PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

RECONNECTION WITH THE IDEAL TREE: A NEW APPROACH TO REAL-TIME SEARCH

LEÓN ILLANES FONTAINE

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JORGE A. BAIER A.

Santiago de Chile, January 2014

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

RECONNECTION WITH THE IDEAL TREE: A NEW APPROACH TO REAL-TIME SEARCH

LEÓN ILLANES FONTAINE

Members of the Committee:
JORGE A. BAIER A.
JUAN L. REUTTER D.
CARLOS HERNÁNDEZ U.
ANDRÉS GUESALAGA M.

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, January 2014

To my family and friends.

ACKNOWLEDGEMENTS

Above all I want to acknowledge and thank my thesis advisor, Jorge Baier, and my office mate and research partner, Nicolás Rivera –whose ideas form the base over which this work is built. Without either of them, no part of the thesis would have ever existed.

Alongside them, I'd like to thank Carlos Hernández, whose insight and expertise helped guide the project. I also want to thank the other members of my committee, Andrés Guesalaga and Juan Reutter, for helping expediting the final processes involved.

In addition, I'd like to express my gratitude to all other staff members at the Department of Computer Science, and specially acknowledge Soledad Carrión for her endless help throughout every step of my career.

Of course, I wish to acknowledge my family and my friends, without whom all else is unimportant. Among them, I specifically thank the people at (and around) office O10: Andrés, Gabriel, Gonzalo, Martín and –once again– Nicolás.

Finally, I'd like to thank my soon-to-be-wife, Andreíta, for always supporting me and helping me, both in relation to this work and all other activities in my life.

Real stupidity beats artificial intelligence every time.

—Terry Pratchett, Hogfather (1996)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In this thesis we present FRIT, a simple approach for solving single-agent deterministic search problems under tight time constraints in partially known environments. Unlike traditional Real-Time Heuristic Search (RTHS) algorithms, FRIT does not search for the goal but rather searches for a path that connects the current state with a so-called *ideal tree* $\mathcal{T}$. Such a tree is rooted in the goal state and is built initially using a user-given 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 then carries out a reconnection search whose objective is to find a path between the current state and any node in $\mathcal{T}$.

Reconnection is done using an algorithm that is passed as a parameter to FRIT. As such, FRIT is a general framework that can be applied to many search algorithms. If such a parameter is an RTHS algorithm, then the resulting algorithm can be an RTHS algorithm. We show, however, that FRIT may be fed with a complete blind-search algorithm, which in some applications with tight time constraints (including video games) may be acceptable and, perhaps, preferred to a pure RTHS algorithm.

We evaluate over standard grid pathfinding benchmarks including game maps and mazes. 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 very good quality solutions and is perhaps the algorithm that should be preferred in video game applications.

Keywords: Heuristic Search, Real-Time Heuristic Search, Incremental Search, Heuristic Learning, A*, Learning Real-Time A*, FRIT

RESUMEN

En esta tesis se presenta FRIT, un algoritmo simple que resuelve problemas de búsqueda determinística para un agente, en ambientes parcialmente conocidos bajo restricciones de tiempo estrictas. A diferencia de otros algoritmos de búsqueda heurística en tiempo real (BHTR), FRIT no busca el objetivo: busca un camino que conecte el estado actual con un *árbol ideal* $\mathcal{T}$. El árbol tiene su raíz en el objetivo y se construye usando la heurística h. Si el agente observa que un arco en el árbol no existe en el ambiente real, lo saca de $\mathcal{T}$ y realiza una búsqueda de reconexión para encontrar un camino que lleve a cualquier estado en $\mathcal{T}$.

La búsqueda de reconexión se lleva a cabo por medio de otro algoritmo. Así, FRIT puede aplicarse sobre muchos algoritmos de búsqueda y si se trata de un algoritmo para BHTR, el algoritmo resultante puede serlo también. Por otro lado, mostramos que FRIT también puede usar un algoritmo de búsqueda ciega, resultando en un algoritmo que puede ser aceptable para aplicaciones con restricciones de tiempo estrictas (como videojuegos) y que incluso puede ser preferible a un algoritmo de BHTR.

Evaluamos el algoritmo en problemas estándares de búsqueda en grillas, incluyendo mapas de videojuegos y laberintos. Los resultados muestran que FRIT usado con RTAA*—un algoritmo de BHTR típico—es significativamente mejor que RTAA*, con mejoras de hasta un orden de magnitud bajo restricciones de tiempo estrictas. Además, FRIT(daRTAA*) supera a daRTAA*—el estado del arte en BHTR—y en promedio obtiene soluciones un 50% menos costosas usando el mismo tiempo total. Finalmente, FRIT usando búsqueda en amplitud obtiene soluciones de muy buena calidad y puede ser ideal para aplicaciones de videojuegos.

Palabras Claves: Búsqueda Heurística, Búsqueda en Tiempo Real, Búsqueda Incremental, Aprendizaje de Heurísticas, A*, FRIT

1. INTRODUCTION

1.1. Background

Many algorithmic problems in Computer Science are formulated as search tasks, in which the goal is to find a solution for the original problem. Indeed, search algorithms are one of the core tools used in the development of Artificial Intelligence, where so called *rational agents* act based on their observation of the environment and attempt to achieve positive outcomes (Russell & Norvig, 2010, Ch. 1). Many of the problems solved by such agents are naturally modeled as search problems. Some examples are pathfinding for field robotics and general planning of action sequences.

Generally, many other problems can be formalized as *state-space problems*. This formulation defines a space of states and operations that describe the environment, where the operations modify the environment and change the state. In this setting, we aim to find a set of operations that can modify an initial state into a goal state. This can be modeled as a graph, where states are nodes and operations are the edges that connect different nodes. Here, a solution is a sequence of operations forming a path from the initial state to the goal state. This path can be optimized for different criteria, such as length or cost.

As an example, we can see how this approach can be used to model puzzle problems such as the Rubik's Cube. A $3 \times 3 \times 3$ cube is formed by 26 visible smaller cubes, called cubies. Of these, 6 represent the centers of each face and cannot be moved. The remaining 20 cubies can be either one of the 8 corners or one of the 12 edges. Each of the corner cubies has 3 colors, whereas the edges have 2. This uniquely identifies each cubie, and can be therefore used to identify the state of the full cube as a list describing the position of each of the 20 movable cubies with respect to the static frame of reference defined by the 6 unmovable ones. There is a set of 12 (or 18) primitive or fundamental operations that can be performed on every state. These correspond to rotating any of the 6 faces 90° in

either direction (traditionally, rotating 180° is considered a separate action, although it could be represented as two 90° rotations in the same direction). This way, a graph representation for this problem has a node for each state, and each node is connected to 12 (or 18) other nodes through the corresponding operators. Note that even considering some other restrictions regarding the respective positions of the fixed center cubies, such a representation has over 4.3×10^{19} states (Korf, 1997). However, it has been computationally proved that for every state in the graph resulting from considering 18 operators, there exists at least one path to the goal of length shorter or equal to 20 (Rokicki, Kociemba, Davidson, & Dethridge, 2013).

