper to propose strategies to make various algorithms satisfy the real-time property.

Finally we compare our algorithm to Bug2 (Lumelsky & Stepanov, 1987) a so-called bug algorithm, which is an algorithm specifically designed for path-planning. Bug algorithm need very limited computational requirements to make decisions.

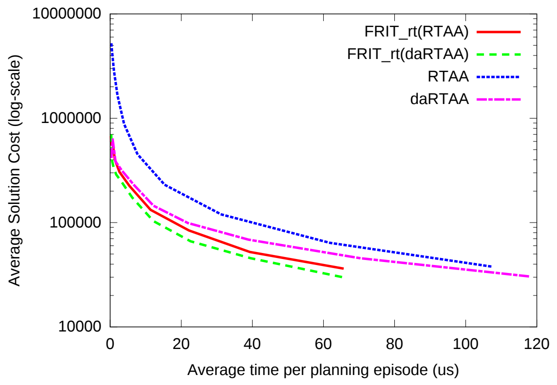
Repeated A* and Adaptive A* both run a complete A* search over the currently known search graph until the goal is reached. Then the path found is followed. While following the path, the search graph is updated with newly found obstacles. The agent stops when it reaches the goal is reached or when the path is blocked by an obstacle. When this happens, they iterate by running another A* to the goal. To make both algorithms satisfy the real-time property, we follow an approach similar to that employed in the design of the algorithm Time-Bounded A* (Björnsson, Bulitko, & Sturtevant, 2009). In each iteration, we only allow the algorithm to expand at most k states. If no path to the goal is found the agent does not move. Otherwise (the agent has found a path to the goal), the agent makes a single move on the path.

For the case of FRIT(BFS), we satisfy the real-time property as discussed above by setting both constants, N_E and N_T, to 1. This means that in each iteration, if the current state has no parent then only k states can be expanded/visited during the reconnection search and if no reconnection path is found the agent is not moved. Otherwise, if the current state has a non-null parent pointer, the agent follows the pointer.

Therefore in each iteration of FRIT(BFS), Repeated A* or Adaptive A* two things can happen: either the agent is not moved or the agent is moved one step. This moving strategy is sensible for applications like video games where, although characters are expected to move fluently, we do not want to force the algorithm to return an arbitrary move if a path has not been found, since that would introduce moves that may be perceived as pointless by the users. In contrast, real-time search algorithms return a move at each iteration.



(a) Real-time algorithms in games.



(b) Real-time algorithms in mazes.

Figure 2: Real-time algorithms: Total Iterations versus Time per Episode

We use eight-neighbor grids in the experiments since they are often preferred in practice, for example in video games (Bulitko et al., 2011). The algorithms are evaluated in the context of goal-directed navigation in a priori unknown grids. The agent is capable of detecting whether or not any of its eight neighboring cells is blocked and can then move to any one of the unblocked neighboring cells. The user-given h-values are the octile distances (Bulitko & Lee, 2006).

To carry out the experiments, we used twelve maps from deployed video games and four different mazes. Six of the maps are taken from the game *Dragon Age*, and the remaining six are taken from the game *StarCraft*. Both the maps and the mazes were retrieved from Nathan Sturtevant's pathfinding repository (Sturtevant, 2012).[3]

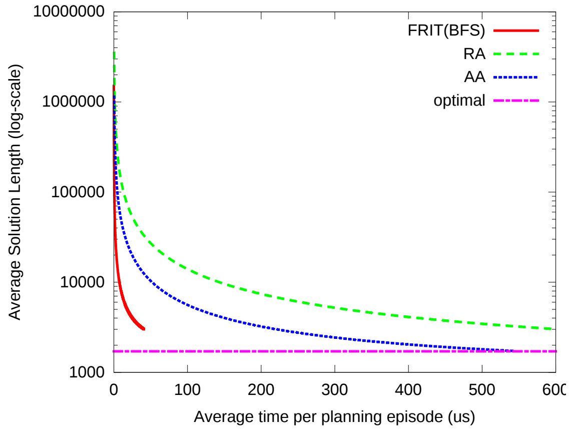
We averaged our experimental results over 500 test cases with a reachable goal cell for each map. For each test case the start and goal cells were chosen randomly. All real-time algorithms were run with 10 different parameter values. The incremental algorithms were run to completion once per test case, after which the results were processed to show the behavior corresponding to using 150,000 different values for the k parameter. All the experiments were run on a 2.00GHz QuadCore Intel Xeon machine running Linux.

