REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Backtracking and Value Back-Propagation in Real-Time Search

Sverrir Sigmundarson
Master of Science
June 2006

Supervisor:
Yngvi Björnsson
Associate Professor

Reykjavík University - Department of Computer Science

M.Sc. Thesis

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

BACKTRACKING AND VALUE BACK-PROPAGATION IN REAL-TIME SEARCH

by

Sverrir Sigmundarson

Thesis submitted to the Department of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of
Master of Science

June 2006

The undersigned hereby certify that they recommend to the Department of Computer Science at Reykjavík University for acceptance this thesis entitled **Backtracking and Value Back-Propagation in Real-Time Search** submitted by Sverrir Sigmundarson in partial fulfillment of the requirements for the degree of **Master of Science**.

Date

Dr. Yngvi Björnsson, Supervisor
Associate Professor, Reykjavik University, Iceland

Dr. Vadim Bulitko
Assistant Professor, University of Alberta, Canada

Dr. Anna Ingólfsdóttir
Associate Professor, Reykjavik University, Iceland

Abstract

Learning real-time search allows intelligent agents to improve their performance by learning from experience. By interleaving their planning and execution steps they can, in constant time, decide on which action to take from their current state based on limited or incomplete information. This thesis focuses primarily on real-time algorithms that converge to optimal solutions through repeated trials. This work concentrates on two things, firstly it provides an extensive evaluation of established real-time algorithms that seek optimal solutions. While validating earlier research, this evaluation sheds new light on certain behavioral aspects of the algorithms and summarizes their most important properties. Secondly, the thesis examines in detail two types of real-time search enhancements, backtracking and value back-propagation. Based on this analysis, these two enhancements are re-formulated and avenues for their integration into the original LRTA* algorithm are presented. The new search enhancements are shown to improve significantly upon other real-time algorithms and provide a better combination of value back-propagating and backtracking than previous backtracking algorithms.

Útdráttur

Rauntíma lærdómsaðferðir gera greindum forritum kleift að bæta afköst sín með því að læra af reynslu. Með samtvinnaðri áætlunargerð og framkvæmd þá geta forritin, innan skildgreinds tíma, ákvarðað áætlaða bestu ákvörðun frá núverandi stöðu. Þessa ákvörðun geta þau tekið þó einungis liggi fyrir takmarkaðar eða ófullkomnar upplýsingar um umhverfi og stöðu þeirra. Í þessari ritgerð er sjónum eingöngu beint að rauntíma aðferðum sem tryggja að bestu lausnir á vandamálum finnist með ítruðum prófunaraðferðum. Efni ritgerðarinnar skiptist í tvennt, annars vegar er kynnt víðtæk samanburðarrannsókn á rauntímaaðferðum sem tryggja bestu lausnir. Um leið og niðurstöður þessarar samanburðarrannsóknar staðfestir niðurstöður fyrri rannsakenda, þá varpar rannsóknin nýju ljósi á ákveðin hegðunarmynstur aðferðanna og gefur yfirlit yfir afköst og helstu eiginleika þeirra. Hins vegar skoðum við í ritgerðinni ítarlega tvær tegundir betrumbóta á rauntíma aðferðum, þ.e. *rakningu* og *upplýsingaútbreiðslu*. Við umbreytum framsetningu þeirra lítillega og kynnum hvernig mögulegt er að nýta breyttu aðferðirnar sem hluta af LRTA* reikniritinu. Við birtum niðurstöður sem sýna að betrumbætur okkar auka umtalsvert afköst eldri rauntímaleitaraðferða ásamt því að sameina betur en áður hefur verið mögulegt, notkun rakningar og upplýsingaútbreiðslu í rauntímaleitum.

To my parents

Acknowledgements

First and foremost, I would like to address special thanks to my advisor, Dr. Yngvi Björnsson for his patient guidance, insightful comments and unending support during my research work.

My deepest love and affection go to my fiancée Rannveig. She has been an unfailing source of strength during my university studies.

I am also very grateful for my family's encouragement and invaluable moral support during my academic work.

I thank Gunnar Kristjánsson for his help during coding and debugging of the testing software and throughout the experiments sessions.

I am also deeply grateful to all the people that spent countless hours helping me proofread this text and for their comments and advice. You were more helpful than you may think.

Finally, I would like to thank the members of my thesis committee for reading this thesis and providing me with valuable suggestions and comments.

This research was supported by grants from The Icelandic Centre for Research (RANNÍS) and by a Marie Curie Fellowship of the European Community programme *Structuring the ERA* under contract number MIRG-CT-2005-017284.

Publications

A part of the material in this thesis was published as *“Value Back-Propagation versus Backtracking in Real-Time Heuristic Search”* to be presented at the twenty-first national conference on Artificial Intelligence (AAAI06), in the workshop *Learning for Search* to be held July 16, 2006 in Boston, Massachusetts, USA.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Imagine yourself stranded in an unfamiliar maze, left with only a compass, a blank piece of paper, pencil and a vague idea that the maze has an exit somewhere to the east. What do you do? More importantly, how can you exit the maze?

A variety of pathfinding algorithms exist, such as A* (P. E. Hart, Nilsson, & Raphael, 1968), depth-first search or breadth-first search, that can help you find a path through the maze you have so unfortunately got yourself into. However soon you would discover that running an algorithm such as A* is more problematic than simply keeping track of the utility of visited areas. A* belongs to a class of algorithms referred to as *off-line*, indicating their dependency on a completion of an exhaustive planning phase prior to instructing you on what your first action should be. Off-line algorithms are specially designed to find solutions in environments that are fully known beforehand. Many of them, for example A*, are guaranteed to find the best possible solution to the problem. Therefore these algorithms are not only interested in finding *a* solution to their problem but also in finding the best solution. This perfectionism unfortunately contributes to one of their major weakness; an exponential running time (Dechter & Pearl, 1985). This severe resource restriction hinders these algorithms in solving very large problems, and problems that have restricted running time (e.g. computer games). Since the maze you are stuck in is unfamiliar to you and your main goal is to find an exit as soon as possible you are pretty much stranded.

But do not fret. These types of problems are exactly what real-time algorithms aim to solve. Namely problems where you have little or no initial knowledge of your surroundings and a limited set of available resources. In Richard Korf's pioneering work on real-time search he observed these limitations and the obvious disadvantages of current single-agent search algorithms (Korf, 1990). He brought attention to the extensive research previously done in the area of two-player heuristic search, which contrary to their off-line single-agent counterparts, operate almost exclusively in extremely large or resource restricted environments. These algorithms include methods such as the classic minimax search (Shannon, 1950) and its $\alpha\beta$-minimax extension (T. Hart & Edwards, 1963). The most interesting real-time aspect of two-player algorithms is that they must be ready to take actions based on a limited lookahead search within a certain resource limit. Consequently, two-player search research is mainly focused on finding the currently best action available in each state given the search's limiting factors. Two-player search methods make this possible by interleaving their planning and execution phase, enabling them to commit to actions based on only initial or learned assumptions rather than complete information of their end result.

Korf proposed two new types of algorithms to deal with real-time problems, Real-time A* (RTA*) and Learning-RTA* (LRTA*), both of which are based on principles adopted from two-player search. These algorithms (and their descendants) can be classified as *online* as they interleave their planning and execution phase based on available resources. Contrary to their off-line counterparts, they are intended to operate in environments that, as an example, have a limited search horizon, bounded resources, or must commit to actions in constant time. These types of online algorithms are a special case of *Agent-Centered Search* (Koenig, 2001). Offline algorithms, such as A*, are not bound by the same restrictions as agent-centered searches and can evaluate states in a non-continuous fashion during planning. While the intertwining of the planning and execution phase restricts the planning phase of online algorithms to the part of the problem domain that surrounds the current state of the agent, this limited lookahead search however enables them to commit to actions which can be immediately executed from the agents current state, although a complete path from the start state to the goal state may still be unknown. As a result the fundamental design principles of online algorithms (hereon after referred to as real-time) are vastly different from those of A* and other off-line algorithms.

The problem solving approach of real-time algorithms is in many ways similar to humans. We humans mainly rely on experimental problem solving, approaching the final solution through a series of small experiments. At the conclusion of each experiment we perform an evaluation to determine whether or not the experiment brought us closer to solving the larger problem. This is basically what real-time algorithms do, they perform a limited planning phase to decide on what experiment (action) they should perform next, execute it and finally evaluate their resulting state. This approach of *discovery through experimentation* allows real-time algorithms to operate in environments that are unknown beforehand. This technique also enables real-time algorithms to find solutions to very large and complex problems where finding optimal solutions is intractable due to constraints or limitations in resource usage.

The motivations for this thesis are mainly the need for an unified comparison study of the current real-time algorithms and their performance in a variety of testbeds. In the light of recent research, backtracking and value back-propagation in real-time search can be reformulated and presented in a slightly modified form. Finally to facilitate the discovery of new search extensions and further development of current ones, a better understanding is needed of how certain domain properties influence the performance of real-time algorithms.

The main contributions of this thesis are a detailed dissection of backtracking and its accompanying value back-propagation phase in real-time search. Followed by a formulation of a simple real-time back-propagation extension. Also the introduction of a new real-time algorithm using this new value back-propagation extension augmented with an optional backtracking phase. This algorithm utilizes information gained from its back-propagation phase to make more intelligent backtracking decisions. An extensive evaluation of current real-time algorithms is also presented which focuses primarily on real-time algorithms that find optimal solutions. The main contribution of this evaluation is that it uses the same implementation of test domains and identical problem instances to provide a fair basis for comparison.