The general problem of finding optimal paths in graphs has been studied extensively both in Mathematics and Computer Science, and many algorithms have been designed specifically for this. Algorithms employing *uninformed* search strategies make no assumptions on the structure and characteristics of the problem, and work well in the general case. However, they can be inefficient when dealing with very large state spaces, such as the ones found in many real-world applications. *Informed* search strategies aim to solve this issue by means of *heuristic functions* that try to guide the search efforts towards the goal, avoiding unnecessary computation. The heuristics are designed specifically for each application, and often attempt to imitate techniques used by humans when dealing with similar tasks. The area of Artificial Intelligence concerned with heuristics and informed search is *Heuristic Search*.

For some applications, additional constraints and limitations are placed upon the formalization of the problem. For instance, the information available for a robot moving in an unknown environment can be limited to what the robot has already observed, or the amount of computation allowable before moving a character in a real-time video game is limited. This thesis is concerned with problems that combine both of this constraints, in what we call Real-Time Heuristic Search in unknown environments.

1.1.1. State-Space Problems

As discussed above, many problems in Artificial Intelligence and Computer Science can be formulated as state-space search problems, and can subsequently be solved with search techniques. Below, we formalize state-space search problems.

A state-space search problem is defined as a tuple $P = (S, A, c, s_{start}, G)$. The directed graph (S, A) represents the state-space, where S is the set of all possible states and A is the set of operations or actions that can transition the problem through different states. This way, if $a = (s, t)$ and $a \in A$, then it is possible to transition from state s to state t by executing action a. $s_{start} \in S$ corresponds to the initial state and $G \subseteq S$ is the set of all goal states. The function $c : A \to \mathbb{R}_0^+$ assigns costs to the actions, so that an agent performing action a will incur in a cost of $c(a)$.

Note that without loss of generality, we can assume one single goal state g, such that $G = \{g\}$. To use this formulation with a problem that does have multiple goal states we can add a new state, g, and connect every state $s \in G$ to g with an action $a = (s, g)$ such that $c(a) = 0$. Below, we mostly use this formulation.

A sequence of states $\sigma = s_0, s_1, \ldots, s_n$ such that for every $i < n$, we have $(s_i, s_{i+1}) \in A$ is called a path. Any path starting in s_{start} and finishing in g is a solution for P. An optimal solution is one that minimizes the sum of the costs of the actions used. Depending on the application, the goal can be to find an optimal path or to quickly find a suboptimal path. The standard algorithm for finding an optimal path is Dijkstra's Algorithm. Pseudo-code for the algorithm is shown in Algorithm 1.

In very general terms, this algorithm extends the search outwards from s_{start} until it reaches g. More specifically, it searches the states of the problem in order of cumulative distance from s_{start}, ensuring each node is reached through an optimal path. Using an appropriate data structure for the priority queue Q, the algorithm can run in $O(|A| + |S| \log |S|)$ steps (Fredman & Tarjan, 1984).

Algorithm 1: Dijkstra's Algorithm

Input: S, A, c, s_{start}, G
Output: A sequence of states representing the shortest path from s_{start} to a state in G.

Initialization:
for each $x \in S$ **do**
    $x.distance \leftarrow \infty$
    $x.expanded \leftarrow False$
    $x.previous \leftarrow null$
$s_{start}.distance \leftarrow 0$
Insert s_{start} into priority queue Q.
Search:
while Q *is not empty* **do**
    Remove s from Q with smallest *distance*.
    if $s \in G$ **then**
        return `// The solution can be extracted from the` *previous* `pointers.`
    $s.expanded \leftarrow True$
    for each $n \in S$ *such that* $(s, n) \in A$ **do**
        $d \leftarrow s.distance + c(s, n)$
        if $d < n.distance$ **then**
            $n.distance \leftarrow d$
            $n.previous \leftarrow s$
            if $\neg n.expanded$ **then**
                Insert n into Q.

1.1.2. Heuristic Search

For some real-world applications where the number of states is large, Dijkstra's Algorithm is somewhat inefficient. To solve these issue, we can guide the search by using information specific to the problems involved. Usually, we include this information into the search as a *heuristic function*, a function that somehow ranks the various available states.

In our formal setting, a heuristic is defined as a function $h : S \to \mathbb{R}_0^+$, that assigns to each state $s \in S$ an estimated value of the cost needed to go from s to g through a path in the state-space graph. We define the perfect or optimal heuristic

h^* as the heuristic that correctly estimates the minimum distances for each state. This way, for any $s \in S$, $h^*(s)$ corresponds to the cost of the best path between s and g. Additionally, we say that a heuristic h is *admissible* if for every state $s \in S$ it holds that $h(s) \leq h^*(s)$. That is, a heuristic is admissible if it never overestimates the distances. Finally, we say that h is *consistent* if for every $(s,t) \in A$ it holds that $h(s) \leq c(s,t) + h(t)$, and for every $g \in G$ it holds that $h(g) = 0$. It is easy to prove that all consistent heuristics are also admissible.

1.1.2.1. The A* Algorithm

The standard algorithm used for finding shortest paths in Heuristic Search is called A* (Hart, Nilsson, & Raphael, 1968). It works by maintaining and updating a *merit function* f that estimates, for each state s, the cost of an optimal path going from s_{start} to g and passing through s. For a given state $s \in S$, it is defined as

$$f(s) = g(s) + h(s),$$

where $g(s)$ corresponds to the accumulated cost of the best path already discovered that goes from s_{start} to s. Initially, $g(s_{start}) = 0$ and $g(s) = \infty$ for all other states. Pseudo-code for this algorithm is shown in Algorithm 2.

A* uses two lists, *Open* and *Closed*, which intuitively represent the information known for each state. As states are discovered, they are put in the *Open* list. When the state is expanded (i.e.: all its neighbors are discovered) it is moved to the *Closed* list. Whenever a shorter path to a state in *Closed* is discovered, the state is moved back to *Open*, to eventually check if this implies better paths for any of it neighbors. Note that the order in which the states are explored and expanded depends on the merit function f, which is determined by both the heuristic function h and the costs of the discovered paths. This contrasts with Dijkstra's Algorithm, where the order of expansion is determined exclusively by the costs. Indeed, if $h(s) = 0$ for all states $s \in S$, and assuming tie-breaking for states with the same merit is done in the same

Algorithm 2: The A* Algorithm

Input: $S, A, c, s_{start}, G, h(\cdot)$
Output: A sequence of states representing the shortest path from s_{start} to a state in G.

