Reconnecting with the Ideal Tree: An Alternative to Heuristic Learning in Real-Time Search

Nicolás Rivera
Dept. of Computer Science
Pontificia Universidad
Católica de Chile
Santiago, Chile

León Illanes
Dept. of Computer Science
Pontificia Universidad
Católica de Chile
Santiago, Chile

Jorge A. Baier
Dept. of Computer Science
Pontificia Universidad
Católica de Chile
Santiago, Chile

Carlos Hernández
Depto. de Informática
Universidad Católica
de la Ssma. Concepción
Concepción, Chile

Abstract

In this paper, we present a conceptually simple, easy-to-implement real-time search algorithm suitable for a priori partially known environments. Instead of performing a series of searches towards the goal, like most Real-Time Heuristic Search Algorithms do, our algorithm follows the arcs of a tree $\mathcal{T}$ rooted in the goal state that is built initially using the heuristic h. 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 our algorithm 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 need not be guided by h, since the search objective is not to encounter the goal. Furthermore, h need not be updated. We implemented versions of our algorithm that utilize various blind search algorithms for reconnection. We show experimentally that these implementations significantly outperform state-of-the-art real-time heuristic search algorithms for the task of pathfinding in grids. In grids, our algorithms, which do not incorporate any geometrical knowledge, naturally behaves similarly to a bug algorithm, moving around obstacles, and never returning to areas that have been visited in the past. In addition, we prove theoretical properties of the algorithm.

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 et al. 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 the provide the main loop with quick moves that allow implementing continuous character moves.

To apply real-time heuristic search algorithms in game technology when the objective is to move an agent that behaves as if the environment were *known* to it, one needs some sort of preprocessing (e.g., Bulitko et al. 2007, Bulitko, Björnsson, and Lawrence 2010, Hernández and Baier 2011) to produce an algorithm that moves the agent in a way that looks reasonable to a human observer. Unfortunately, no game developer would want to use standard real-time heuristic search algorithms to simulate the movement of an intelligent agent in a *partially known* or *unknown* terrain. This is because these algorithms generate back-and-forth moves that look very irrational to a human observer (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. A perfect heuristic update, while possible, cannot be afforded under very tight time constraints. Left with limited time for updating, the heuristic cannot be learned perfectly and hence agents are not moved perfectly.

In this paper we present FRIT, a real-time search algorithm that does not rely on heuristic learning to control the agent. 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 search is performed to 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 the ideal tree, and it can continue following a branch.

We evaluated our algorithm over standard game and maze pathfinding benchmarks using a blind-search algorithm 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 the state-of-the-art real-time heuristic search algorithms we compared to, when the search effort is fixed. Upon inspection of the route followed by the agent, we observe 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 and 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 and Bulitko 2011; Sharon, Sturtevant, and Felner 2013)—without implementing a specific mechanism to detect them.

We also compared our algorithm to incremental heuristic search algorithms 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 a simple proof for termination and provide a bound on the number of moves required to find a solution in arbitrary graphs.

The rest of the paper is organized as follows. In the next section we describe the background necessary for the rest of the paper. Then we describe our algorithm in detail. We continue presenting a short theoretical analysis, followed by a description of our experimental evaluation. We then describe other related work, and finish with a summary.

Background

The search problems we deal with in this paper can be described by a tuple $P = (G, c, s_0, g)$, where $G = (S, A)$ is a digraph that represents the search space. The set S represents the *states* and the arcs in A represent all available actions. We assume that S is finite, that A does not contain elements of form (s, s), and that G has a strongly connected component that contains both s_0 and g, and furthermore that all states reachable from s_0 are in such a component. 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.

Real-Time Search The objective of a real-time search algorithm problem is to move an agent from s_0 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.

When searching in partially known environments, real-time algorithms assume the search space has a particular structure. 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 there is sufficient information to the opposite.

Below we design our real-time search algorithm for general problems. As other real-time search algorithms do, we assume a certain 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 furthermore that such a graph satisfies a generalized version of the free-space assumption. In particular, if the actual search graph is $G = (S, A)$, then G_M corresponds a *spanning supergraph* of G, i.e. $G_M = (S, A')$, with $A \subseteq A'$.