6.1 Analysis of the Results for Real-Time Search Algorithms

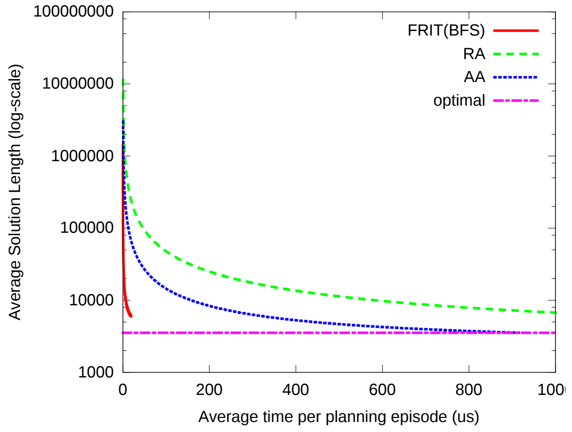
Figure 2 shows two plots of the average solution cost versus average time per planning episode for the four real-time search algorithms in games and mazes benchmarks.

We observe that for the games benchmarks FRIT_{RT} outperforms RTAA* and daRTAA* substantially. FRIT_{RT}(daRTAA*) finds solutions of about half the cost of those found by daRTAA* for any given time deadline. Moreover, the average planning time per episode needed by FRIT_{RT}(daRTAA*) to obtain a particular solution quality is about one half of that needed by daRTAA*. The improvements are more pronounced with FRIT_{RT}(RTAA*),

3. Maps used from Dragon Age: brc202d, orz702d, orz900d, ost000a, ost000t and ost100d whose sizes are 481×530, 939×718, 656×1491, 969×487, 971×487, and 1025×1024 cells respectively. Maps from StarCraft: ArcticStation, Enigma, Inferno, JungleSiege, Ramparts and WheelofWar of sizes 768×768, 768×768, 768×768, 768×768, 512×512 and 768×768 cells respectively.
The four mazes all have the same size, 512×512, and different corridor widths: 4, 8, 16 and 32.



(a) Incremental algorithms in games.



(b) Incremental algorithms in mazes.

Figure 3: Incremental algorithms: Total Iterations versus Time per Episode

where solutions for a given time deadline are at least three times cheaper than pure RTAA* and up to one order of magnitude cheaper for very small time frames. It is interesting to note that even though daRTAA* improves significantly over RTAA*, $\mathrm{FRIT}_{\mathrm{RT}}$(daRTAA*) is only marginally better than $\mathrm{FRIT}_{\mathrm{RT}}$(RTAA*).

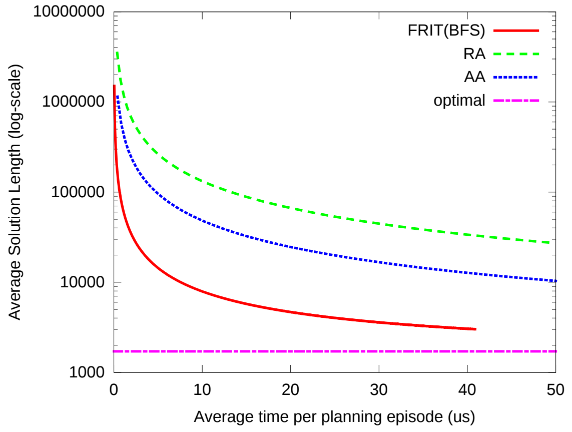
For mazes, the $\mathrm{FRIT}_{\mathrm{RT}}$ variants seem to be slightly better than daRTAA*, with bigger improvements in performance noticeable as the time deadlines are increased. The best solutions found by daRTAA* and $\mathrm{FRIT}_{\mathrm{RT}}$(daRTAA*) are of comparable lengths, but $\mathrm{FRIT}_{\mathrm{RT}}$(daRTAA*) finds these solutions requiring slightly more than half of the time per planning episode than daRTAA*.