Initialization:
$Closed \leftarrow \varnothing$
$Open \leftarrow \{s_{start}\}$
for each $s \in S$ **do**
    $g(s) \leftarrow \infty$
    $s.previous = null$
$g(s_{start}) \leftarrow 0$
$f(s_{start}) \leftarrow h(s_{start})$
Search:
while *Open is not empty* **do**
    Remove s from *Open* with minimum $f(s)$.
    Insert s into *Closed*.
    if $s \in G$ **then**
        return `// The solution can be extracted from the` *previous* `pointers.`
    for each $n \in S$ *such that* $(s, n) \in A$ **do**
        if $g(s) + c(s, n) < g(n)$ **then**
            $n.previous \leftarrow s$
            $g(n) \leftarrow g(s) + c(s, n)$
            $f(n) \leftarrow g(n) + h(n)$
            if $n \in Closed$ **then**
                Remove n from *Closed*.
        if $n \notin Open$ *and* $n \notin Closed$ **then**
            Insert n into *Open*.

way, then A* expands the states in the same order as Dijkstra's Algorithm and gives identical results.

Other relevant properties of A* are that if h is admissible, it will return an optimal solution (Hart et al., 1968). Moreover, if h is consistent then A* is *optimally efficient* and no algorithm can be shown to expand fewer states than A* when using the same heuristic (Edelkamp & Schrödl, 2011) and tie-breaking strategy. Furthermore, given two different consistent heuristics h_1 and h_2 such that $h_2(s) \geq h_1(s)$ for

all states $s \in S$, A* using h_2 will expand fewer states than A* using h_1. Intuitively, this means that the performance of A* improves when the heuristic is a better approximation. This motivates the concept of *heuristic learning*, which has been a key tool for real-time heuristic search and is further discussed in the following sections.

Additionally, A* can be easily used to find suboptimal paths by using *weights* that modify an admissible heuristic, making it inadmissible. Typically, this is done by redefining the merit function f to $f(s) = g(s) + w \cdot h(s)$. This is known as the Weighted A* Algorithm (wA*), and—when compared to simply using A* with the admissible heuristic—will usually result in a much faster performance time wise, albeit at a loss in solution quality (i.e.: cost). When using an admissible heuristic and a weight w, the obtained solution is suboptimal by at most a factor of w.

1.1.3. Incremental Heuristic Search

An alternative method for speeding up searches is to use *Incremental Search*. This technique is used when searches are done repeatedly in the same or similar environment, so algorithms attempt to reuse information obtained in previous efforts in order to solve newer problems faster than by doing a completely new search. It is particularly relevant when dealing with dynamic environments—such as those encountered by most applications involving the real, physical world or by applications involving multiple independent agents. Here, planning needs to be repeated as the state-space graph changes. Many uninformed algorithms focused on finding optimal paths on these graphs have been proposed in literature. Indeed, Deo and Pang (1984) refer to several such algorithms published in the 1960s.

Since then, algorithms that combine Incremental Search with Heuristic Search have been developed. Some examples are D* (Stentz, 1995), D*Lite (Koenig & Likhachev, 2002), Lifelong Planning A* (Koenig & Likhachev, 2001; Koenig, Likhachev, & Furcy, 2004) and Adaptive A* (Koenig & Likhachev, 2006a). These algorithms can be used to solve search problems in initially unknown environments, which are frequently seen in the field of Robotics. A robot moving in an unknown terrain which

knows its relative position with regards to a goal can plan a path to it by assuming all unknown areas to be traversable. This is known as planning with the *free-space assumption*. When traveling through the resulting path, the robot might encounter obstacles and will need to replan.

1.1.3.1. Adaptive A*

As mentioned above, Adaptive A* (AA*) is an incremental search algorithm that guides the search with the use of a heuristic function. It is based on A*, and as such has certain guarantees on optimality. Effectively, if the given heuristic is consistent, then at every point of the search the planned path-to-go is optimal with regards to the currently known information. Algorithm 3 shows pseudo-code for an implementation of AA* designed for use in pathfinding in initially partially known terrain.

Algorithm 3: Adaptive A*

Input: $S, A, c, s_{start}, G, h(\cdot)$
Effect: The agent is moved from s_{start} to a state in G, if such a path exists.

1 **Initialization:**
2 Observe the environment around s_{start} and remove non-traversable arcs from A.
3 $s \leftarrow s_{start}$
4 **Search:**
5 **while** $s \notin G$ **do**
6 Call A*$(S, A, c, s, G, h(\cdot))$. Extract a solution path σ and keep the *Open* and *Closed* lists.
7 $f^* \leftarrow \min_{x \in Open} f(x)$
8 **for each** $x \in Closed$ **do**
9 $h(x) \leftarrow f^* - g(x)$
10 Move the agent through the path σ. While moving, observe the environment and remove non-traversable arcs from A. Stop if an arc in σ is removed from A.

The algorithm works by repeatedly searching with A*, and uses the returned path and the *Open* and *Closed* lists (Line 6). The agent is moved through the path and the state-space graph is updated according to observation of the agent's

environment (Line 10). Whenever an arc in the path is removed, the movement stops and the path is replanned. Finally, note that Lines 7–9 modify the heuristic function. This is known as *heuristic learning* and is used here to ensure that information discovered in a planning episode of the algorithm can be used in further planning stages. Heuristic learning is a key idea in real-time heuristic search, and will be discussed in more detail below.

1.1.4. Real-Time Heuristic Search

Certain search applications impose restrictions on the amount of time that can pass between the execution of two consecutive actions. Effectively, this limits the amount of computation that can be performed by an agent before deciding on a path to follow.

One of the original goals for the field was to limit the search to only consider states in a vicinity of the agent (Korf, 1990). This idea is sometimes called *agent-centered search* (Koenig, 2001). The underlying motivation for this is that it represents an efficient way of handling problems in very large space-states.

For this thesis, we are mostly concerned with the application of real-time heuristic search algorithms in a priori unknown environments. One of the most straightforward applications of this is pathfinding in video games, where characters must often move automatically in a partially known map. For interactivity reasons, systems are sometimes given only a few milliseconds to decide where to move a number of different characters (Bulitko, Björnsson, Sturtevant, & Lawrence, 2011). A different application considers very fast robots, which physically move in real-time.

1.1.4.1. Learning Real-Time A*

Most real-time heuristic search algorithms are variants of one of the original algorithms proposed by Korf (1990), Learning Real-Time A* (LRTA*). As such, most of these algorithms share a familiar structure based around four distinct parts: observing, planning, learning, and moving.

As for Adaptive A*, the learning process corresponds to heuristic learning. In general terms, the goal of learning is to update the heuristic to consider some of the information discovered during search.

In Algorithm 4 we show the pseudo-code for an implementation of LRTA*.

Algorithm 4: Learning Real-Time A*

Input: $S, A, c, s_{start}, G, h(\cdot)$
Effect: The agent is moved from s_{start} to a state in G, if such a path exists.

```
1 Initialization:
2 s ← s_start
3 Search:
4 while s ∉ G do
5     Observe the environment around the location of the agent and remove
      from A all newly discovered non-traversable edges.
6     n ← arg min_{t:(s,t)∈A} c(s,t) + h(t)
7     if h(s) < c(s,n) + h(n) then
8         h(s) ← c(s,n) + h(n)
9     Move the agent to n, setting s ← n.
```

Note that the four concepts defined are clearly represented in the algorithm. Indeed, in Line 5 the agent observes the environment, updating the relevant information. Then, in Line 6, the agent plans which of its direct neighbors it will move to. Lines 7 and 8 define how the heuristic is updated, which corresponds to the learning stage. Finally, Line 9 moves the agent according to the plan.