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. In particular, 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 that implicit to our definition is that 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.

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 between s and t in the graph G. Note that if G' is a spanning subgraph of G, then $d_G(s, t) \leq d_{G'}(s, t)$.

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 $s \in S$. Observe that if G' is a spanning subgraph of G, and h is admissible for G, then h is also admissible for G'.

Many standard real-time search algorithms inherit the structure of the LRTA* algorithm (Korf 1990) (Algorithm 1), which solves the problem by iterating through a loop that runs four procedures: observation, lookahead, heuristic update and movement. In LRTA*, the observation phase (Line 2) prunes arcs from G_M, the lookahead phase (Line 4), chooses a neighbor of the current state based on the estimated cost to the final goal. The heuristic update phase (Line 5), updates the heuristic value for the current state based on those of its neighbors. This phase is necessary to prove termination of the algorithm. Finally, in the movement phase (Line 7), the agent moves to the position chosen in the lookahead phase. It is easy to see that LRTA* satisfies the real-time property since all operations carried out prior to the movement take constant time. Generalizations of LRTA*, such as LSS-LRTA* (Koenig and Sun 2009), replace the lookahead by an A* search towards the goal, and the update phase by some algorithm that may update the heuristic of several states. The update of the heuristic is key to enable a proof of termination of LRTA* and most of its variants. Furthermore, LRTA* can solve any search problem in $(|S|^2 - |S|)/2$ agent iterations, where $|S|$ is the number

Algorithm 1: LRTA*

Input: A search graph G_M, a heuristic function h

while *the agent has not reached the goal state* **do**
- Observe the environment and update G_M, removing any arcs to states that are observed inaccessible.
- $s \leftarrow$ the current state
- $next \leftarrow \arg\min_{t:(s,t) \text{ in } G_M} [c(s,t) + h(t)]$
- **if** $h(s) < c(s, next) + h(next)$ **then**
  - $h(s) \leftarrow c(s, next) + h(next)$
- Move to $next$.

nodes in the search graph (Edelkamp and Schrödl 2011, Ch. 11).

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 does not satisfy the real-time property, and then show how this algorithm can be transformed into one that does. Prior to that, we describe how $\mathcal{T}$ is built initially.

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 problem. Formally,

Definition 1 (Ideal Tree) *Given a search problem* $P = (G, c, s_0, g)$ *and graph* G_M *that satisfies the generalized free-space assumption, the* ideal tree $\mathcal{T}$ *over* P *and* G_M *is a tree of states that satisfies the following properties.*

1. *its root is the goal state* g*, and*
2. *if* s *is the parent of* t *in* $\mathcal{T}$*, then* (t, s) *is an arc in* G_M.

Properties 1 and 2 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 only correspond to arcs in G_M.

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.

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 $\mathrm{p}(g) = null$.

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 the Dijkstra 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 clearly 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 videogames, when the environment is partially known a priori it is reasonable to assume that there is sufficient time for preprocessing (Bulitko, Björnsson, and Lawrence 2010). In preprocessing time, one could run Dijkstra for every possible goal state. If memory is a problem, one could use so-called *compressed path databases* (Botea 2011), which actually define ideal trees for every possible goal state of a given grid.

Moreover, in gridworld pathfinding in initially 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 no sufficient 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 $\mathrm{p}(s)$ for every s before starting the search; instead, we set $\mathrm{p}(s)$ only when it is needed for the first time. As such, there is no time is spent initializing a spanning ideal tree before search. More generally, depending on the problem structure, different implementations can exploit the fact that $\mathcal{T}$ need not be an explicit tree.

Moving and Reconnecting

Our search algorithm (Algorithm 2) receives as input a search graph G_M, an initial state s_0, 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.