6.2 Analysis of the Results for Incremental Algorithms Modified to Satisfy the Real-Time Property

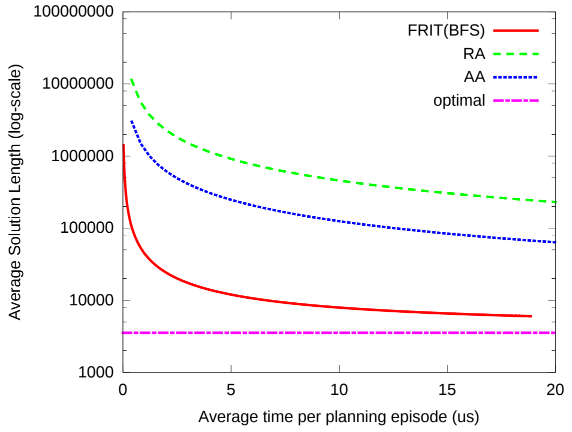
Figure 3 shows two plots of the average number of agent steps versus average time per planning episode for the incremental search algorithms used as real-time algorithms as described aboved in games and mazes benchmarks. Figure 4 shows the regions of the same plots as Figure 3 in which FRIT(BFS) appears.

We observe that FRIT(BFS) returns significantly better solutions when time constraints are very tight. Indeed, for games benchmarks our algorithm does not need more than 41μ sec per planning episode to return its best solution. Given such a time as a limit per episode, AA* requires over four times as many iterations on average. Furthermore, to obtain a solution of the quality returned by FRIT(BFS) at 41μ sec, AA* needs around 220μ sec; i.e., more than 5 times as long as FRIT(BFS). This behavior is more extreme in the case of mazes, where the best solutions for FRIT(BFS) are obtained with less than 19μ sec per planning episode. With this time limit, the number of steps required on average by AA* is a whole order of magnitude larger than the number required by FRIT(BFS).

Generally, FRIT(BFS) behaves much better than both RA* and AA*, requiring fewer iterations and less time. Nevertheless, when provided more time, FRIT(BFS) does not take advantage of it and the resulting solutions cease to improve. This can be seen both in



(a) Incremental algorithms in games.



(b) Incremental algorithms in mazes.

Figure 4: Incremental algorithms: Total Iterations versus Time per Episode (zoomed)

	FRIT(BFS)			RA*			AA*		
k	Avg. Its	Time/ep (μ s)	No moves (%)	Avg. Its	Time/ep (μ s)	No moves (%)	Avg. Its	Time/ep (μ s)	No moves (%)
1	1508631	0.0430	99.80	3505076	0.3754	99.95	1144680	0.4152	99.84
5	303483	0.2148	99.01	702029	1.8761	99.76	229967	2.0727	99.25
10	152858	0.4283	98.03	351648	3.7499	99.51	115628	4.1376	98.51
50	32401	2.0940	90.71	71343	18.655	97.60	24156	20.378	92.86
100	17370	4.0678	82.67	36305	37.077	95.29	12723	40.004	86.44
500	5449	16.115	44.74	8304	175.89	79.42	3607	172.41	52.15
1000	4035	24.840	25.38	4901	322.74	65.13	2583	274.35	33.20
5000	3073	39.316	2.046	2261	915.66	24.44	1854	474.29	6.904
10000	3026	40.487	0.501	1947	1171.9	12.26	1775	514.88	2.764
50000	3011	40.851	0.030	1726	1458.9	1.041	1728	524.55	0.117
100000	3011	40.869	0.007	1711	1484.7	0.133	1726	543.66	0.014

Table 1: Relationship between search expansions and number of iterations in which the agent does not move in games maps. The table shows a parameter k for each algorithm. In the case of AA* and Repeated A* the parameter corresponds to the number of expanded states. In case of FRIT, the parameter corresponds to the number of visited states during an iteration. In addition, it shows average time per search episode (Time/ep), and the percentage of iterations in which the agent did not move with respect to the total number of iterations (No moves).

Figure 3, across both sets of benchmarks, and Table 1. As an example of this, we can see that for $k = 5000$ to $k = 100000$ the number of iterations required to solve the problem only decreases by 62 steps, and the time used per search episode only increases by 1.55μ sec. Effectively, this means that the algorithm does not use the extra time in an advantageous way. This is in contrast to what is usually expected for real-time search algorithms.