The learning process is focused on inflating the heuristic values of visited states as much as possible without making the heuristic inconsistent. Therefore, the update sets the heuristic of a state s to the highest possible value that does not break consistency, which is $\min_{t:(s,t)\in A} c(s,t) + h(t)$.

Variants of LRTA* algorithm modify the different stages, usually performing more elaborate computations in some of them. For example, both RTAA* (Koenig & Likhachev, 2006b) and LSS-LRTA* (Koenig & Sun, 2009) modify the planning and learning stages, allowing for a longer path planned and more heuristic values updated in each iteration. Both of these algorithms use—as their planning stage—an

A* search that stops whenever a predetermined number of states has been expanded. They differ in how they perform the heuristic update. LSS-LRTA* extends the idea used in LRTA*, inflating the heuristic as much as possible in a larger region. RTAA* uses the same the update strategy as AA*, which results in a faster procedure where the states' heuristic values are not raised as much.

1.1.4.2. Convergence

One of the key properties of LRTA* is that repeatedly running the algorithm over the same problem will continue to increase the heuristic values of the visited nodes until they converge to optimal values. That is, running sufficient trials over the same problem without reseting the heuristic function will inflate it for states in the visited paths until it reaches h^*. Eventually, all states in the optimal path between s_{start} and G will have accurate heuristic values and running the algorithm again will result in the optimal solution.

Many real-time search algorithms share this property, which we call *optimal convergence*. Additionally, some algorithms converge to suboptimal solutions. The number of repeated trials needed for convergence varies between algorithms, problem domains and problem instances.

1.1.4.3. Heuristic Depressions

Among the most important issues involved is real-time search, is the fact that heuristics often contain *depressions*. Intuitively, depressions are bounded regions of the state-space that do not contain a goal state and where the heuristic values are comparatively low with respect to the heuristic values of states just outside the region. It is well known that real-time search algorithms can perform poorly in these regions. Indeed, Korf (1990) provides the example shown in Figure 1.1. Here, if an agent based on LRTA* starts in one of the two nodes of the graph that have heuristic value 1, it will move back and forth between them increasing their heuristic values in small steps until at least one of them reaches a heuristic value of 5.

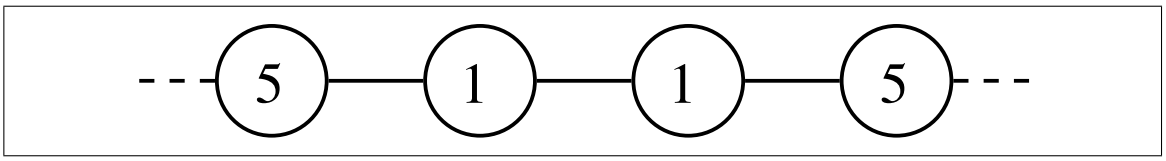


FIGURE 1.1. A heuristic depression in a graph. The numbers represent the heuristic values for each state.

Figure 1.2 shows an example of how a heuristic depression can naturally occur in simple pathfinding problems in grids using a common heuristic function. This heuristic is known as the Manhattan Heuristic, and it estimates the cost to the goal (marked as G) as the sum of the horizontal and vertical distances.

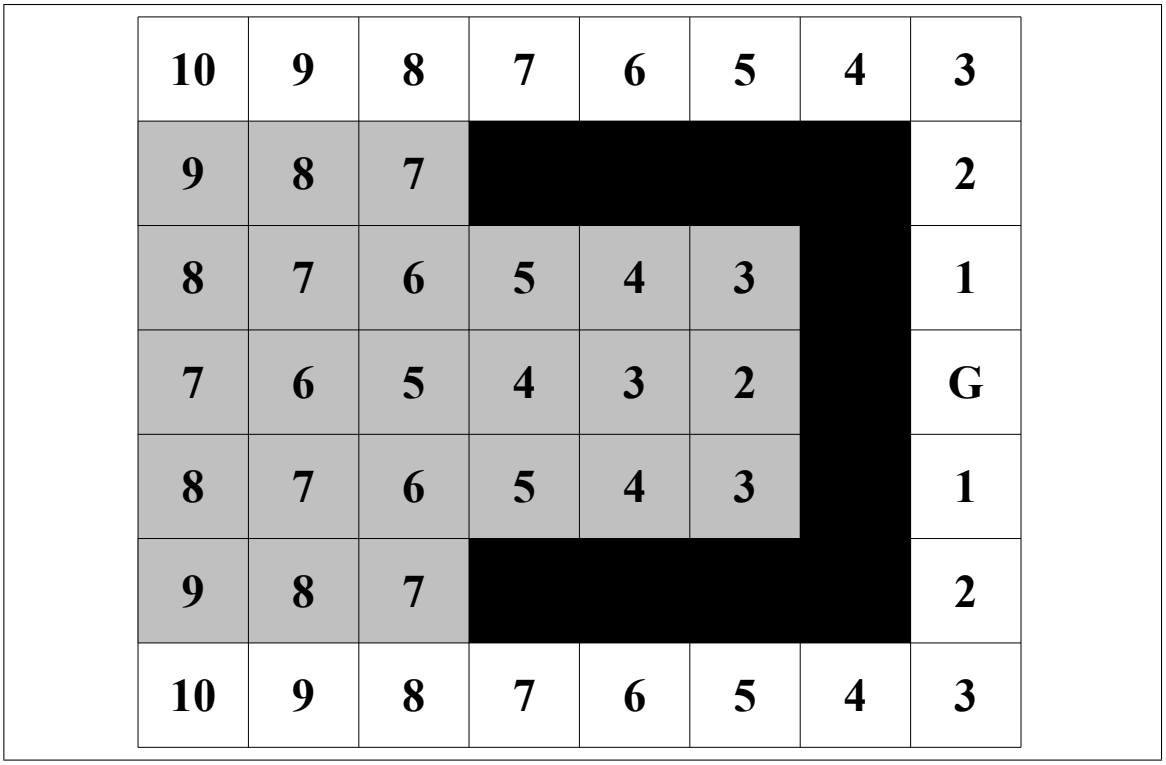


FIGURE 1.2. A heuristic depression in a grid pathfinding problem. The numbers represent the heuristic value for each cell of the grid. Black cells are obstacles. The gray colored cells are a heuristic depression.

Algorithms such as LSS-LRTA* are not very effective when dealing with heuristic depressions. Although learning the heuristic correctly in a big enough are will ensure that the depressions are mostly avoided, this approach will not work when the relevant regions are larger that what is feasible to learn in limited time. More elaborate

approaches consider removing or pruning states that are proved to be not conducive towards the goal (Sharon, Sturtevant, & Felner, 2013). A different family of algorithms attempts to actively identify depressions and escape them quickly (Hernández & Baier, 2012).