In its initialization (Lines 1–3), 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 $\mathcal{T}$ can be computed on the fly. In addition it sets value of the variable c and the color of every

Algorithm 2: FRIT: Follow and Reconnect with The Ideal Tree

Input: A search graph G_M, an initial state s_0, a goal state g, and a search algorithm $\mathcal{A}$

1 **Initialization:** Let $\mathcal{T}$ be a spanning ideal tree for G_M.
2 Set s to s_0
3 Set c to 0 and the color of each state in G_M to 0
4 **while** $s \neq g$ **do**
5   Observe the environment around s and prune from $\mathcal{T}$ and G_M any arcs that lead to newly discovered inaccessible states.
6   **if** $\mathrm{p}(s) = null$ **then**
7   **Reconnect:**
8         $c \leftarrow c + 1$
9   Let σ be the path returned by a call to $\mathcal{A}(s, G_M, \textsc{InTree}(\mathcal{T}, c))$
10   Assuming $\sigma = x_0, x_1, ..., x_n$ make $\mathrm{p}(x_i) = x_{i+1}$ for every $i \in \{0, \ldots, n-1\}$.
11   **Movement:** Move the agent from s to $\mathrm{p}(s)$ and set s to the new position of the agent

Algorithm 3: InTree function

Input: a vertex s, a color c

while $s \neq g$ **do**
    paint s with color c.
    if $\mathrm{p}(s) = null$ *or* $\mathrm{p}(s)$ *has color* c **then**
        return false
    $s \leftarrow \mathrm{p}(s)$
return true

state to 0. The role of state colors will become clear below when we describe reconnection.

After initialization, in the main loop (Lines 5–11), the agent observes the environment and prunes from G_M and from $\mathcal{T}$ those arcs that do not exist in the actual graph. If the current state is s and the agent observes that its parent 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 by calling algorithm $\mathcal{A}$. The objective of this search is to reconnect to some state in $\mathcal{T}$: the goal function $\textsc{InTree}(\mathcal{T}, c)$ returns true when invoked over a state in $\mathcal{T}$ and false otherwise. Once a path is returned, it reconnects the current state with $\mathcal{T}$ through the path found and then move to the parent of s. The loop finishes when the agent reaches the goal.

The InTree Function A key component of reconnection search is the InTree function that determines whether or not a state is in $\mathcal{T}$. Our implementation, shown in Algorithm 3, follows the parent pointers of the state being queried and returns true if the goal can be reached. 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 while in the same reconnection search).

Figure 1 shows an example execution of the algorithm in an a priori unknown grid pathfinding task. As can be observed, the agent is moved until a wall is encountered, and then continues bordering the wall until it solves the problem. It is simple to see that, were the vertical wall 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 and 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 before. A closer look shows that some times the agent does not walk exactly besides the wall but moves very close to them perform 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 contrasts with traditional real-time heuristic search algorithms, which rely on increasing the heuristic value of the heuristic h to exit the heuristic depressions generated by obstacles. In such a process they may need to revisit the same cell several times.

Satisfying the Real-Time Property

As presented, Algorithm 2 does not satisfy the real-time property. Indeed, each call to $\mathcal{A}$ or InTree may visit a number of states dependent on the size of the search graph. It is straightforward, however, to convert this algorithm to a real-time one by using ideas previously used in algorithms such as Time-Bounded A* (Björnsson, Bulitko, and Sturtevant 2009). Time-Bounded A* is a real-time algorithm for a priori known domains that simply runs an A* search to the goal. In each time interval, it expands k nodes and then moves the agent. Eventually, when the goal is found, no further search is needed and the agent is moved straight to the goal.

Analogously, given a parameter k, our algorithm can be modified to stop reconnection search as soon as $n_E^{\mathcal{A}} + m \times n_V^{\textsc{InTree}} > k$, where $n_E^{\mathcal{A}}$ is the number of states expanded by $\mathcal{A}$, $n_V^{\textsc{InTree}}$ is the number of states visited by InTree, and m is a constant chosen specifically for the application. Once search is stopped a decision on the movement has to be