An interesting variable to study is the number of algorithm iterations in which the agent did not return a move because the algorithm exceeded the amount of computation established by the parameter without finishing search. As we can see in Table 1, FRIT, using BFS as its parameter algorithm, has the best relationship between time spent per episode and the percentage of no-moves over the total number of moves. To be comparable to other real-time heuristic search algorithms, it would be preferrable to reduce the number of incomplete searches as much as possible. With this in mind, we can focus on the time after which the amount of incomplete searches is reduced to less than 1%. Notice that for FRIT(BFS) this is somewhere around 40 μs, whereas for AA* and RA* this requires times of over 514 μs and 1458 μs respectively.

6.3 Comparison of the Two Approaches

Figure 5 shows a plot of the average time per planning episode versus average number of agent steps for both FRIT_{RT}(daRTAA*) and FRIT(BFS) in games benchmarks. We observe that FRIT(BFS) obtains better resuts for most time limits. Indeed, for any given time deadline of more than 10μ sec, FRIT(BFS) finds a solution that is about half as long as that found by FRIT_{RT}(daRTAA*). For smaller time deadlines the results are similar for both algorithms. Furthermore, the best solution obtained by FRIT(BFS) is, on average, less than 60% as long as the best solution obtained by FRIT_{RT}(daRTAA*). As mentioned above, this particular solution requires a time deadline of less than 41μ sec per planning episode. The number of no-moves incurred with this time limit in our experiments was of only 465 iterations throughout all the experiments in games benchmarks, which corresponds to approximately one no-move every 40,000 moves.

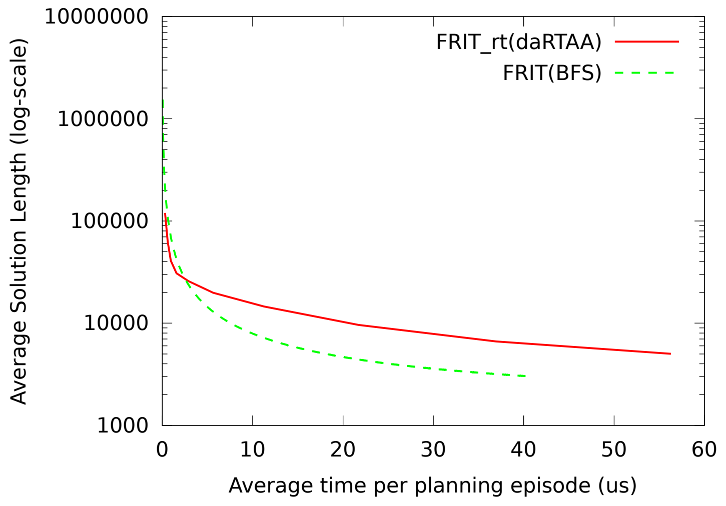


Figure 5: Comparison of FRIT using a real-time algorithm versus FRIT as an incremental algorithm in games benchmarks.

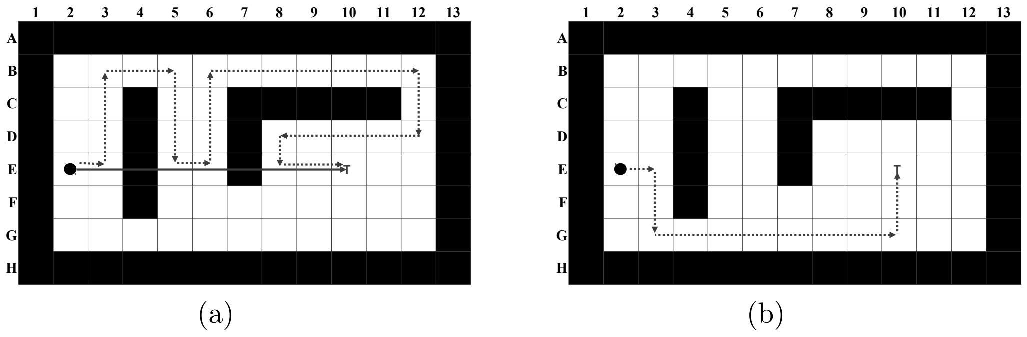


Figure 6: Bug2 (a) and FRIT (b) in a pathfinding scenario in which the goal cell is E10 and the initial cell is E2. The segmented line shows the path followed by the agent and the filled arrow is the m-line.