1.2. Thesis Work

The goal of the research project described here is to apply a new approach to real-time search. Although the approach shares similarities with both Incremental Search and traditional Real-Time Heuristic Search, we believe the core idea behind it has not been previously considered in literature. To understand the motivation behind this idea, we can somewhat reformulate search problems in initially unknown environments.

As we have discussed in the previous sections, the *planning* stage of algorithms for these problems is repeated when previous plans are discovered to be incorrect. Usually, each planning stage is treated as a new search problem in which the objective is to find a path to a goal state. We propose that the objective could rather be to find a path to an area where we believe finding a path to the goal will be easy. Basically, the agent should move to regions in which the heuristic seems to be correct. This is similar to what is described by Hernández and Baier (2012), but our approach is very different in spirit.

1.2.1. Major Contributions

The major contributions presented in this thesis are outlined below.

- We describe FRIT, a new family of search algorithms for initially partially known environments.
- We show two different ways to use FRIT with real-time constraints.
- We prove that both versions of our algorithm terminate, and we give explicit bounds on the returned solutions.

- We prove that both algorithms converge after a second trial run.
- We empirically show that our algorithms perform well. Tests were run on standard grid pathfinding benchmarks (Sturtevant, 2012), where FRIT significantly outperforms state-of-the-art RTHS algorithms.

1.2.2. Future Work

Future work in this area can be based upon applying the proposed algorithms to other problems where real-time heuristic search is relevant. Some of these problems void some of the key assumptions made by algorithms such as LRTA*. For example, the number of states in the vicinity of an agent is usually assumed to be constantly bounded, which is not true in a dense graph. We believe our algorithms may be well suited for some of these situations.

2. ARTICLE SUBMITTED TO JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH

2.1. 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 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 they provide the main loop with quick moves that allow implementing continuous character moves.

Standard Real-Time Heuristic Search algorithms—e.g., LRTA*(Korf, 1990) or LSS-LRTA* (Koenig & Sun, 2009)—however, are not algorithms of choice for videogame developers, since they will require 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 algorithms based on 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 generate 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 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 the ideal tree, and it can continue following a branch.

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 fixed. 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 et al., 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 paper have been published in conference papers (Rivera, Illanes, Baier, & Hernandez, 2013). This articles extends the work and includes new material that has not been presented before. 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 2.3.
- We extend some of the previous empirical results by including maze benchmarks, which had not been previously considered, and by evaluating on more problem instances.

The rest of the paper is organized as follows. In Section 2.2 we describe the background necessary for the rest of the paper. In Section 2.3 we describe a simple version of our algorithm that is not real-time. In Section 2.4 we describe two alternative ways to make the algorithm satisfy the real-time property. In Section 2.5 we present a theoretical analysis, followed by a description of our experimental evaluation in Section 2.6. We then describe other related work, and finish with a summary.

2.2. 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 digraph 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 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 between s and 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.

2.2.1. 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 et al., 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 there is sufficient information to the opposite.

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 5, 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 the path identified previously by lookahead search. 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.

RTAA* (Koenig & Likhachev, 2006b) is an instance of Algorithm 5. 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

Algorithm 5: 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 the path. While moving, observe the
      environment and update G_M, removing any non traversable arcs. Stop if
      an arc in path is removed.
```

using Algorithm 6, which resets the heuristic of all states expanded by the lookahead according to the f-value of the best state in *Open*.

Algorithm 6: RTAA*'s heuristic learning.

```
1 procedure Update ()
2     f* ← min_{s∈Open} g(s) + h(s)
3     for each s ∈ Closed do
4         h(s) ← f* − g(s)
```

LRTA* (Korf, 1990) is also instance of Algorithm 5; indeed, LRTA* is an instance of RTAA* when the k parameter is set to 1. In a nutshell, 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).

2.3. 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.

2.3.1. 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:*

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

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 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 $\mathrm{p}(s)$ 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.

2.3.2. Following and Reconnecting

Our search algorithm, Follow and Reconnect with the Ideal Tree (FRIT, Algorithm 7) 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.

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 $\mathcal{T}$

Algorithm 7: 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 ∞.
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** $\mathrm{p}(s) = null$ **then**
12 $c \leftarrow c + 1$
13 RECONNECT $(\mathcal{A}, s, G_M, \text{INTREE}[c](\cdot))$.
14 **Movement:** Move the agent from s to $\mathrm{p}(s)$ and set s to the new position of the agent.

Algorithm 8: 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 σ 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 $\mathrm{p}(s_i) = s_{i+1}$ for every $i \in \{0, \ldots, n-1\}$.

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 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 8. 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 7 finishes when the agent reaches the goal.

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 9—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 all observed obstacles 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 9: InTree$[c]$ function