This thesis is structured as follows; Chapter 2 details the main principles of real-time algorithms and presents a brief overview of previous research in the field of real-time search. Then Chapter 3 presents an evaluation of several well established real-time methods. A detailed dissection of backtracking and value back-propagation techniques in real-time algorithms are found in Chapter 4. In Chapter 5 the new back-propagating and backtracking real-time algorithm is introduced. Finally Capter 6 concludes the work presented here with a summary of findings and means of future work.

Chapter 2

Real-Time Search Algorithms

The purpose of this chapter is to introduce the design concepts of real-time search and familiarize the reader with the body of work done in real-time research to this day. Section 2.1 begins by outlining the four real-time principles mentioned in the previous chapter and how they apply to real-time search. The chapter then details the seminal work done by Korf, while using the opportunity to introduce the notation used throughout this thesis. It then goes on, offering an overview of the main enhancements made to real-time search methods since Korf's initial work. Finally, Section 2.8 offers a brief description of the work done in combining aspects of prior research to form new methods.

2.1 Principles of Real-Time Search

What makes the design principles of real-time algorithms different from their off-line counterparts is the fact that real-time algorithms must make an informed and deliberate decision in constant time. These constant time constraints may include a limited search time, restricted number of expanded states, bounded number of CPU cycles or frames-per-second, or in the case of our maze predicament the endurance, thirst, hunger or patience of its captive. Consequently real-time algorithms should strife to reduce unnecessary resource consumption while those resources that they do use they should use as efficiently as possible. This means that for any additional memory allocated, new area explored or CPU time reserved the search must draw as much learning as possible from the associated expenses. This results in a slightly different set of principles for real-time algorithms. These principles are as follows (listed in no particular order):

- **Economical propagation of new information**. Since propagation of learned information is in most cases expensive (requires revisiting or re-expansion of previously visited areas) it should be limited only to states where new information is likely to provide the search with the most improvements. Consequently timing the occurrence of learning is crucial. Intuitively learning should only occur where the least amount of effort is wasted or when the newly learned information potentially improves the search performance.

- **Minimizing delays and maximizing responsiveness**. While the search should try to optimize its overall efficiency, it should also strife to minimize its move delay. Move delay is the amount of planning time the agent needs before it is able to commit to an action from its current state. The delay of each trial[1] as well as the delay of the first-trial are also important metrics. The trial delay is the amount of traveling the agent does before it is able to return a complete solution from its initial start position to a goal position (also termed solution delay). The first-trial delay is the time span from when the search started searching with only initial knowledge of the environment until it first returns a complete solution path from the start to the goal. In many real-time environments shorter solution delays are preferable than long delays, although delaying longer may result in solutions closer to optimal.
- **Iterative solution refinement process**. Real-time algorithms should also be considered *any-time* algorithms. As such they should balance their execution time to present a solution to the user as quickly as possible. Any remaining time should then be utilized to improve upon that solution. These improvements should be performed in short iterations each improving its prior solution. As a result all real-time algorithms should distribute their costs equally or as efficiently as possible across their consecutive trials.
- **Stable Convergence Process**. As learning progresses the search must explore new areas. The increased travel cost that results from such exploration should be kept to a minimum between trials. Consequently the algorithm should perform exploration in controlled steps and aim to detect and terminate extensive and unnecessary exploration as soon as possible (Shimbo & Ishida, 2003).

Any real-time algorithm should satisfy as many of these four basic principles as possible. To this day balancing these real-time principles has been the main focus of real-time research. Identifying how environmental properties interact with individual search enhancements is crucial to discovering how to best balance these principles. In the context of the prior maze example, this equals best to the interest that its captive would have in knowing exactly how successful drawing a map is compared to continually retracing steps to chalking out directional arrows.

Before presenting the algorithms, a formal definition of the search problem is needed. A learning real-time search problem is defined as a 5-tuple: (S, A, c, s_0, S_g) where S is a finite set of states; A is a finite set of actions; $c : S \times A \rightarrow (0, \infty)$ describes the transitional cost of moving from one state to the next, with $c(s, a)$ being the cost of taking action a in state s; s_0 is the initial start state and S_g is the set of goal sets where $S_g \subset S$. Additionally every action is reversible in every state and there exists a goal state in S_g that is reachable from the initial start state s_0.

Now the rest of this chapter will concentrate on describing the main body of research done in real-time search today.

[1] A trial is defined as a single run of the algorithm from its initial start state s_0 to a goal.

2.2 Real-Time A* (RTA*)

Real-Time A* (RTA*) introduced by Korf operates using a basic greedy successor selection policy (Korf, 1990). The policy is simple; in each state, select an action leading to a neighboring state that has the lowest estimated cost of solving the problem plus the cost of traversing to that state. Pseudo-code of RTA* is shown as Algorithm 1; s represents the state the algorithm is currently exploring, $succ(s)$ retrieves all the successor states of state s, and $c(s_1, s_2)$ is the transitional cost of moving from state s_1 to state s_2. At the beginning s is initialized to the start state s_0, and the algorithm then iterates until a goal state is reached (S_g is the set of all goal states). The variable $solutionpath$ keeps the solution found during the trial, which in RTA* case is the same path as traveled (see further discussion in Section 3.3). The algorithm breaks ties arbitrarily.

Algorithm 1 Real-Time A*

```
1: $s \leftarrow$ initial start state $s_0$
2: $solutionpath \leftarrow< empty >$
3: **while** $s \notin S_g$ **do**
4:     $s_{min} \leftarrow argmin_{s' \in succ(s)}(c(s, s') + h(s'))$
5:     update $h(s) \leftarrow secondmin_{s' \in succ(s)}(c(s, s') + h(s'))$
6:     push $s$ onto $solutionpath$
7:     $s \leftarrow s_{min}$
8: **end while**
```

For each state the search visits it performs a limited lookahead search of its successor states (line 4) and uses this information to decide which state it traverses to next. To find a path RTA* only requires that the heuristic values be *admissible*, a bidirectional state-space and there exists a traversable path from the start state to the goal state. However since the heuristic values may underestimate the travel cost to the goal the search must update their values when a more accurate estimate becomes available (line 5). Each time RTA* decides on an action to take, it updates the h-value of its current state with the second lowest successor h-value. Figure 2.1 shows an example of RTA*'s update rule.

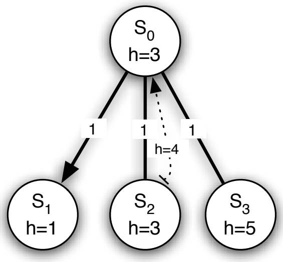


Figure 2.1: While RTA* is traversing the search space it constantly updates its knowledge by recording updated heuristic values for states it visits. Due to RTA*'s update policy it must update the heuristic value of its current state with that of its second best successor to avoid entering and getting stuck in an infinite loop.

By updating the heuristic values this way, RTA* guarantees that it does not enter infinite loops and that it makes, in each state, *locally optimal decisions*. That is RTA* will not select a successor that it has already traversed to unless the second lowest heuristic value of that node is the current lowest cost. Although RTA* makes optimal choices based on the information immediately available to it, it is not guaranteed to make globally optimal decisions (therefore it cannot guarantee finding an optimal solution).

Another problem with RTA* is that when using this heuristic update scheme, RTA* does not preserve the admissibility of its heuristic values, thus making it impossible to reuse the learned values should the same problem be re-encountered at a later time.

2.3 Learning Real-Time A* (LRTA*)

Korf introduced, in addition to RTA*, an algorithm which he termed Learning Real-Time A*, LRTA* for short (Korf, 1990). LRTA* extends the RTA* algorithm onto an iterative solution approach, where the agent is allowed to run repeatedly on the same problem instance using the same start-goal state pair. The agent is run in multiple consecutive trials continually storing and using updated heuristic values from previous trials. Using this technique combined with admissible heuristic values, LRTA* hopes to eventually obtain a globally optimal path through the search space, termed *convergence*.

The LRTA* algorithm, shown as Algorithm 2, is almost identical to that of RTA* apart from its heuristic update function. Figure 2.2 illustrates the update rule for the LRTA* algorithm. As previously mentioned, RTA* updates its heuristic in such a way that after it has finished running, the heuristic values are not guaranteed to retain their admissibility. Therefore RTA* can not guarantee convergence to optimal solutions through repeated trials. However the LRTA* algorithm guarantees convergence through repeated runs on the same problem, therefore its update function must guarantee that the heuristic stays admissible between trials. Consequently it updates the heuristic value of the current state using the heuristic value of its minimum successor state plus the traversal cost (lines 4 to 7). Korf proved that this update scheme retains the admissibility of the h-values while still

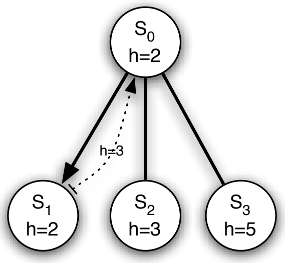


Figure 2.2: LRTA* updates the heuristic value of its current state using the heuristic value of its minimum successor. Using this policy LRTA* retains the admissibility of its heuristic values and is therefore able to re-use them in future trials. This allows LRTA* to eventually make globally optimal decisions, it however hinders it from making locally optimal ones.

enabling them eventually to converge to their exact values (h^*). After convergence has been achieved the search is guaranteed to traverse the optimal path (Korf, 1990).[2]

Algorithm 2 LRTA*

```
1: $s \leftarrow$ initial start state $s_0$
2: $solutionpath \leftarrow < empty >$
3: while $s \notin S_g$ do
4:     $h'(s) \leftarrow min_{s' \in succ(s)}(c(s, s') + h(s'))$
5:     if $h'(s) > h(s)$ then
6:         update $h(s) \leftarrow h'(s)$
7:     end if
8:     push $s$ onto $solutionpath$
9:     $s \leftarrow argmin_{s' \in succ(s)}(c(s, s') + h(s'))$
10: end while
```

Apart from LRTA*'s constant move-delay, its *guaranteed solution optimality* is one of its strongest attributes. However this attribute directly contributes to one of LRTA*'s major flaws. Due to its optimality seeking nature, LRTA* is destined to suffer similar problems as its optimal off-line counterparts, namely their excessive running times. The core of the problem being that while the search is seeking the optimal path it *must* first visit all states that have estimated cost lower or equal to the cost of the optimal path. Therefore LRTA* is prone to excessive wandering within the search space during its convergence process, resulting in considerable fluctuations in travel cost between trials (Ishida & Shimbo, 1996). Another more subtle flaw in LRTA*'s convergence process is that after the search has converged, finding an optimal solution, it has in-fact found *all* the optimal paths (Ishida & Shimbo, 1996). Computing all optimal solutions is obviously a wasted effort since returning only the first optimal solution suffices. This is however impossible to prevent when guaranteeing optimality.