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

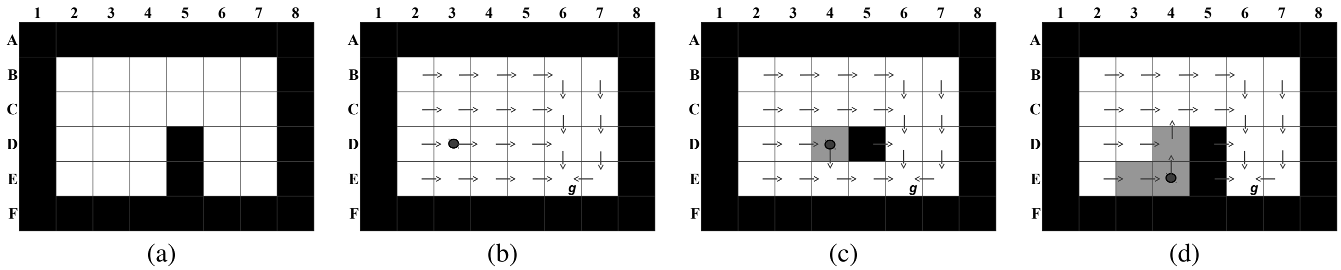

Figure 1: An illustration of some of the steps of an execution over a 4-connected grid pathfinding task, where the initial state is cell D3, and the goal is E6. The search algorithm $\mathcal{A}$ is breadth-first search, which, when expanding a cell, generates the successors in clockwise order starting 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.

made. Depending on the type of application, an implementation may choose not to move the agent at all, or to move it in a meaningful way. We leave a thorough discussion on how to implement such a movement strategy out of the scope of the paper since we believe that such a strategy is usually application-specific. If a movement ought to be carried out, 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.

Finally, we note that implementing this stop-and-resume mechanism is easy for most search algorithms.

Theoretical Analysis

Our first result proves termination of the algorithm and provides an explicit bound on the number of agent moves until reaching the goal.

Theorem 1 *Given an initial tree G_M that satisfies the generalized free-space assumption, then Algorithm 2 solves P in at most $\frac{(|S|+1)^2}{4}$ agent 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_0. Furthermore, let $\mathcal{T}$ be the ideal tree computed at initialization. Note that the goal state g is always part of $\mathcal{T}$, thus $\mathcal{T}$ never becomes empty and therefore reconnection search always succeeds. Because reconnection search is only invoked after a new inaccessible state is detected, it 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 a single 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

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

which maximizes when $|\mathcal{M}| = \frac{|S|-1}{2}$. Substituting such a value in (1), we obtain the desired result. Note that the algorithm follows a path of the tree $|\mathcal{M}|+1$ times because it reconnects $|\mathcal{M}|$ times. ■

The average complexity can be expected to be much lower. Indeed, the number of reconnection searches is at most the number of inaccessible states that can be reached by some state in G_M, which in many cases is much lower than the total number of obstacles.

The following intermediate result is necessary to prove that after termination, the agent knows a solution to the problem that is possible shorter than the one just found.

Lemma 1 *After every reconnection search, $s_0 \in \mathcal{T}$.*

Proof: The proof is by induction on the number of reconnection searches. At the start of search, the property holds by definition of spanning ideal tree. For the induction, let s denote the current state and suppose there is a path σ from s_0 to s in $\mathcal{T}$. Let σ' denote the path from s to g in $\mathcal{T}$ after reconnection. It is clear that σ and σ' contain at least one state in common, s. Let x be the first state in σ that appears also in σ'. Then, after reconnecting s with $\mathcal{T}$, the parent pointer of x would be reset in such a way that there will still be a path from s_0 to g in $\mathcal{T}$. ■

Theorem 2 *Running the algorithm for a second time over the same problem, without initializing 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: Straightforward from Lemma 1. ■

Note that Theorem 2 implies that our algorithm can return a different path in a second trial, which is an "optimized version" that does not contain the loops that the first version had. The second execution of the algorithm is naturally very fast because reconnection search is not required.

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 to two classes of search algorithms. For the first class, we considered state-of-the-art real-time heuristic search algorithms. Specifically, we compare to LSS-LRTA* (Koenig and Sun 2009), and wLSS-LRTA* (Rivera, Baier, and Hernández 2013), a variant of LSS-LRTA* that may outperform it significantly. For the second class, we compared to the incremental heuristic search algorithms Repeated A* and Adaptive A*. We chose them because it is easy to modify them to satisfy the real-time property following the same approach we follow with FRIT. We do not include D*Lite (Koenig and 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 et al. 2012). 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.