```
   Input: a vertex s
1  while s ≠ g do
2      if h(s) < h_obstacle then
3          return true
4      Paint s with color c.
5      if p(s) = null or p(s) has color c then
6          return false
7      s ← p(s)
8  return true
```

Figure 2.1 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, 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 before. A closer look shows that some times the agent does not walk exactly besides the wall but moves very close to them 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, Illanes, et al., 2013), and the overall improvements were shown to be small. For this paper, we use standard BFS. See Section 2.7.2 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 the h to exit the heuristic depressions generated by obstacles. In such a process they may need to revisit the same cell several times.

2.4. Satisfying the Real-Time Property

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

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

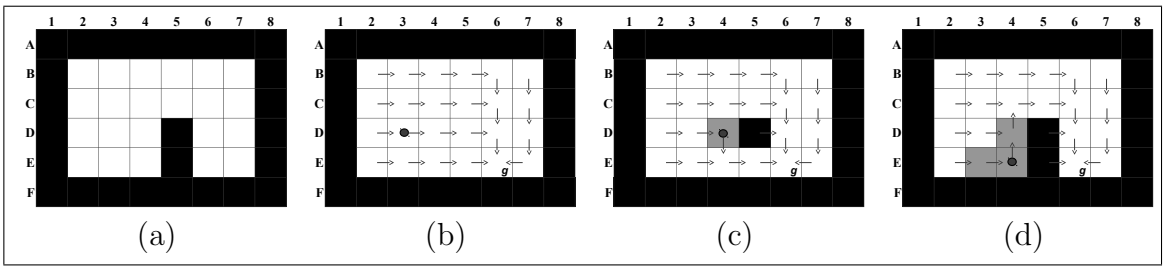


FIGURE 2.1. An illustration of some of the steps of an execution over a 4-connected grid pathfinding task, where the initial state is cell D3, and the goal is E6. The search algorithm $\mathcal{A}$ is breadth-first search, which, when expanding a cell, generates the successors in clockwise order starting 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.

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.

2.4.1. 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 5 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 5 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 10 shows the pseudocode for the modified generic real-time heuristic search algorithm, which has two main differences with respect to Algorithm 5. 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 10 connects the states in the path found by the lookahead search to $\mathcal{T}$. This implies also that the RECONNECT procedure described in Algorithm 8 needs to be changed by that described in Algorithm 11.

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 subset of states in S that are connected to g via arcs in $\mathcal{T}$. Now we are ready to define reconnecting heuristics formally.

Algorithm 10: 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 $path \leftarrow$ `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 11: 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 $f_{\mathrm{GOAL}}(\cdot)$

1 Call $\mathcal{A}(s, G_M, f_{\mathrm{GOAL}})$.

DEFINITION 2 (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 a* 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 10 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 10 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 the 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 $\sigma = s_0 s_1 \dots s_n$, with $s_0 = s$ and $s_n = s$,

[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.

be a shortest path between s and s'. Since h is consistent, it holds that

$$h(s_i) \leq c(s_i, s_{i+1}) + h(s_{i+1}), \tag{2.1}$$

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

$$h(s) - h(s') = \sum_{i=0}^{n-1} h(s_i) - h(s_{i+1}) \leq \sum_{i=0}^{n-1} c(s_i, s_{i+1}) = d(s, s') \tag{2.2}$$

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 Inequation 2.2, proves the desired result. □

2.4.1.1. 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 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.

2.4.1.2. Making InTree[c] Real-Time

Above 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 10 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).

2.4.2. 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 and sometimes require to re-visit 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 2.4.1.2, 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.

2.5. 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 algorithm 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.

2.5.1. Proofs for InTree[c]

To determine whether or not a state s belongs to the ideal tree, our InTree[c] function (Algorithm 9) 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 9, 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 2.3 and the color c is set to increment for each reconnection search,* InTree[c]*, as described in Algorithm 9, 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 as set through the construction of $\mathcal{T}$ described in Section 2.3. Note that all the paths in the initial Ideal Tree are monotonous; 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 at all. □

2.5.2. Termination and bound for $\mathbf{FRIT_{RT}}$

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 10, such that for any problem with x states guarantees termination in at most $f_A(x)$ steps, and such that it never updates the h-value of the goal state. Then $FRIT_{RT}$ (A) solves the problem in $\mathcal{O}(|S| f_A(|S|))$ steps.*

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_{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 number of total inaccessible states.

2.5.3. 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 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

$$(|\mathcal{M}| + 1)(|S| - |\mathcal{M}|), \tag{2.3}$$

which maximizes when $|\mathcal{M}| = \frac{|S|-1}{2}$. Substituting such a value in (2.3), gives the desired result. □

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

2.5.4. 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 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 have 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.

2.6. 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 two 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 RTAA* (Koenig & Likhachev, 2006b) and daRTAA* (Hernández & Baier, 2012), a variant of RTAA* that may outperform it significantly. 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.