Due to these limitations inherent to the original LRTA* algorithm, a wide variety of LRTA*-based algorithms have been proposed that provide solutions to these flaws while still retaining some or all of its better qualities.

2.4 Bounded LRTA* (δLRTA*)

Though LRTA* maintains estimated lower bounds on the solution cost in the form of a heuristic function it is not informed of any upper-bounds to the current solution cost. This may cause the search to wander and explore states that do not belong to the optimal solution path, thus causing the excessive fluctuations in the execution cost observed between trials in LRTA*.

Shimbo and Ishida proposed a bounded version of the LRTA* algorithm, which they termed δLRTA*, to address this problem (Ishida & Shimbo, 1996). The pseudo-code for δLRTA* is shown as Algorithm 3. The idea is that if an upper-bound can be posed on the

[2] LRTA* is said to have *converged* when it updates no heuristic values during a trial.

solution cost then the search could narrow its search window even further, enabling it to eliminate states that exceed the bound. δLRTA* achieves this by maintaining a second heuristic function for the upper-bound of each state in the search space (termed $h_u(s)$). The δ factor controls how wide these upper-bounds on exploration are allowed to be; higher the values of δ the wider the bounds.[3]

Algorithm 3 Bounded LRTA*

```
1: $s \leftarrow$ initial start state $s_0$
2: $solutionpath \leftarrow < empty >$
3: $\Theta \leftarrow (1+\delta) * h_u(s_0)$
4: while $s \notin S_g$ do
5:     for all $s' \in succ(s)$ do
6:         $h_u(s') \leftarrow min[h_u(s'), (h_u(s) + c(s', s))]$
7:     end for
8:     update $h(s) \leftarrow max[h(s), min_{s' \in succ(s)}(c(s, s') + h(s'))]$
9:     update $h_u(s) \leftarrow min[h_u(s), min_{s' \in succ(s)}(c(s, s') + h_u(s'))]$
10:    push $s$ onto $solutionpath$
11:    $s \leftarrow argmin_{\{s' \in succ(s) \wedge h_u(s') \leq \Theta\}}(c(s, s') + h(s'))$
12: end while
13: for all states $s'$ in a LIFO order from $solutionpath$ do
14:     update $h_u(s') \leftarrow min(h_u(s'), h_u(s_g))$
15: end for
```

The behavior of δLRTA* is similar to that of LRTA*. However when it selects the lowest estimated successor it does not do so from the set of all successors of state s. Instead it calculates a *safe list* ($Safe(s)$) containing all successor states that fall within the current state's estimated upper-bound. The $Safe(s)$ set consists of all successor states of state s, where each successor state s' satisfies the currently calculated upper-bound, denoted by Θ, or $h_u(s') \leq \Theta$, where $\Theta = (1+\delta)h_u(s_0)$. After computing this set of *safe* successors the search selects the minimum successor from it, which it then moves to. To ensure these upper-bounds reflect correct information from the previous trial, each trial must end with the back-propagation of new upper-bound values through all the states on the solution path (lines 5 to 7).

Besides the added overhead of storing and maintaining the upper-bound heuristic function, obtaining the initial upper-bound values is far from trivial. Since $h_u(s) \geq h^*(s)$ must be kept, great care must be taken when providing and maintaining these limits. A simple solution is to initially assign the upper-bound values of all states to a predefined infinity value (i.e., a *very* large number). While this is a relatively cost-free implementation it results in the first trial of δLRTA* being identical to that of LRTA*.

δLRTA* maintains its upper-bound values in two ways:

1. At the end of each trial new upper-bound values are back-propagated from the goal state back to the start state, updating all states that lie on the current trial $solutionpath$ (lines 13 to 15).

[3] δLRTA* can be viewed as a special case of the LRTA* algorithm, since for a fixed value of $\delta = \infty$ it behaves just as LRTA*.

2. Before the successor selection policy is invoked, upper-bound values of the current state are propagated *towards* the successors of the current state (lines 5 to 7).

The latter propagation is done to ensure the explorative nature of the algorithm after the upper-bound values become finite. As pointed out by the algorithm's authors, the latter update is not valid in directed domains thus limiting the current implementation of δLRTA* to bidirectional domains.

When Shimbo and Ishida later proved the completeness of the δLRTA* algorithm (Shimbo & Ishida, 2003) they proved that the δLRTA* algorithm converges to an optimal path only if $\delta \geq 2$, whereas all δ values lower than 2 converge to suboptimal paths. This could be explained by the fact that values lower than 2 reduce the amount of exploration that δLRTA* is allowed to perform to such an extent that the search can never explore other paths than those explored during its early trials.

2.5 FALCONS

Furcy and Koenig argued, contrary to prior remarks made by Shimbo and Ishida (Shimbo & Ishida, 2003), that speeding up convergence could be attained without sacrificing the optimality of the final solution (Furcy & Koenig, 2000). They introduced a LRTA* variant they termed FALCONS (FAst Learning and CONverging Search). FALCONS is shown as Algorithm 4. It employs a different successor selection policy than the traditional greedy policy of LRTA* (lines 4 and 5). FALCONS alternatively uses the $f()$ cost value function similar to the traditional A* search, where $f(s) = g(s) + h(s)$ and $g(s)$ is the best known cost of traveling to the current state s from the initial start state s_0. As a result FALCONS not only tries to minimize the path to the goal from its current state but also seeks to minimize the distance traveled so far from the start. By this, FALCONS attempts to focus the search more on states that lie in the vicinity of the optimal path.

Furcy and Koenig showed that FALCONS does indeed converge in fewer trials than LRTA* and with a slightly lower total travel cost. FALCONS first-trial travel cost is however significantly higher than that of LRTA*. The authors explain that this could be a result of FALCONS requiring additional travel cost to correct for the error in its g-values.

The FALCONS algorithm requires a second heuristic table to store its estimated $g()$ values (similar to that of δLRTA* discussed in section 2.4). This causes the algorithm to double its memory requirement over that of traditional LRTA* (this additional memory increase is evaluated in Section 3.6). The implementation of the algorithm is also more complex than LRTA*'s since special consideration must be taken to preserve the admissibility of both heuristic functions when updating (lines 6 to 11 in Algorithm 4).

The authors also note that, although applicable, FALCONS is likely to perform very poorly in domains with non-uniform state transitional cost and even converge with a higher travel cost than LRTA*. This is due to the fact that the successor selection rule $f(s)$ chooses to expand successors even though their transitional cost is very high (when

Algorithm 4 FALCONS

```
1: $s \leftarrow$ initial start state $s_0$
2: $solutionpath \leftarrow< empty >$
3: while $s \notin S_g$ do
4:     $S' \leftarrow argmin_{s'' \in succ(s)}(max[g(s'') + h(s''), h(s_0)])$
5:     $s' \leftarrow argmin_{s'' \in S'}(c(s, s'') + h(s''))$
6:     if $s \notin S_g$ then
7:         $h(s) \leftarrow max[\ h(s), min_{s'' \in succ(s)}(h(s'') + c(s, s'')), max_{s'' \in pred(s)}(h(s'') - c(s'', s))]$
8:     end if
9:     if $s \neq s_0$ then
10:        $g(s) \leftarrow max[\ g(s), max_{s'' \in succ(s)}(g(s'') - c(s, s'')), min_{s'' \in pred(s)}(g(s'') + c(s'', s))]$
11:    end if
12:    push $s$ onto $solutionpath$
13:    $s \leftarrow s'$
14: end while
```

their corresponding h-value is very low). Thus theoretically taking a much more expensive route than necessary.

2.6 Search and Learning Algorithm (SLA*)

Shue and Zamani introduced a LRTA* variant which they termed Search and Learning Algorithm (SLA*) (Shue & Zamani, 1993). SLA* was the first real-time algorithm that brought the notion of backtracking to the field of real-time search. Backtracking is simply the act of retracing your steps and reevaluate your previous options based on newly discovered information. Backtracking has been successfully utilized to a large extent in the field constraint satisfaction problems (Dechter & Frost, 1998).

The SLA* algorithm, shown as Algorithm 5, is similar to LRTA*. Their difference is only apparent after a heuristic update occurs in state s (line 6). In that case SLA* does not move to the minimum successor of state s but rather backtracks immediately to the state it previously came from (lines 7 and 8). SLA* continues backtracking as long as it encounters updates to the current state's heuristic value. There are two reasons for this backtracking; first to back-propagate newly discovered information (in the form of updated h-values) as far back as possible and second to offer the search the chance to re-evaluate its previous actions given new information.