Repeated A* and Adaptive A* both run a complete A* until the goal is reached. Then the path found is followed until the goal is reached or until 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, and Sturtevant 2009). In each iteration, if the algorithm does not have a path to the goal (and hence it is running an A*) we only allow it to expand at most k states, and if no path to the goal is found the agent is not moved. Otherwise (the agent has a path to the goal) the agent makes a single move on the path.

For the case of FRIT, we satisfy the real-time property as discussed above by setting the m constant 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, 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 videogames 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.

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 and Lee 2006).

We used twelve maps from deployed video games to carry out the experiments. The first six are taken from the game *Dragon Age*, and the remaining six are taken from the game *StarCraft*. The maps were retrieved from Nathan Sturtevant's pathfinding repository (Sturtevant 2012).[2] We average our experimental results over 300 test cases with a reachable goal cell for each map. For each test case the start and goal cells are chosen randomly. All the experiments were run on a 2.00GHz QuadCore Intel Xeon machine running Linux.

As a parameter to FRIT we used the following alogorithms for reconnection:

- bfs: a standard breadth-first search algorithm.
- iddfs(n): a modified iterative deepening depth-first search which in iteration k runs a depth first search to depth kn. To save execution time, after each iteration, it stores the last layer of the tree so that the next iteration does not need to re-expand nodes.

Figure 2 shows average total iterations until the goal is reached versus time per planning episode. AA and rA represent Adaptive A* and Repeated A*. "bfs" and "dfs-n" represent FRIT using the algorithms described above. 1-LSS corresponds to LSS-LRTA*, and 16-LSS and 32-LSS correspond, respectively, to wLSS-LRTA* with $w = 16$ and $w = 32$. We use these values since Rivera, Baier, and Hernández (2013) report them as producing best results in game maps.

We observe that FRIT returns significantly better solutions when time constraints are very tight. Indeed, our algorithm does not need more than 45μ sec to return its best solution. Given such a time as a limit per episode, 32-LSS, the algorithm that comes closest requires between three and four times as many iterations on average. Furthermore, to obtain a solution of the quality returned by FRIT at 45μ sec, AA* needs around 150μ sec; i.e., slightly more than 3 times as long as FRIT. We observe that the benefit of iddfs is only marginal over bfs.

LSS-LRTA* and its variants are completely outperformed by FRIT as the solutions returned are much better in terms of quality for any given time deadline, and, moreover, the best solution returned by FRIT is 3 times cheaper than the best solution returned by the best variant of LSS-LRTA*.

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

[2]Maps used for 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 for StarCraft: ArcticStation, Enigma, Inferno JungleSiege, Ramparts and WheelofWar of size 768×768, 768×768, 768×768, 768×768, 512×512 and 768×768 cells respectively.

	FRIT(BFS)			RA*			AA*		
k	Avg. Its	Time/ep (μs)	No moves (%)	Avg. Its	Time/ep (μs)	No moves (%)	Avg. Its	Time/ep (μs)	No moves (%)
1	1717724	0.015	99.82	3449725	0.077	99.95	1090722	0.072	99.84
5	345316	0.076	99.13	690952	0.387	99.75	219162	0.357	99.22
10	173765	0.152	98.27	346105	0.774	99.50	110217	0.714	98.45
50	36524	0.746	91.77	70228	3.850	97.58	23061	3.512	92.61
100	19369	1.460	84.49	35744	7.651	95.25	12167	6.892	86.00
500	5665	6.176	46.98	8185	36.29	79.27	3482	29.57	51.12
1000	4069	9.905	26.17	4838	66.53	64.94	2513	46.74	32.26
5000	3064	15.97	1.978	2240	188.4	24.29	1822	79.69	6.607
10000	3017	16.45	0.435	1931	240.9	12.16	1748	86.23	2.632
50000	3004	16.58	0.012	1713	299.8	1.009	1703	90.68	0.100
100000	3004	16.58	0.003	1698	305.0	0.122	1702	90.85	0.007

Table 1: Relationship between search expansions and number of iterations in which the agent does not move. 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, and the percentage of iterations in which the agent was not moved by the algorithm with respect to the total number of iterations.