Repeated A* and Adaptive A* both run a complete A* search 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, & 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(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.

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]

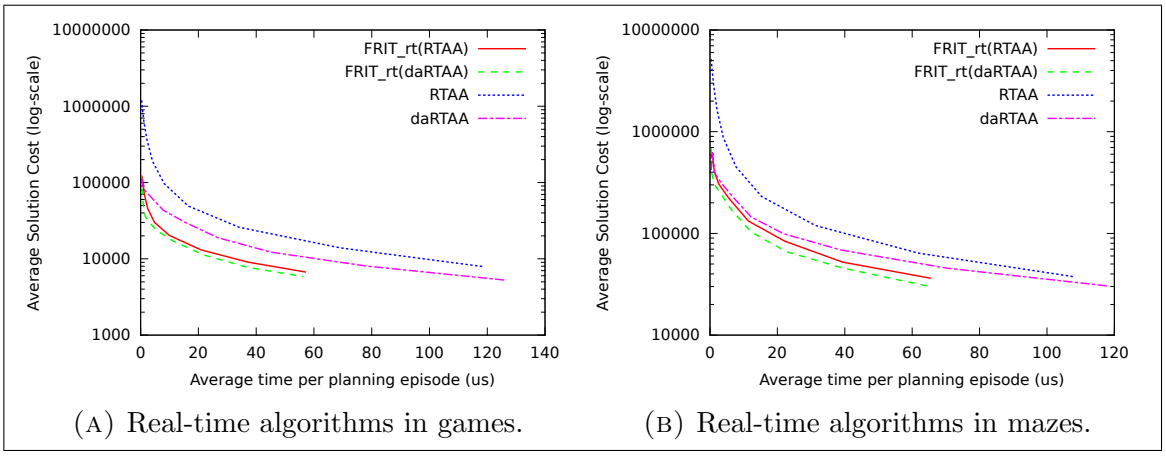
We averaged our experimental results over 500 test cases with a reachable goal cell for each map. For each test case the start and goal cells were chosen randomly. All real-time algorithms were run with 10 different parameter values. The incremental algorithms were run to completion once per test case,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

2.6.1. Analysis of the results for real-time search algorithms

Figure 2.2 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*), where solutions for a given time deadline are at least three times cheaper than pure RTAA* and up to one order of magnitude

[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) Real-time algorithms in games.

(B) Real-time algorithms in mazes.

FIGURE 2.2. Real-time algorithms: Total Iterations versus Time per Episode

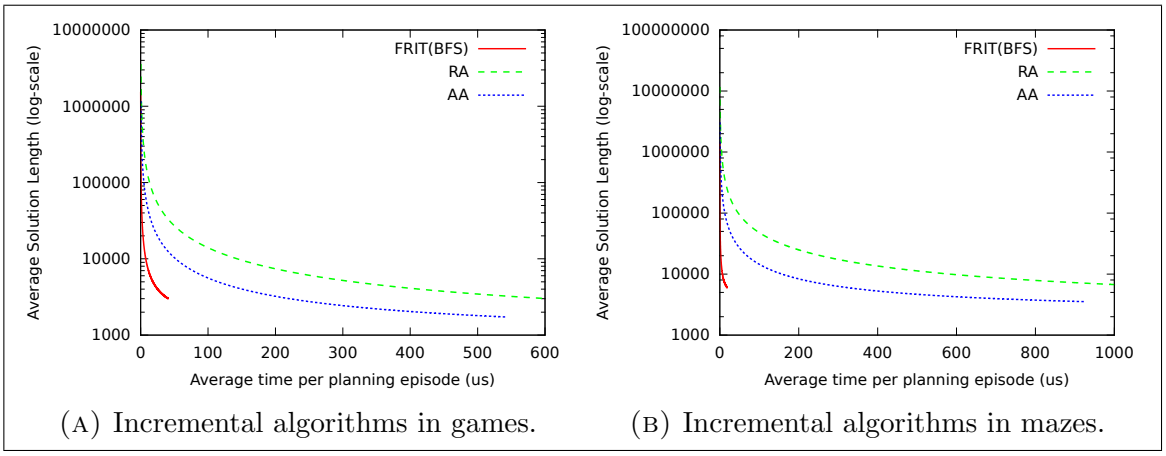
cheaper for very small time frames. It is interesting to note that even though daRTAA* improves significantly over RTAA*, FRIT_{RT}(daRTAA*) is only marginally better than FRIT_{RT}(RTAA*).

For mazes, the FRIT_{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 FRIT_{RT}(daRTAA*) are of comparable lengths, but FRIT_{RT}(daRTAA*) finds these solutions requiring slightly more than half of the time per planning episode than daRTAA*.

2.6.2. Analysis of the results for incremental algorithms modified to satisfy the real-time property

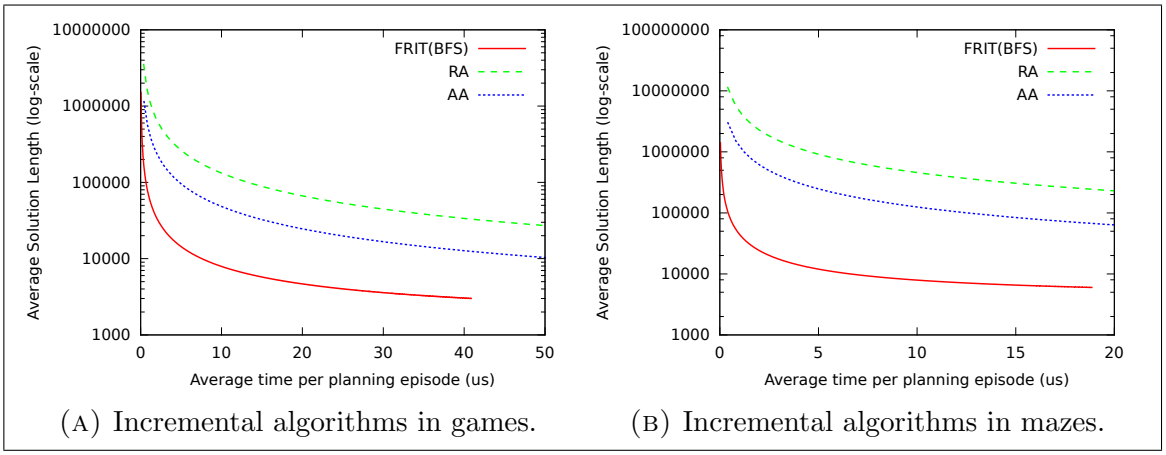
Figure 2.3 shows two plots of the average number of agent steps versus average time per planning episode for the incremental search algorithms used as real-time algorithms as described aboved in games and mazes benchmarks. Figure 2.4 shows the regions of the same plots as Figure 2.3 in which FRIT(BFS) 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



(A) Incremental algorithms in games. (B) Incremental algorithms in mazes.

FIGURE 2.3. Incremental algorithms: Total Iterations versus Time per Episode



(A) Incremental algorithms in games. (B) Incremental algorithms in mazes.

FIGURE 2.4. Incremental algorithms: Total Iterations versus Time per Episode (zoomed)

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 behaviour 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

	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 2.1. 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 was not moved by the algorithm with respect to the total number of iterations (No moves).

can be seen both in Figure 2.3, across both sets of benchmarks, and Table 2.1. 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 2.1, 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.

2.6.3. Comparison of the two approaches

Figure 2.5 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 1 no-move every $40,000$ moves.

2.7. Related Work

There are two bodies of work that are related to our algorithm: real-time and incremental heuristic search algorithms, and the *bug* family of pathfinding algorithms. We discuss both of them below.

2.7.1. Incremental and Real-Time Heuristic Search Algorithms

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),

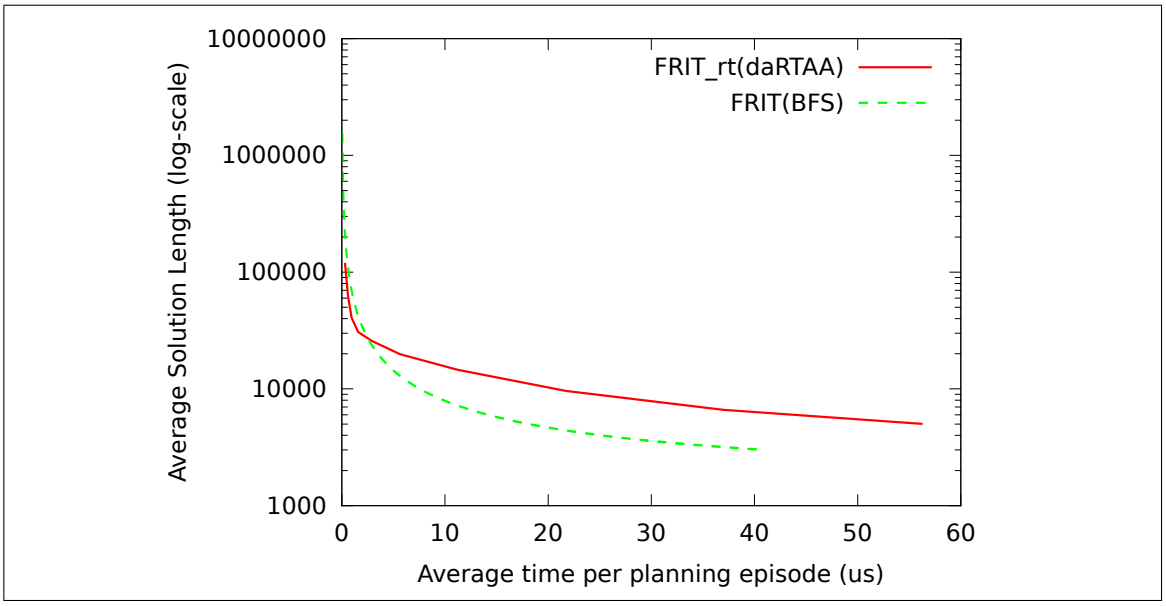


FIGURE 2.5. Comparison of FRIT using a real-time algorithm versus FRIT as an incremental algorithm in games benchmarks.

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. Different 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, 2006b; Sturtevant & Bulitko, 2011; Hernández & Baier,

2012; Rivera, Baier, & 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 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 quality, even though experimental results are reasonable.

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). Recent Incremental Search algorithms such as Tree Adaptive A* 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.

2.7.2. Bug Algorithms

Above we observed that in pathfinding applications, FRIT tends to follow walls, which is precisely what a family of algorithms called *bug algorithms* (LaValle, 2006;

Taylor & LaValle, 2009) do. Bug algorithms are a family of algorithms for pathfinding in 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). There are several strategies for implementing bug algorithms (Lumelsky & Stepanov, 1987; Sankaranarayanan & Vidyasagar, 1990; Kamon & Rivlin, 1997; Horiuchi & Noborio, 2001; Magid & Rivlin, 2004; Gabriely & Rimon, 2008). The relative performance of these algorithms vary significantly depending on the environment (Ng & Bräunl, 2007).

Some bug algorithms navigate towards the goal state following a line (mline) which is the shortest trajectory between the initial state and the goal state under the assumption that the terrain is obstacle-free. As we have seen above, FRIT may traverse the perimeter of an obstacle just like a bug algorithm does. However, the main difference between FRIT and bug algorithms is that the decision in FRIT is made using search, whereas for bug algorithms the turning direction (right or left) is determined by the design of the algorithm and thus not based on search. As such, depending on the situation, Bug algorithms could work better or worse than FRIT.

Figure 2.6 compares the behaviors of FRIT and Bug2, a popular bug algorithm which is the base of a number of other bug algorithms. 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 a bug algorithm will outperform FRIT.

An important difference between FRIT and Bug algorithms is that FRIT is designed to work on *general* search spaces, whereas Bug algorithms are specifically designed for 2D pathfinding applications.