A result of the aggressive nature of SLA*'s backtracking is that it effectively performs all of its learning during the first trial and immediately converges to an optimal path. This excessive backtracking results in an extremely high travel cost during the first-trial. Although the overall travel cost of the algorithm is sometimes faster than LRTA*, this

Algorithm 5 SLA*

```
1: $s \leftarrow$ initial start state $s_0$
2: $solutionpath \leftarrow < empty >$
3: while $s \notin S_g$ do
4:    $h'(s) \leftarrow min_{s' \in succ(s)}(c(s,s') + h(s'))$
5:    if $h'(s) > h(s)$ then
6:       update $h(s) \leftarrow h'(s)$
7:       $s \leftarrow$ top state of $solutionpath$
8:       pop the top most state off $solutionpath$
9:    else
10:      push $s$ onto $solutionpath$
11:      $s \leftarrow argmin_{s' \in succ(s)}(c(s,s') + h(s'))$
12:   end if
13: end while
```

poor first-trial performance is considered SLA* most serious flaw; largely since most real-time environments would prefer a suboptimal solution returned as early as possible rather than an optimal solution that takes an excessively long time to converge to. This is especially relevant in vast state spaces where optimal solutions are unattainable.

SLA*T (Shue & Zamani, 1999), shown as Algorithm 6, addresses the single-trial convergence problem of SLA* by introducing a user-definable learning threshold. The threshold, T, is a cumulative parameter which controls the amount of heuristic updates that occur before backtracking is allowed. That is, SLA*T only backtracks after it has overflowed its T parameter.

Algorithm 6 SLA*T

```
1: $s \leftarrow$ initial start state $s_0$
2: $solutionpath \leftarrow < empty >$
3: $t = 0$
4: while $s \notin S_g$ do
5:    $h'(s) \leftarrow min_{s' \in succ(s)}(c(s,s') + h(s'))$
6:    if $h'(s) > h(s)$ then
7:       update $h(s) \leftarrow h'(s)$
8:       $t \leftarrow t + \Delta h(s)$
9:    end if
10:   if $t \geq T$ then
11:      $t \leftarrow T$
12:      $s \leftarrow$ top state of $solutionpath$
13:      pop the top most state off $solutionpath$
14:   else
15:      push $s$ onto $solutionpath$
16:      $s \leftarrow argmin_{s' \in succ(s)}(c(s,s') + h(s'))$
17:   end if
18: end while
```

It can be observed that SLA*T with a threshold $T = \infty$ behaves exactly as the LRTA* algorithm with a lookahead of one. With T set to ∞ the algorithm never performs the backtracking step and thus strictly follows the LRTA* greedy function. However a threshold value of $T = 0$ is identical to the SLA* algorithm since the threshold condition is always true. The search then performs SLA*'s aggressive backtracking policy each time the heuristic value is updated.

SLA*T suffers from two serious flaws; first, that the most effective value of the T parameter is highly dependent on the domain and its structure (the length of the solution path, the error of the heuristic etc.). This makes it necessary to tune the SLA*T algorithm specifically to each problem that it is made to solve. Second, the SLA*T algorithm does not form a pure hybrid method from both LRTA* and SLA* but instead uses the T parameter to control which is active at any given time. Therefore at the same time that SLA*T draws on LRTA* to provide it with its much needed iterative convergence process it makes itself susceptible to all of LRTA*s flaws. SLA*T consequently suffers from the combined flaws and limitations of both SLA* and LRTA*.

2.7 Other methods

Several other methods have been proposed for agent navigation in resource bounded environments. Among them several methods deal with increased planning efficiency by re-using learned information over a number of consecutive trials. CRTA* and SLRTA* (Edelkamp & Eckerle, 1997) apply to navigation through unknown domains. CRTA* extends RTA* such that it can be used iteratively when problem solving (similar to LRTA*). CRTA* achieves this by back-propagation of correct heuristic information through its entire solution path at the end of each trial, correcting inadmissible heuristic values. As the authors mention then CRTA*'s update procedure is rather costly since it requires the agent to physically revisit each state during the update procedure. SLRTA* uses the same technique but attempts to improve upon CRTA* by delaying the updating until the search re-encounters a state.

Recently a method termed LRTA*(k) (Hernández & Meseguer, 2005b, 2005a) was proposed; it is closely related to the LCM algorithm (Pemberton & Korf, 1992). LRTA*(k) improves upon LRTA* by using back-propagation techniques to update already visited states. In its most simple form when the algorithm discovers under-estimated heuristic values it starts an update procedure that updates heuristic values of all states in a fixed radius around the current state. The size of the update radius is limited by the algorithm's k parameter. Three variants of LRTA*(k) were proposed, each varying in what and how states are considered for updating. Koenig proposed a version of LRTA* that uses a limited A* search in its local search space (LSS) to determine what action should be taken (Koenig, 2004). It then updates all the states expanded during its limited A* search using a form of the Dijkstra algorithm, a computationally expensive step. Due to the limited A* search the algorithm is easily distracted and mislead if given an error prone heuristic. Koenig's algorithm is strongly influenced by the early observations of Russell and Wefald (Russell & Wefald, 1991) that updating of all values in the LSS are more beneficial than just the updating of the current state. Recently a version of LRTA* that prioritizes the updating of heuristic values was proposed. When Prioritized-LRTA*, P-LRTA* for short, (Rayner, Davison, Bulitko, & Lu, 2006) updates the h-value of state s it stores in a priority queue each of the successors of s along with the Δh of state s. For each state expanded the algorithm then systematically updates the states from the priority queue (in descending order of Δh) up to a maximum of n states (where n is a user-definable parameter). A drawback of P-LRTA* is that because its state-update function updates states in a non-continuous fashion, it consequently assumes knowing successor information of unexplored states. Additionally, although not addressed by the authors, P-LRTA* with $n = \infty$ likely performs all its learning during its first trial.

Shimbo and Ishida, in addition to their δLRTA* algorithm, also introduced a weighted version of the LRTA* algorithm. This version, termed ϵLRTA*, scales the initial heuristic by a constant factor ϵ (where $\epsilon > 0$), thus trying to reduce the under-estimation error in the heuristic values. However by scaling the heuristic the search also sacrifices its optimality attribute. Results showed that by increasing the scaling factor travel cost could be reduced but at the cost of increased sub-optimality of the solutions.

2.8 Combining Extensions

eFALCONS In a later paper Furcy and Koenig created a hybrid version of their FALCONS called eFALCONS (Furcy & Koenig, 2001). This new version addresses the original FALCONS poor performance during its early trials (notably its first-trial). Their reported improvement to the total travel cost was on average only around 2-6%. The eFALCONS algorithm not only uses its original g-value estimates but introduces two new estimates, consequently increasing its memory requirement two-fold to accommodate their storage. Unfortunately this added memory requirement completely dwarfs the negligible travel cost improvements.

$\epsilon\delta$-LRTA* Ishida and Shimbo also noted that their weighted and bounded LRTA* enhancement could be combined to create an even stronger algorithm (Shimbo & Ishida, 2003). The algorithm, termed $\epsilon\delta$-LRTA*, is identical to δLRTA* but uses an initial heuristic function scaled by the ϵ factor. They provided a small example of how this algorithm, while stabilizing the convergence process, reduces the overall learning time at the cost of sacrificing optimality.

γ-Trap (Gamma-Trap) Bulitko proposed an algorithm in (Bulitko, 2004) termed γ-Trap. This algorithm combines the sub-optimality element from ϵLRTA* and the backtracking technique from SLA*. The algorithms γ parameter is user definable and is equivalent to the ϵ parameter of the weighted LRTA* algorithm. It uses a variable lookahead during its planning phase to detect (and subsequently avoid) minima in the heuristic function, termed *traps*. The algorithm, by using variable lookahead, updates its heuristic value in a more aggressive manner than the LRTA* algorithm, termed max-of-min. Additionally the algorithm intuitively executes (and backtracks) multiple actions at once, to a maximum length equal to its lookahead depth. Like the weighted LRTA* it sacrifices optimality to speed up convergence using its γ parameter. However as both the SLA* and FALCONS algorithm, the γ-Trap exhibits more expensive travel costs during its early trials.

The LRTS Framework The Learning Real-Time Search (LRTS) (Bulitko & Lee, 2006) builds on the LRTA* algorithm and formally combines it with: SLA*'s backtracking, SLA*T's threshold control and ϵLRTA*'s idea of heuristic scaling; forming a single coherent algorithm. Additionally it utilizes the variable lookahead element coupled with the max-of-min heuristic updating introduced in the γ-Trap algorithm. By a careful selection of values for each of LRTS's three parameters (γ, d and T) it achieves better convergence performance than both FALCONS and LRTA* on the 15-puzzle, achieving up to 50% convergence (albeit to sub-optimal solutions). Selecting the optimal set of parameter values is however far from trivial. An evaluation demonstrating that the LRTS algorithm indeed performs better than its extensions would individually, was not provided by the authors. Due to LRTS inclusion of the thresholding control parameter T, it is suspected to suffer from the same problem specific value selection as the SLA*T algorithm. In a recent study of lookahead pathologies the LRTS framework was augmented with a dynamic selection of lookahead depths from pre-computed data (Luštrek & Bulitko, 2006).