6.4 Comparison to the Bug2 Algorithm

Bug algorithms (LaValle, 2006; Taylor & LaValle, 2009) are a family of pathfinding algorithms for continuous 2D terrain. They make their decisions based on sensory input, require very limited time and memory resources, and are inspired by the behavior of insects while finding their way through obstacles. Bug algorithms are not heuristic as they do not utilize a heuristic function to make decisions (Rao, Kareti, Shi, & Iyengar, 1993).

Bug2 (Lumelsky & Stepanov, 1987) is a bug algorithm that is simple to implement and is guaranteed to reach the goal. An agent controlled by Bug2 will follow a straight line connecting the initial position with the final position—the so-called m-line—, until encountering an obstacle or reaching the goal. If an obstacle is encountered, it saves the position at which the obstacle was hit in a variable called *hit point* and then starts following the boundary of the obstacle (either clockwise or counterclockwise) until the m-line is encountered again. Then, if the current position is closer to the goal than the hit point, the agent starts following the m-line again and the process repeats.

Figure 6 compares the behaviors of FRIT and Bug2. In this particular situation, Bug2 does not make a good decision and FRIT solves the problem fairly quickly. Of course it is possible to contrive families of problems in which Bug2 will always outperform FRIT.

We implemented Bug2 for 8-connected grid worlds. We forbid diagonal movements between two obstacles, as this essentially has the effect of changing the direction in which an obstacle's boundary is being followed. To make the comparison fair, we also ran FRIT(BFS) with this additional restriction. We used the same game maps, and generated 500 solvable random problems for each of them.

Results for FRIT(BFS) are shown in Table 2. The average cost of solutions obtained by Bug2 was 6546, requiring 5766 iterations. In addition, the average runtime was 2317 μs, which yields, on average, 0.4 μs spent per iteration. FRIT(BFS) spends around 0.4 per search episode when $k = 12$, and requires about 20 times more iterations to solve the problem yielding about 97% of no-moves. To obtain a solution comparable to that of Bug2, FRIT(BFS) requires k to be set to around 462, which yields an average time per search episode close to 12 μs, returning a no-move on 47% of the iterations. Finally, if more time

k	Avg. Its	Time/ep (μ s)	No moves (%)
1	1544087	0.0346	99.81
5	310579	0.1725	99.03
10	156411	0.3442	98.07
50	33314	1.6830	90.91
100	14715	3.2712	83.01
500	5521	13.009	45.45
1000	4070	20.132	26.02
5000	3079	32.162	2.207
10000	3027	33.193	0.542
50000	3012	33.520	0.026
100000	3011	33.532	0.006

Table 2: Performance Indicators for FRIT(BFS) when no diagonal movements are allowed between obstacles.

is available, FRIT(BFS) returns solutions on average more than 50% cheaper than those obtained by Bug2.

As a conclusion we observe that in pathfinding applications Bug2 runs faster than any other search algorithm we tried. As a disadvantage, Bug2 is specific to pathfinding and cannot exploit additional time per episode to obtain a better solution, yielding solutions that are longer than those obtained by FRIT(BFS) when more time is available. Therefore bug algorithms seem to be recommended for real-time pathfinding applications in which there is very little time available per iteration. When more time is available FRIT(BFS) is the recommended algorithm, leaving AA* as a choice for applications in which there is significantly more time available.

6.5 On the Usefulness of Reconnecting Heuristics

Definition 2 introduced the idea of *admissible reconnecting heuristics*, which we argued are important to guide search towards reconnection. A natural question to ask is whether or not these heuristics are key to the performance of FRIT$_{\text{RT}}$. Indeed, an admissible heuristic for the problem is formally an admissible reconnecting heuristic since we simply need to define $B = \{g\}$ in Definition 2. Nevertheless, intuitively the problem's heuristic does not guide towards reconnection with the ideal tree.

To evaluate the usefulness of reconnecting heuristics, we implemented a version of FRIT$_{\text{RT}}$ (RTAA*) that:

1. uses the problem's heuristic to guide the A* search,

2. breaks ties in favor of states with greater g-value, and that

3. learns h using RTAA*'s learning rule.

We compared it to our standard FRIT$_{\text{RT}}$ (RTAA*), which uses the admissible reconnection $h = 0$ to guide the A* search, uses the problem heuristic as a first tie breaker, uses the g-value as a second tie breaker, and learns h using RTAA*'s learning rule. We ran the algorithms over the same 12 game maps, using the same configuration described above.