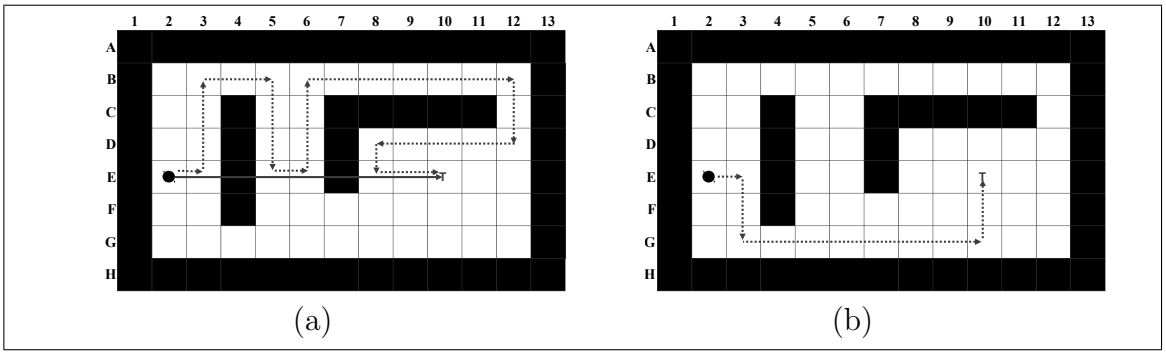


FIGURE 2.6. Bug2 (a) and FRIT (b) in a pathfinding scenario in which the goal cell is E10 and the initial cell is E2. The segmented line shows the path followed by the agent.

2.8. Summary

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 branches in the ideal tree when the agent is disconnected from it. We show that with slight modifications we can use a real-time search algorithm for this purpose, 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.

We provide theoretical results proving that both FRIT and FRIT_{RT} always find solutions if these 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 all tight time constraints.

As a disadvantage of our approach, we note that FRIT cannot exploit 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.

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)* (p. 431-436).

Bond, D. M., Widger, N. A., Ruml, W., & Sun, X. (2010, July). 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)* (p. 1936-1941). Barcelona, Spain.

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

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)* (p. 49-56).

Bulitko, V., Björnsson, Y., Sturtevant, N., & Lawrence, R. (2011). Real-time heuristic search for game pathfinding. In P. A. Gonzalez Celero (Ed.), (p. 1-30). Springer Verlag.

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

Deo, N., & Pang, C.-Y. (1984). Shortest-path algorithms: Taxonomy and annotation. *Networks*, *14*(2), 275-323.

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

Fredman, M. L., & Tarjan, R. E. (1984). Fibonacci heaps and their uses in improved network optimization algorithms. In *Proc. of the 25th ieee symposium on foundations of computer science (FOCS)* (p. 338-346).

Gabriely, Y., & Rimon, E. (2008). CBUG: A quadratically competitive mobile robot navigation algorithm. *IEEE Transactions on Robotics*, *24*(6), 1451-1457.

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.

Hernández, C., & Baier, J. A. (2011, June). 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, May). Tree adaptive A*. In *Proc. of the 10th Int'l joint Conf. on autonomous agents and multi agent systems (AAMAS).* Taipei, Taiwan.

Horiuchi, Y., & Noborio, H. (2001). Evaluation of path length made in sensor-based path-planning with the alternative following. In *Proc. of the 2001 ieee Int'l Conf. on robotics and automation (ICRA)* (Vol. 2, pp. 1728–1735).

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

Kamon, I., & Rivlin, E. (1997). Sensory-based motion planning with global proofs. *IEEE Transactions on Robotics and Automation*, *13*(6), 814–822.

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

Koenig, S., & Likhachev, M. (2001). Incremental a*. In *Proc. of the 14th Conf. on advances in neural information processing systems (NIPS)* (p. 1539-1546).

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)* (p. 1311-1312).

Koenig, S., & Likhachev, M. (2006a). A new principle for incremental heuristic search: Theoretical results. In *Proc. of the 16th Int'l Conf. on automated planning and scheduling (ICAPS)* (p. 402-405). Lake District, UK.

Koenig, S., & Likhachev, M. (2006b). 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., Likhachev, M., & Furcy, D. (2004). Lifelong planning a*. *Artificial Intelligence*, *155*(1-2), 93-146.

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.

Korf, R. E. (1997). Finding optimal solutions to rubik's cube using pattern databases. In (p. 700-705).

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* (p. 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*(1-4), 403–430.

Magid, E., & Rivlin, E. (2004). CAUTIOUSBUG: A competitive algorithm for sensory-based robot navigation. In *Proc. of the 2004 Int'l Conf. on intelligent robots and systems (IROS)* (Vol. 3, pp. 2757–2762).

Ng, J., & Bräunl, T. (2007). Performance comparison of bug navigation algorithms. *Journal of Intelligent and Robotic Systems*, *50*(1), 73–84.

Pohl, I. (1971). Bi-directional heuristic search. In *Machine intelligence 6* (p. 127-140). Edinburgh, Scotland: Edinburgh University Press.

Rao, N. S., Kareti, S., Shi, W., & Iyengar, S. S. (1993, July). *Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms* (Tech. Rep. No. ORNL-TM-12410). Oak Ridge National Laboratory.

Rivera, N., Baier, J. A., & Hernández, C. (2013). Weighted real-time heuristic search. In *Proc. of the 11th Int'l joint Conf. on autonomous agents and multi agent systems (AAMAS).* Retrieved from `http://www.cs.toronto.edu/~jabaier/publications/RiveraBH13.pdf`

Rivera, N., Illanes, L., Baier, J. A., & Hernandez, C. (2013). Reconnecting with the ideal tree: An alternative to heuristic learning in real-time search. In *Proceedings of the 6th symposium on combinatorial search (socs).* Retrieved from `http://www.cs.toronto.edu/~jabaier/publications/RiveraIBH13.pdf`

Rokicki, T., Kociemba, H., Davidson, M., & Dethridge, J. (2013). The diameter of the rubik's cube group is twenty. *SIAM J. Discrete Math.*, *27*(2), 1082-1105.

Russell, S. J., & Norvig, P. (2010). *Artificial intelligence - a modern approach (3. internat. ed.)*. Pearson Education.

Sankaranarayanan, A., & Vidyasagar, M. (1990). A new path planning algorithm for moving a point object amidst unknown obstacles in a plane. In *Proc. of the 1990 ieee Int'l Conf. on robotics and automation (ICRA)* (pp. 1930–1936).

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.

Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *Proc. of the 14th Int'l joint Conf. on Artificial Intelligence (IJCAI)* (pp. 1652–1659).

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, July). 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)* (p. 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)* (p. 3981-3986).

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