Chapter 3

Comparison of Algorithms

In earlier research on real-time algorithms comparisons were done on several real-time algorithms in disjoint sets of test domains (Shimbo & Ishida, 2003; Bulitko & Lee, 2006). Due to the implementation difference of both the algorithms and the test domains used, the results obtained from these researches are not entirely comparable. A clear and unified overview is needed that covers all the previous independent research results. To ensure fair comparison, such comparison must be based on a unified implementation of both the algorithms themselves and a diverse set of testbeds. The algorithms must then all be tested using the same initial problem instances. This chapter presents such a research and while primarily confirming the findings of prior research it sheds new light on a few previously unknown algorithm properties.

3.1 Experimental Setup

In this chapter the methods described in Chapter 2: LRTA*, δLRTA*, FALCONS, SLA* and SLA*T, will be empirically evaluated and compared using three different application domains. The performance of the LRTA* algorithm will be used as a baseline for all comparisons. For simplicity and fair comparison all the algorithms were run with a fixed lookahead of one (see Appendix A). The SLA*T algorithm used T values of 100, 1,000 and 10,000 respectively and δLRTA* was run with δ set to 2, 4 and 10. The values for δLRTA* were selected to provide comparisons with previous research (Shimbo & Ishida, 2003). δLRTA* is omitted from all tables showing first-trial performance throughout this chapter. This is due to the fact that since the ∞-heuristic is used as the initial upper-bound heuristic, δLRTA*'s first-trial is identical to that of LRTA*. All the algorithms were implemented using a constant access-time hash-table to manage updated heuristic values.

The first domain used was pathfinding in the Gridworld (Shimbo & Ishida, 2003). All grid instances were of size 100x100. Three different obstacle ratios were used: 20%, 30% and 40%. One hundred instances and start-goal state pairs were randomly generated for each obstacle ratio. The second domain was also a pathfinding problem, but on eight game maps from the commercial computer game Baldur's Gate. This domain was specifically chosen to provide a more realistic evaluation (Björnsson, Enzenberger, Holte, Schaeffer, & Yap, 2003). For each of the eight game maps a set of 400 randomly chosen start-goal state pairs were chosen. Both pathfinding domains used the Manhattan-distance heuristic defined as the sum of difference between the current state's and the goal state's horizontal and vertical positions.[1] The third domain was the sliding-tile puzzle (Korf, 1990); 100 unique start configurations of both the 8- and 15-piece versions of this puzzle were used. The 8-puzzle instances were randomly created but Korf's 100 instances were used in the 15-piece version (Korf, 1985). The admissible Manhattan-distance heuristic was used as the initial heuristic in the sliding-tile puzzle. It is defined as the sum, for all tiles, of their horizontal and vertical distances from their respective goal positions. All problem domains utilized the 4-way tile-based movement when generating successor states and all actions were *unit-cost*. A more detailed description and discussion of the test domains can be found in Appendix A.

Neither LRTA*, FALCONS nor SLA*T were observed to converge on any of the 15-puzzle instances provided by Korf within a travel cost of 50 million states. Therefore only the first-trial information from the 15-puzzle is used for comparison. The only algorithm that converges on any of the 15-puzzle instances is SLA*. It converged on 95 instances (out of 100) incurring an average travel cost of 32,754,655 states while on average storing only 12,432,457 states in memory. On several 15-puzzle instances both SLA* and SLA*T exceeded their resource limit of 50 million states traveled during their first-trial. Therefore comparison of the 15-puzzle had to be limited to only 82 out of Korf's 100 instances, which all algorithms were able to find a solution to during their first trial.

3.2 Travel Cost

Real-time search algorithms are most commonly compared on the basis of their *travel cost* (also known as execution cost, solution cost or running time). Travel cost, as the name indicates, is the total cost per trial that the search incurs while traveling through the problem domain in the search of a solution. In the experiments presented in this chapter the lookahead is limited to a fixed depth of one, therefore in all cases the search travel cost equals the number of states *expanded* by the search. Since real-time algorithms do not have complete information of their environment they must perform physical actions to discover new information. In this thesis it is assumed that, since deliberation cost per move is upper-bounded by a constant, the cost of physical travel outweighs the cost of internal computation. Consequently travel cost incurred by the algorithms is considered their dominant cost-factor.

This section will start by evaluating the average total travel cost of the algorithms and then comparing their first-trial travel cost. Finally a special section will discuss a pathological anomaly discovered in the SLA*T algorithm.

[1] The FALCONS algorithm uses the same equation when estimating its g-values. The only difference is that it substitutes the goal state for its initial start state.

Table 3.1: Average Travel Cost to Convergence

	Baldur's Gate		**Gridworld**		**8-puzzle**	
Algorithm	Travel Cost	% of LRTA*	Travel Cost	% of LRTA*	Travel Cost	% of LRTA*
FALCONS	25,112	41.9%	16,999	57.1%	29,346	40.0%
LRTA*	59,916	100.0%	29,760	100.0%	73,360	100.0%
SLA*	**17,374**	**29.0%**	**11,030**	**37.1%**	**2,226**	**3.0%**
SLA*T(100)	29,518	49.3%	17,705	59.5%	149,646	204.0%
SLA*T(1,000)	51,559	86.1%	24,495	82.3%	77,662	105.9%
SLA*T(10,000)	59,691	99.6%	28,426	95.5%	73,357	100,0%
δLRTA*-2	62,236	103.9%	28,734	96.6%	28,330	38.6%
δLRTA*-4	59,144	98.7%	27,871	93.7%	42,105	57.4%
δLRTA*-10	58,917	98.3%	28,840	96.9%	60,079	81.9%

3.2.1 Total Travel Cost

Table 3.1 shows the average travel cost of the real-time algorithms. For each algorithm their total travel cost and its relative cost compared to LRTA* is reported. The table shows that SLA* outperforms LRTA* in all domains, requiring only a fraction of LRTA*'s total travel cost. In the most extreme case, in the 8-puzzle domain, SLA*'s travel cost is only 3% of that reported by LRTA*. SLA* performs the best of all the algorithms. This is most clearly demonstrated in the sliding-tile domain. There, while only reporting travel costs a fraction of what is needed by the second lowest method, SLA* is also the only optimal seeking algorithm that converges to optimal solutions on any of the 15-puzzle problems. Though the other methods are roughly equal in the pathfinding domains, FALCONS performs noticably better than both LRTA* and the SLA*T variants in the sliding-tile puzzle. FALCONS however harbors hidden costs since maintain its g-values, it requires twice the memory of LRTA*. Memory usage is evaluated in more detail in Section 3.6.

The 8-puzzle results reported for δLRTA* in Table 3.1 are consistent with previous research (Shimbo & Ishida, 2003). However the results from both pathfinding domains somewhat contradict previous findings. Earlier Shimbo and Ishida reported that their δLRTA* algorithm decreased its travel cost in the Gridworld for lower values of δ. While the Gridworld results reported here indicate only a marginal increase in travel cost when δ is decreased from 4 to a value of 2, the extent of the discrepancy is most obvious in the Baldur's Gate domain. There δLRTA* reports a 60% higher travel costs than FALCONS, resulting in a travel cost close to that of LRTA*. No solid explanation for this behavior can be offered at this time. This however indicates that the values chosen for the δ parameter are to some degree domain dependent.

3.2.2 First-Trial Travel Cost

Table 3.2 shows the first-trial travel cost of the algorithms presented in Table 3.1. As already mentioned, the aggressive backtracking of SLA* concentrates the whole of its travel cost into the first trial. This is clearly evident as SLA* has by far the highest first-trial travel cost in all domains. By this SLA* clearly violates two of the four real-time principles of Chapter 2, namely: a short trial delay and an iterative solution refinement process.

Table 3.2: First-Trial Travel Cost

δLRTA* is omitted since its first-trial is identical to that of LRTA* when using the initial ∞heuristic.

	Baldur's Gate		**Gridworld**	
Algorithm	Travel Cost	% of LRTA*	Travel Cost	% of LRTA*
FALCONS	14,102	390.6%	3,058	136.7%
LRTA*	**3,610**	100.0%	**2,237**	100.0%
SLA*	17,308	479.4%	10,947	489.4%
SLA*T(100)	15,621	432.7%	9,223	412.3%
SLA*T(1,000)	14,026	388.5%	8,404	375.7%
SLA*T(10,000)	10,470	290.0%	7,329	327.6%
	8-puzzle		**15-puzzle**	
Algorithm	Travel Cost	% of LRTA*	Travel Cost	% of LRTA*
FALCONS	680	178.9%	479,187	1,731.5%
LRTA*	**380**	100.0%	**27,674**	100.0%
SLA*	2,205	580.3%	31,498,532	113,819.9%
SLA*T(100)	651	171.3%	5,469,369	19,763.6%
SLA*T(1,000)	**380**	100.0%	1,730,754	6,254.1%
SLA*T(10,000)	**380**	100.0%	268,745	971.1%

By incurring all its travel cost during the first-trial SLA* uses unacceptable amounts of resources while it, during its entire convergence process, is unable to produce any solutions. Although SLA* still clearly satisfies the basic real-time requirements, namely movement in constant time given a limited search horizon, SLA* does however violate two of the four real-time principles discussed in Chapter 2. It is therefore debatable whether or not SLA* can be considered a fully qualified real-time algorithm. For algorithms that do fulfill the real-time principles, the first-trial performance of LRTA* is notably the best of all the algorithms. This is most obvious in the 15-puzzle where LRTA* has a 10 times shorter first-trial travel cost than the second lowest, SLA*T(10,000).

3.2.3 Pathology of SLA*T

As previously mentioned, poorly chosen values for SLA*T's T parameter can have a detrimental impact on its performance. This can be observed in Table 3.1, where the results in the 8-puzzle domain give the best indication of how serious this impact can be. While SLA*T(1,000) performs close to that of LRTA*, SLA*T(100) has by far the worst performance of all the algorithms in the sliding-tile puzzle. One possible cause is that by backtracking the search effectively doubles its LRTA* travel cost.