Figure 7 shows the relative performance of the algorithms, confirming that indeed in these pathfinding applications, using reconnection heuristics is key to performance. Using a goal heuristic to guide for reconnection performs more similar to the baseline (RTAA*). This is because FRIT$_{RT}$, used with the problem's heuristic, can be seen as a version of RTAA* that stops the A* search early if it expands a state the agent believes to be connected to the goal (i.e., in the ideal tree). Although stopping search early saves time with respect to RTAA*, the more expensive goal condition—which verifies that a state is in the ideal tree—seems to counter time savings unless the lookahead parameter is high.

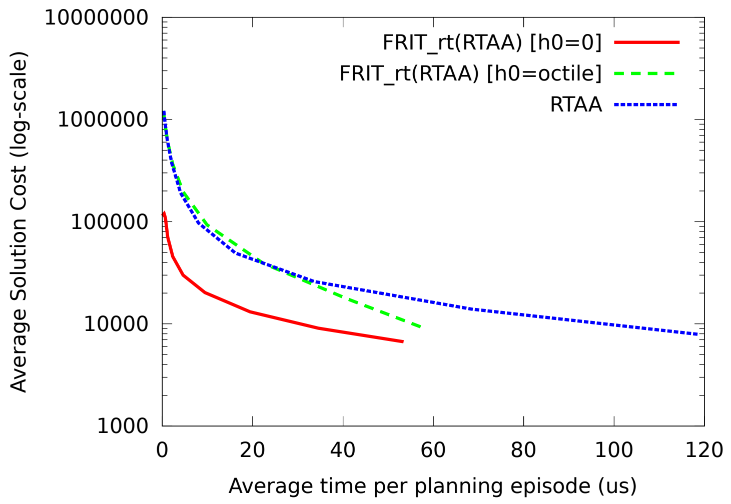


Figure 7: Comparison of FRIT$_{RT}$ using a reconnecting heuristic (h=0) to guide search versus FRIT$_{RT}$ using the problem's heuristic.

7. Related Work

Incremental Heuristic Search and Real-time Heuristic Search are two heuristic search approaches to solving search problems in partially known environments using the free-space assumption that are related to the approach we propose here. Incremental search algorithms based on A*, such as D* Lite (Koenig & Likhachev, 2002), Adaptive A* (Koenig & Likhachev, 2005) and Tree Adaptive A* (Hernández, Sun, Koenig, & Meseguer, 2011), reuse information from previous searches to speed up the current search. The algorithms can solve sequences of similar search problems faster than Repeated A*, which performs repeated A* searches from scratch.

During runtime, most incremental search algorithms, like our algorithm, store a graph in memory reflecting the current knowledge of the agent. In the first search, they perform a complete A* (backward or forward), and in the subsequent searches they perform less intensive searches. In contrast to our algorithm, such searches return *optimal* paths connecting the current state with the goal. FRIT is similar to incremental search algorithms in the sense that it uses the ideal tree, which is information that, in some cases, may have been computed using search, but differs from them in that the objective of the search is not to compute optimal paths to the goal. Our algorithm leverages the speed of simple blind search and does not need to deal with a priority queue, which is computationally expensive to handle.

Many state-of-the-art real-time heuristic search algorithms (e.g., Koenig & Sun, 2009; Koenig & Likhachev, 2006; Sturtevant & Bulitko, 2011; Hernández & Baier, 2012; Rivera, Baier, & Hernández, 2013a), which satisfy the real-time property, rely on updating the heuristic to guarantee important properties like termination. Our algorithm, on the other hand, does not need to update the heuristic to guarantee termination. Like incremental search algorithms, real-time heuristic search algorithms usually carry out search for a path between the current node and the goal state. Real-time heuristic search algorithms cannot return a likely better solution after the problem is solved without performing any search at all (cf. Theorem 4). Instead, when running multiple trials they eventually converge to an optimal solution or offer guarantees on solution quality. Our algorithm does not offer guarantees on solution quality, even though experimental results are positive.

HCDPS (Lawrence & Bulitko, 2010) is a real-time heuristic algorithm that does not employ learning. This algorithm is tailored to problems in which the agent knows the map initially, and in which there is time for preprocessing.