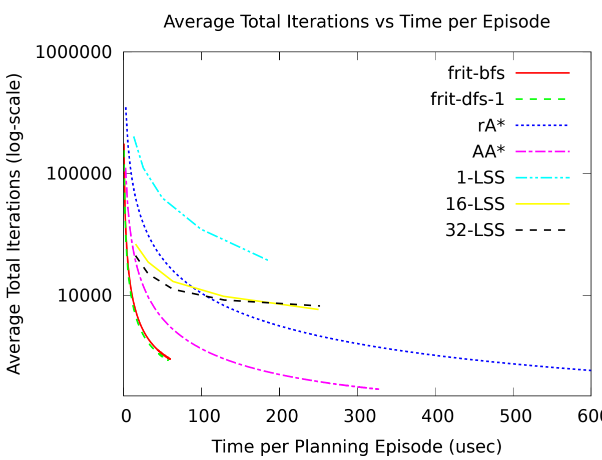

Figure 2: Total Iterations versus Time per Episode

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 this means approximately 16 μs, whereas for AA* and RA* this requires times of over 86 μs and 299 μs respectively. Additionally, Table 1 shows that for a given time constraint, FRIT behaves much better than both RA* and AA*, requiring fewer iterations and less time. Nevertheless, when provided more time, FRIT does not take advantage of it and the resulting solutions cease to improve. 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 60 steps, and the time used per search episode only increases by 1.65μs. 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.

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 and Likhachev 2002), Adaptive A* (Koenig and Likhachev 2005) and Tree Adaptive A* (Hernández et al. 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. Different to our algorithm, such searches return *optimal* paths connecting the current state with the goal. Our algorithm 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 and Sun 2009, Koenig and Likhachev 2006, Sturtevant and Bulitko 2011, Hernández and Baier 2012, Rivera, Baier, and Hernández 2013), 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 carrying any search at all (cf. Theorem 2). 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 quality, even though experimental results are reasonable.

HCDPS (Lawrence and 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). Recent Incremental Search algorithms such as Tree Adaptive A* exploits this idea to make subsequent searches faster. Real-Time D* (RTD*) (Bond et al. 2010) use 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.

Summary

We presented FRIT, a real-time 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. In our experiments, it returns solutions much faster than other state-of-the-art algorithms when time constraints are tight. We proved that our algorithm finds a solution in at most a quadratic number of states, however in grids it is able to find very good solutions, faster than all algorithms we compared to. In our experiments we evaluated two blind-search algorithms for reconnection: bfs and a version of iterative deepening depth-first search (iddfs). Their performance is very similar but bfs is much simpler to implement.

A disadvantage of our algorithm is that if given arbitrarily more time our algorithm cannot return a better solution, even if we use better blind search algorithms. As such, our algorithm is recommended over incremental search algorithms only under very tight time constraints per planning episode.

Acknowledgments

Nicolás Rivera, León Illanes, and Jorge Baier were partly funded by Fondecyt Project Number 11110321.

References

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

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

Bonet, B., and Geffner, H. 2011. Planning under partial observability by classical replanning: Theory and experiments. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 1936–1941.

Botea, A. 2011. Ultra-fast Optimal Pathfinding without Runtime Search. In *Proceedings of the 7th Annual International AIIDE Conference (AIIDE)*.

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

Bulitko, V.; Björnsson, Y.; Lustrek, M.; Schaeffer, J.; and Sigmundarson, S. 2007. Dynamic control in path-planning with real-time heuristic search. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, 49–56.

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

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

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

Hernández, C., and Baier, J. A. 2011. Fast subgoaling for pathfinding via real-time search. In *Proceedings of the 21th International Conference on Automated Planning and Scheduling (ICAPS)*.

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

Hernández, C.; Sun, X.; Koenig, S.; and Meseguer, P. 2011. Tree adaptive A*. In *Proceedings of the 10th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*.

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

Koenig, S., and Likhachev, M. 2002. D* lite. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, 476–483.

Koenig, S., and Likhachev, M. 2005. Adaptive A*. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, 1311–1312.

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

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

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

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

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

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

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

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

Sturtevant, N. R., and Bulitko, V. 2011. Learning where you are going and from whence you came: h- and g-cost learning in real-time heuristic search. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 365–370.

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

Taylor, K., and LaValle, S. M. 2009. I-bug: An intensity-based bug algorithm. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA)*, 3981–3986.

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