This can occur when the LRTA* phase of SLA*T is very close to convergence when the T parameter overflows. As a result, when SLA*T shifts to its backtracking SLA* phase it effectively backtracks the entire length of LRTA*'s excessively long traveled path doubling the travel cost of the search in the process. This is supported by the backtracking information presented in Table 3.3. The table shows, for each domain, the average number of backtracking decisions made and the total average number of actions backtracked for both SLA* and SLA*T. While the numbers of actions that SLA*T(1,000) backtracks in the 8-puzzle domain stays relatively comparable with the other domains the SLA*T(100) backtracks significantly more. SLA*T(100) backtracks on average roughly as many actions as the average LRTA* travel cost. This supports the suspicion that the

Table 3.3: Backtracking Statistics

Domain	Algorithm	Backtrack	Actions Backtracked
Baldur's Gate	SLA*	666.5	**8,615.2**
	SLA*T(100)	1,152.0	11,648.7
	SLA*T(1,000)	1,204.1	15,093.4
	SLA*T(10,000)	**610.2**	9,270.2
Gridworld	SLA*	1,548.9	5,428.3
	SLA*T(100)	1,857.8	6,346.2
	SLA*T(1,000)	2,000.5	6,896.7
	SLA*T(10,000)	**1,128.8**	**3,902.4**
8-puzzle	SLA*	**354.5**	**1,090.5**
	SLA*T(100)	21,620.8	64,379.0
	SLA*T(1,000)	1,149.7	3,437.8
	SLA*T(10,000)	0.0	0.0

T parameter is overflowing at a crucial time during the search, causing the performance degradation.

The travel cost of the algorithms varies greatly. SLA* and FALCONS were shown to provide the overall shortest travel cost in all the test domains. The travel cost during their first-trial is however significantly worse than that of LRTA*. In all domains the LRTA* algorithm was shown to have by far the lowest first-trial travel cost of all the algorithms. The importance of the trial cost metric for real-time algorithms is significant, mostly due to the fact that while these algorithms are searching they require a certain degree of exploration. In most cases this exploration results in the execution of a relatively costly travel actions. As a result comparing real-time algorithms foremost on the basis of their travel cost ensures a clear understanding of how efficiently they use their most costly resource. Due to the importance of this metric when evaluating real-time search, it will from now on form the basis of the comparison from here on after (unless stated otherwise). Further discussion about the importance of travel cost and how it should be used to evaluate an algorithms learning quality is detailed in the next section.

3.3 Learning Quality

When evaluating the performance of a real-time search algorithm it is important to measure both its computational efficiency and the quality of the solutions it produces. Different real-time algorithms may produce solutions of different cost. Solution quality can be measured by their relative cost compared to an optimal solution. As interesting as the algorithms efficiency, is the quality of its produced solutions both after the first and the final trial. All the algorithms experimented with in this thesis use an admissible heuristic and are guaranteed to converge to an optimal solution. Consequently their final solution quality is the same. However, the intermediate solution quality may differ significantly from one algorithm to the next. The cost of the path traveled in a trial is not a good metric of the quality of the solution found in that trial. One reason for this is that real-time algorithms may wander in loops and repeatedly re-expand the same states (Korf, 1990; Shimbo & Ishida, 2003). Whereas the cost of traversing these loops rightfully count to-

Table 3.4: First-Trial Solution Length

	Baldur's Gate			**Gridworld**		
	With Loops	No Loops	% of Length	With Loops	No Loops	% of Length
FALCONS	14,102	91	0.6%	3,058	105	3.4%
LRTA*	3,610	90	2.5%	2,237	102	4.6%
SLA*	71	71	100.0%	82	82	100.0%
SLA*T(100)	110	80	72.7%	132	91	68.9%
SLA*T(1,000)	314	85	27.1%	372	97	26.1%
SLA*T(10,000)	1,073	87	8.1%	1,133	100	8.8%
	8-puzzle			**15-puzzle**		
	With Loops	No Loops	% of Length	With Loops	No Loops	% of Length
FALCONS	680	112	16.5%	479,187	4,567	1.0%
LRTA*	380	61	16.1%	27,674	986	3.6%
SLA*	20	20	100.0%	52	52	100.0%
SLA*T(100)	96	41	42.7%	136	76	55.9%
SLA*T(1,000)	380	61	16.1%	1,033	206	19.9%
SLA*T(10,000)	380	61	16.1%	8,067	679	8.4%

wards the travel cost, the loops themselves are clearly superfluous in the solution path and should be removed. Not all algorithms are affected equally by loops. For example, the SLA* algorithm is immune because of its backtracking scheme, whereas LRTA* and FALCONS are particularly prone. As a result when comparing solution quality of different algorithms it is important to eliminate loops from the solution path to ensure a fair comparison. This can either be done online[2] or as a post-processing step after each trial.

Table 3.4 shows how profound the effect of loop elimination can be. The extreme case in our experiments was FALCONS pathfinding game maps. After eliminating loops from the first-trial solutions, the remaining solution path length became only 0.6% of the path traveled. The resulting paths were then on average only sub-optimal by 28%. Similar results can be observed for LRTA* in the same domain. In this domain the true first-

[2] This however comes at a cost of increased move-delay of the algorithm which could possibly break the movement in constant-time requirement of the algorithm.

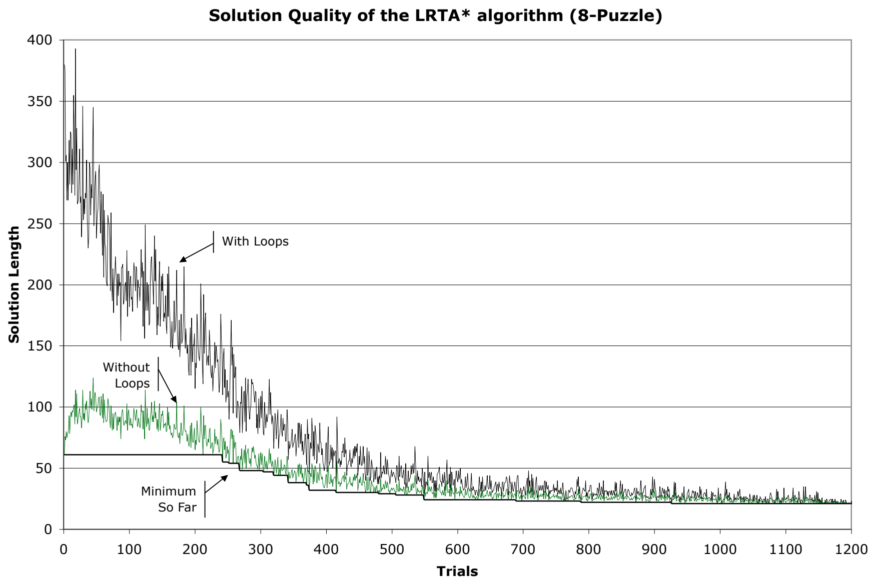


Figure 3.1: Effects of solution path loop elimination in the 8-puzzle.

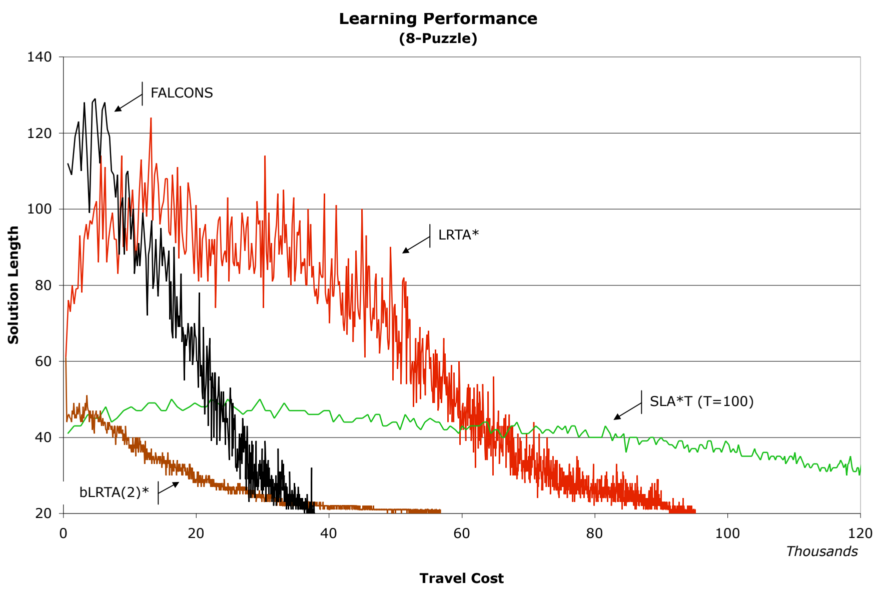


Figure 3.2: FALCONS, δLRTA*($\delta = 2$), LRTA* and SLA*T(100) learning performance on the 8-puzzle domain.

trial solution quality of both LRTA* and FALCONS is clearly much better than it has previously been reported in the literature.

Figure 3.1 illustrates how profound loop elimination is on the performance of LRTA* in the 8-puzzle domain. In early trials there is a large difference in solution costs depending on whether loops are eliminated or not, although on later trials the two gradually converge to the same optimal path. For comparison, a line is included showing the best solution found so far. If the algorithm's execution was stopped before convergence this would be the best solution path it could return at any given moment.