The idea of reconnecting with a tree rooted at the goal state is not new and can be traced back to bi-directional search (Pohl, 1971). A recent Incremental Search algorithm, Tree Adaptive A* (Hernández et al., 2011), exploits this idea to make subsequent searches faster. Real-Time D* (RTD*) (Bond, Widger, Ruml, & Sun, 2010) uses bi-directional search to perform searches in dynamic environments. RTD* combines Incremental Backward Search (D*Lite) with Real-Time Forward Search (LSS-LRTA*).

Finally, our notion of generalized free-space assumption is related to that proposed by Bonet and Geffner (2011), for the case of planning in partially observable environments. Under certain circumstances, they propose to set unobserved variables in action preconditions in the most convenient way during planning time, which indeed corresponds to adding more arcs to the original search graph.

8. Summary and Future Work

We presented FRIT, a search algorithm that follows a path in a tree—the ideal tree—that represents a family of solutions in the graph currently known by the agent. The algorithm is simple to describe and implement, and does not need to update the heuristic to guarantee termination. FRIT uses a secondary search algorithm to search for a state in the ideal tree when the agent becomes disconnected from it. We show that, with slight modifications, we can a real-time search algorithm to search for reconnection, and we obtain a real-time version of FRIT, FRIT_{RT}. In addition, we propose a different way of using FRIT in some applications that use real-time search by feeding it with bounded, complete search algorithms.

We provide theoretical results proving that both FRIT and FRIT_{RT} always find solutions if they exist. Furthermore, we give explicit bounds on the length of the obtained solutions. Finally, we prove that both algorithms converge after two trial runs.

Our experiments show that the proposed algorithms return solutions faster than other state-of-the-art real-time search algorithms. In particular FRIT(daRTAA*) substantially improves performance over daRTAA*, a state-of-the-art Real-Time Search algorithm. Larger performance improvements are observed when time constraints are tighter. Additionally, we compare both our approaches and show that FRIT(BFS)—that is, FRIT fed with the breadth first search algorithm—produces similar or better results for tight time constraints. In a comparison with the pathfinding-specific Bug2 algorithm, we concluded than when very little time per search episode is given Bug2 delivers best-quality solutions, followed by FRIT(BFS), and eventually, when significantly more time is available, followed by state-of-the-art incremental search algorithms.

As a disadvantage of our approach, we note that FRIT cannot exploit more computational time as other algorithms do. Indeed, other incremental heuristic search algorithms will return better quality solutions if allowed large time constraints, while FRIT will generally not converge asymptotically to the optimal path if given arbitrary time.

We have left out of the scope of the paper how FRIT could be used in more general search spaces like the ones that can be described using a planning language like STRIPS (Fikes & Nilsson, 1971) or PDDL (McDermott, 1998). In planning domains, search graphs are implicitly defined by actions defined in terms of preconditions an effects. Computing ideal trees using Dijkstra's algorithm, as we suggested above, is not simple since it requires the generation of a number of states exponential in the size of the problem description. Moreover, it is not immediately obvious either how to build an ideal tree from a standard domain-independent heuristic, as we have done in pathfinding. Indeed, while there exist domain-independent planning heuristics that are admissible (e.g., Haslum & Geffner, 2000), it is easy to show that they do not correspond to a perfect heuristic over a spanning supergraph of the original search space, which implies that it is not possible to use them directly to construct an ideal tree, as loops may be easily formed. Therefore, further investigation is needed in order to adapt the techniques we have presented here for planning applications.

Acknowledgments

Nicolás Rivera, León Illanes, and Jorge Baier were partly funded by Fondecyt Project Number 11110321. We thank the anonymous JAIR reviewers, for their valuable input, and the SoCS'2013 reviewers for their comments on an earlier version of this paper.

References

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

Bond, D. M., Widger, N. A., Ruml, W., & Sun, X. (2010). Real-time search in dynamic worlds. In *Proc. of the 3rd Symposium on Combinatorial Search (SoCS)*, Atlanta, Georgia.

Bonet, B., & Geffner, H. (2011). Planning under partial observability by classical replanning: Theory and experiments. In *Proc. of the 22nd Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, pp. 1936–1941, Barcelona, Spain.

Botea, A. (2011). Ultra-fast Optimal Pathfinding without Runtime Search. In *Proc. of the 7th Annual Int'l AIIDE Conference (AIIDE)*, Palo Alto, California.