A direct performance comparison of different algorithms can be problematic, even when run on the same problem set. For example, some algorithm may produce sub-optimal solutions, either because they are inadmissible or they do not converge in time. Because the final solution quality may then differ, one cannot look at the travel cost in isolation – there is a trade-off. The same applies if one wants to compare the relative efficiency of different algorithms during intermediate trials. To make comparison possible, the solution length is considered as a function of the accumulated travel cost. This learning performance metric is useful for relative comparison of different algorithms and is an indicator of the algorithms' real-time nature. For example, in Figure 3.2 the SLA* T(100) algorithm quickly establishes a good solution but then converges slowly, whereas both LRTA* and FALCONS although starting off worse, in the end learn faster given the same amount of traveling. The figure also shows that FALCONS makes a better use of its learned values than LRTA* (the steep decent in its solution length is a clear indication). The figure also shows that δLRTA* does indeed offer a stable learning quality but uses its learned values more inefficiently than FALCONS (indicated by its long-tailed convergence).

Further analysis and discussion of the learning quality of the algorithms is found in Appendix B.1.

Table 3.5: Average Trials Until Convergence

	Baldur's Gate		**Gridworld**		**8-puzzle**	
Algorithm	Trials	% of LRTA*	Trials	% of LRTA*	Trials	% of LRTA*
FALCONS	33.5	20.0%	32.2	35.6%	72.8	28.4%
LRTA*	167.1	100.0%	90.7	100.0%	256.4	100.0%
SLA*	1.8	1.1%	2.0	2.2%	2.0	0.8%
SLA*T(100)	49.1	29.4%	46.7	51.5%	202.8	79.1%
SLA*T(1,000)	109.6	65.6%	69.9	77.1%	254.0	99.0%
SLA*T(10,000)	152.0	91.0%	83.7	92.3%	256.3	100.0%
δLRTA*-2	190.0	113.7%	109.4	120.6%	262.9	102.5%
δLRTA*-4	176.6	105.7%	97.8	107.8%	280.4	109.3%
δLRTA*-10	169.7	101.6%	92.5	102.0%	266.4	103.9%

This section discussed the importance of eliminating loops from solution paths prior to comparing different algorithms and quantified the effect of loop elimination in the three test domains. When removing loops algorithms including LRTA* and FALCONS are shown to find much better solutions than previously reported in the literature. A new metric for measuring the learning performance of real-time algorithms was introduced. This metric, termed learning quality, gives new insights into evaluating real-time algorithms during intermediate trials. It uses the solution length as a function of accumulated travel cost. Graphs indicating the learning performance of the real-time algorithms where presented. The relative simplicity of the graphs visual representation gives a better evaluation of how the real-time algorithms utilize available resources during their convergence process to improve their solution quality.

3.4 Convergence Speed

According to the real-time principles a real-time algorithm should offer a reasonably short trial delay as well as a good distribution of its learning across available trials. This can be simply presented by using the number of trials needed for the algorithm to converge. Table 3.5 shows the number of trials needed until convergence was achieved by the real-time algorithms. All tables show the number of trials needed for convergence by each algorithm and its percentage compared to those needed by LRTA*.

The SLA* algorithm converges during its first trial.[3] However as previously mentioned this comes at the cost of compressing the search's entire travel cost into that one trial. Apart from SLA*, FALCONS exhibits, for all domains, the shortest convergence process requiring only 20% to 36% of the trials needed by LRTA*. It is immediately obvious that δLRTA* requires the most trials until convergence and tighter upper-bounds only serve to increase that requirement. While the trials needed for the 8-puzzle stay comparable with previous findings by Shimbo and Ishida the pathfinding results indicate a much higher trial requirements of δLRTA* compared to LRTA* than previously reported.

[3] Convergence is defined as when no h-updates occur during a trial, consequently a second trial is sometimes needed to formally detect SLA* convergence.

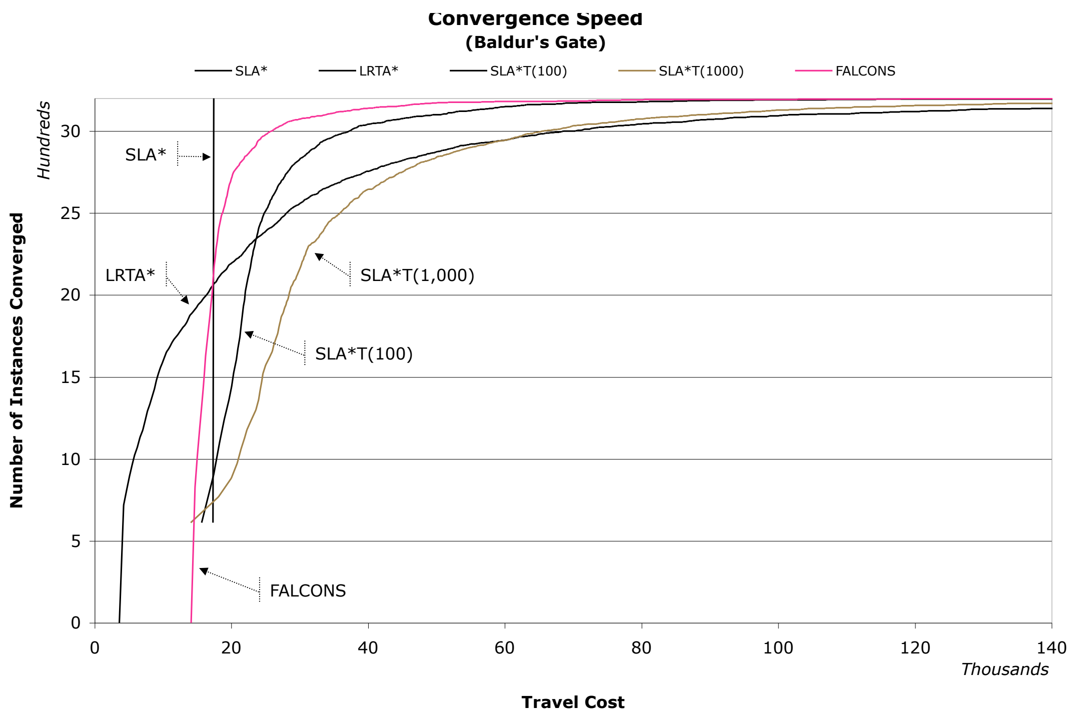


Figure 3.3: Travel cost until convergence in the Baldur's Gate domain.
The graph shows the number of problem instances converged compared to travel cost. SLA* superiority can be clearly observed as it solves all its instances by the time LRTA* and FALCONS have barely solved 2,000 instances.

The values in Table 3.5 coupled with the number of instances converged per travel cost (Figure 3.3) allow for a more precise evaluation of the overall convergence performance of the algorithms. The graph in Figure 3.3 shows how the first-trial delay of FALCONS assists with its overall convergence process. However by the time FALCONS solves its first instances LRTA* has already optimally solved close to 1,700 (although after that FALCONS takes a significant lead in convergence). Consequently during its earlier trials LRTA* far outperforms FALCONS when compared on the basis of the real-time principles. However after an average travel cost of 17,000, LRTA* slows its convergence process significantly while FALCONS increases its convergence speed (this is due to FALCONS high early trial travel costs, after which it utilizes its learned values in a much more effective way than LRTA*).

This section confirmed earlier research stating that the FALCONS algorithm converges fastest in terms of number of trials than any of the other real-time algorithms. The section introduced the number of trials needed for convergence as an important metric when evaluating the algorithms ability to distribute learning across trials. The convergence speed of the algorithms was also depicted using the number of instances converged as a function of accumulated travel cost. This revealed that although FALCONS requires fewer trials to convergence in the Baldur's Gate domain, it has a substantially worse real-time performance in earlier trials than LRTA*.

3.5 Search Stability

An aspect of real-time search performance that has not been addressed in previous sections is how *stable* their convergence process is. The search stability indicates if any fluctuations occur in its travel cost and, if they do, how severe they are. In this section five different indices previously adapted (Shimbo & Ishida, 2003) are used to measure the stability of the convergence process. Four of the indices are adapted from Optimal Control Theory (Dorf, 1988); the integral of absolute error (IAE), integral of square error (ISE) and weighted variants for each (ITAE and ITSE). Additionally the sum of one-sided difference (SOD) is used. Both the IAE and ISE metrics measure the overall excess traveling performed by the search. While these metrics both indicate the overall fluctuations in travel cost by an algorithm they give no hint to if and then how well the algorithm is able to improve its required travel cost over consecutive trials. The notion being that each algorithm should, as the search progresses, reduce its need for exploration by utilizing already learned values; therefore reducing travel cost fluctuations in later trials. To measure this the time weighted versions of both metrics are used (ITAE and ITSE). By punishing fluctuations in later trials more severely than those that happen earlier on these metrics the time weighted indices also favor shorter convergence runs.[4] The equations used to calculate the metrics are shown below. The $travelcost(i)$ function returns the travel cost of the ith trial, $h^*(s)$ returns the optimal travel cost and N is the number of trials the algorithm was allowed.

$$IAE = \sum_{i=1}^{N} \Big| travelcost(i) - h^*(s) \Big|,$$

$$ITAE = \sum_{i=1}^{N} i * \Big| travelcost(i) - h^*(s) \Big|,$$

$$ISE = \sum_{i=1}^{N} \Big(travelcost(i) - h^*(s) \Big)^2,$$

$$ITSE = \sum_{i=1}^{N} i * \Big(travelcost(i) - h^*(s) \Big)^2,$$

SOD indicates whether or not the travel cost worsens and then by how much. If the travel cost is non-increasing between trials this variable only takes its default value of 0, indicating a steady non-increase in travel cost. However increasing travel cost adds to this value.

$$SOD = \sum_{i=1}^{N} max \Big[0, travelcost(i+1) - travelcost(i) \Big]$$

[4] It should be noted that assumptions should not be based solely on the results of these time weighted variants. By design they hide excessive travel costs during earlier trials and punish later ones severely. Therefore aggressive search variants such as SLA* significantly skew their reported results.