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

Bulitko, V., Björnsson, Y., & Lawrence, R. (2010). Case-based subgoaling in real-time heuristic search for video game pathfinding. *Journal of Artificial Intelligence Research*, *38*, 268–300.

Bulitko, V., Björnsson, Y., Lustrek, M., Schaeffer, J., & Sigmundarson, S. (2007). Dynamic control in path-planning with real-time heuristic search. In *Proc. of the 17th Int'l Conf. on Automated Planning and Scheduling (ICAPS)*, pp. 49–56.

Bulitko, V., Björnsson, Y., Sturtevant, N., & Lawrence, R. (2011). *Real-time Heuristic Search for Game Pathfinding*, pp. 1–30. Applied Research in Artificial Intelligence for Computer Games. Springer Verlag.

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

Fikes, R., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3/4), 189–208.

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

Haslum, P., & Geffner, H. (2000). Admissible heuristics for optimal planning. In *Proc. of the 5th Int'l Conf. on Artificial Intelligence Planning and Systems (AIPS)*, pp. 140–149.

Hernández, C., & Baier, J. A. (2011). Fast subgoaling for pathfinding via real-time search. In *Proc. of the 21th Int'l Conf. on Automated Planning and Scheduling (ICAPS)*, Freiburg, Germany.

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

Hernández, C., Baier, J. A., Uras, T., & Koenig, S. (2012a). Position paper: Incremental search algorithms considered poorly understood. In *Proc. of the 5th Symposium on Combinatorial Search (SoCS)*.

Hernández, C., Baier, J. A., Uras, T., & Koenig, S. (2012b). TBAA*: Time-Bounded Adaptive A*. In *Proc. of the 10th Int'l Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS)*.

Hernández, C., Sun, X., Koenig, S., & Meseguer, P. (2011). Tree adaptive A*. In *Proc. of the 10th Int'l Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS)*, Taipei, Taiwan.

Ishida, T. (1992). Moving target search with intelligence. In *Proc. of the 10th National Conf. on Artificial Intelligence (AAAI)*, pp. 525–532.

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

Koenig, S., & Likhachev, M. (2002). D* lite. In *Proc. of the 18th National Conf. on Artificial Intelligence (AAAI)*, pp. 476–483.

Koenig, S., & Likhachev, M. (2005). Adaptive A*. In *Proc. of the 4th Int'l Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS)*, pp. 1311–1312.

Koenig, S., & Likhachev, M. (2006). Real-time Adaptive A*. In *Proc. of the 5th Int'l Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS)*, pp. 281–288.

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

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

LaValle, S. M. (2006). *Planning algorithms.* Cambridge University Press.

Lawrence, R., & Bulitko, V. (2010). Taking learning out of real-time heuristic search for video-game pathfinding. In *Australasian Conf. on Artificial Intelligence*, pp. 405–414.

Lumelsky, V. J., & Stepanov, A. A. (1987). Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, *2*, 403–430.

McDermott, D. V. (1998). PDDL — The Planning Domain Definition Language. Tech. rep. TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Pohl, I. (1971). Bi-directional heuristic search. In *Machine Intelligence 6*, pp. 127–140. Edinburgh University Press, Edinburgh, Scotland.

Rao, N. S., Kareti, S., Shi, W., & Iyengar, S. S. (1993). Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms. Tech. rep. ORNL-TM-12410, Oak Ridge National Laboratory.

Rivera, N., Baier, J. A., & Hernández, C. (2013a). Weighted real-time heuristic search. In *Proc. of the 11th Int'l Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS).*

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

Sharon, G., Sturtevant, N., & Felner, A. (2013). Online detection of dead states in real-time agent-centered search. In *Proc. of the 6th Symposium on Combinatorial Search (SoCS)*, Leavenworth, WA, USA.

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

Sturtevant, N. R., & Bulitko, V. (2011). Learning where you are going and from whence you came: h- and g-cost learning in real-time heuristic search. In *Proc. of the 22nd Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, pp. 365–370, Barcelona, Spain.

Taylor, K., & LaValle, S. M. (2009). I-bug: An intensity-based bug algorithm. In *Proc. of the 2009 IEEE Int'l Conf. on Robotics and Automation (ICRA)*, pp. 3981–3986.

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