Table 3.6: The Travel Cost Stability During Convergence

		IAE (x10^3)	ISE (x10^9)	ITAE (x10^6)	ITSE (x10^9)	SOD (x10^3)
	FALCONS	19.8	5.281	0.723	11.567	**3.1**
	LRTA*	28.9	0.524	12.555	22.414	10.1
	SLA*	**17.2**	7.324	**0.017**	**7.324**	0.0
	SLA*T(100)	24.0	7.624	0.530	28.140	4.8
Baldur's Gate	SLA*T(1,000)	35.2	11.578	3.466	719.681	12.3
	SLA*T(10,000)	32.8	7.788	8.008	392.798	12.1
	δLRTA*-2	26.7	**0.375**	18.427	7.728	3.2
	δLRTA*-4	26.3	0.383	14.318	7.996	4.5
	δLRTA*-10	27.5	0.407	12.944	11.062	6.7
	FALCONS	13.3	0.131	0.265	1.019	4.7
	LRTA*	19.8	0.108	1.081	1.986	7.9
	SLA*	**10.9**	1.304	**0.011**	1.304	0.0
	SLA*T(100)	13.6	1.265	0.114	1.401	**1.8**
Gridworld	SLA*T(1,000)	18.0	1.461	0.296	2.831	5.3
	SLA*T(10,000)	20.0	1.351	0.772	12.940	6.1
	δLRTA*-2	15.6	**0.073**	1.921	**0.804**	2.4
	δLRTA*-4	16.8	0.077	1.360	0.903	4.1
	δLRTA*-10	18.6	0.088	1.145	1.188	6.5
	FALCONS	27.6	0.023	2.311	1.502	12.7
	LRTA*	67.2	0.036	15.622	7.096	32.7
	SLA*	**2.2**	0.013	**0.002**	**0.013**	0.0
	SLA*T(100)	144.8	0.446	21.999	58.209	95.2
8-puzzle	SLA*T(1,000)	71.6	0.068	16.061	12.026	37.3
	SLA*T(10,000)	67.2	0.036	15.622	7.096	32.7
	δLRTA*-2	22.0	**0.002**	6.399	0.519	**2.0**
	δLRTA*-4	35.3	0.006	10.230	1.340	6.6
	δLRTA*-10	53.7	0.016	13.962	3.578	18.7

Table 3.6 shows the stability data for the real-time algorithms. Both the ISE and ITSE values reported for SLA* in the pathfinding domains indicate the instability of its convergence process compared to LRTA*. In the Baldur's Gate domain, this is most apparent when punishing harder for excessive traveling later during the convergence process, as the ITSE metric does and in the Gridworld the ISE value shows the more overall fluctuations (due to SLA*'s first-trial). In the Baldur's Gate domain the ISE value for LRTA* is by far the lowest of all the algorithms. This indicates that LRTA* has the most stable overall trial cost during its convergence process, however when penalizing more for later fluctuations, seen in ITSE, LRTA* suffers for its longer convergence run. The ITSE values illustrate clearly how SLA* is benefitting from its excessive first-trial traveling. Interesting is to notice the SOD values for SLA*T(100) in the 8-puzzle domain. There it reports by far the worst value of all the methods indicating severe fluctuations in its travel cost. Further supporting our previous discussion regarding its excessive travel cost in this domain due to problem dependent T values.

Although most of the indices presented here still indicate that δLRTA* has a more stable travel cost between trials than LRTA*, these values do not indicate that the difference is as distinct as prior findings indicated. Taking the ITSE index in the Gridworld as an example, previously LRTA* was reported to have close to 6 times higher ITSE value than δLRTA*(2). Here however this difference is only 2.5 times higher. Corresponding difference can be observed throughout the pathfinding domains. However the results from the experiments on the 8-puzzle domain still report similar findings.

Contrary to the learning quality metric, the indices provide a way to represent numerically the amount of excess travel cost that each algorithm incurs during its search process. By punishing excess traveling in later trials such a value can be used to evaluate how much

Table 3.7: Average Total Memory Usage

	Baldur's Gate		**Gridworld**		**8-puzzle**	
Algorithm	Used	% of LRTA*	Used	% of LRTA*	Used	% of LRTA*
SLA*	**608**	55.1%	**517**	50.7%	**725**	2.6%
SLA*T(100)	826	74.8%	705	69.0%	36,789	131.3%
SLA*T(1,000)	988	89.4%	845	82.8%	25,002	89.3%
SLA*T(10,000)	1,030	93.2%	931	91.3%	24,266	86.6%
LRTA*	1,105	100.0%	1,020	100.0%	28,013	100.0%
FALCONS	1,529	138.4%	1,528	149.7%	27,314	97.5%

added travel cost improves the search process. These values should however be contrasted with the previously discussed metrics of learning quality and trials until convergence to provide the most accurate results. A serious drawback of these metrics is that they require the optimal solution (or a very accurate estimate) to be known. As a result the use of these measurement indices is limited to problems where obtaining an optimal solution is possible.

3.6 Memory Usage

The LRTA* based approach to solving real-time problems is based on the notion of storing and retriving updated heuristic values. To accommodate this a table is needed for administration of updated heuristic values. This table must also be stored in memory to maintain reasonable access times. Consequently the memory usage of real-time algorithms is very important and must be measured and managed closely.

Table 3.7 shows the average total memory usage of the real-time algorithms across all domains. *Memory Usage* is measured and presented here as the number of states stored in the algorithms heuristic table. In the case of algorithms that use additional tables their size is added to the algorithms final usage (these methods are discussed separately later in this section). Overall the memory usage of the algorithms in the pathfinding domains are negligible, indicating a low error-rate in heuristic values for these two domains. The memory usage in the sliding-tile puzzle is however more interesting. The SLA* algorithm has a small total memory footprint in both the 8- and 15-puzzle domains, using only a fraction of the memory required by LRTA* and other algorithms. It is also surprising is that even though FALCONS maintains a second table in memory its requirement is still 2.5% lower than that of LRTA* in the 8-puzzle. Indicating that the added g-values are indeed successful in helping the algorithm reduce wandering. This is especially interesting since FALCONS, in the 8-puzzle, stores only roughly half the number of h-values that LRTA* stores and the rest is used to store better g-value estimates. This shows that by storing extra g-values, FALCONS explores fewer states and improves its overall convergence process.[5] The first-trial table shown as Table 3.8 gives a better insight into the memory usage for the algorithms on a per-trial basis. The SLA*T algorithms outperform

[5] This indicates that by utilizing FALCONS way of minimizing its travel distance from the initial start state or SLA* backtracking technique the search stays relatively close to its initial start state. This clearly benefits the search considering that the optimal solution for the 8-puzzle domain is on average only 20 moves away from the start.

Table 3.8: Average First-Trial Memory Usage

	Baldur's Gate		**Gridworld**	
Algorithm	Used	% of LRTA*	Used	% of LRTA*
FALCONS	1,102	364.9%	741	311.3%
LRTA*	302	100.0%	238	100.0%
SLA*	608	201.3%	517	217.2%
SLA*T(100)	397	131.5%	304	127.7%
SLA*T(1,000)	290	96.0%	230	96.6%
SLA*T(10,000)	**256**	84.8%	**207**	87.0%
	8-puzzle		**15-puzzle**	
Algorithm	Used	% of LRTA*	Used	% of LRTA*
FALCONS	827	409.4%	645,249	4,597.4%
LRTA*	202	100.0%	**14,035**	100.0%
SLA*	724	358.4%	11,945,144	85,109.7%
SLA*T(100)	193	95.5%	1,971,970	14,050.4%
SLA*T(1,000)	**159**	78.7%	580,378	4,135.2%
SLA*T(10,000)	**159**	78.7%	86,289	614.8%

all other algorithms including LRTA* in the pathfinding domains. This holds also in the 8-puzzle, however the memory usage of SLA*T in the 15-puzzle is considerably higher than that of LRTA*. However when contrasting this with the solution lengths reported in Table 3.4 then the SLA*T algorithms while only using a fraction of the memory of LRTA* (except in the 15-puzzle) return much superior solutions overall.

3.6.1 Impact of Auxiliary Heuristic Functions

Table 3.9 shows the impact that secondary heuristic tables, used by both FALCONS and δLRTA*, have on their total memory usage. The table shows each algorithms heuristic table size, their total memory requirement and the increase in memory these tables have. The table reveals that increasing the δ bounds results in an increased exploration and consequently more memory usage. However, while FALCONS g-values only strictly double its memory usage the the same is not true for the δLRTA* algorithm. Previously δLRTA* was reported only doubling its memory requirement, however the numbers in Table 3.9 indicate that the increase is in-fact 5%-25% more in the 8-puzzle domain.

The additional memory usage suffered by δLRTA*, shown in Table 3.9, is used for storing its upper-bound values. Figure 3.4 shows this average memory usage over the algorithms entire convergence process. Although similar at the very beginning (around 2000 states traveled) the g-values quickly start consuming more memory than their h-value counter-

Table 3.9: Average Memory Overhead of Auxiliary Heuristic Functions

	Baldur's Gate			**Gridworld**			**The 8-puzzle**		
Algorithm	h-values	Total	Ratio	h-values	Total	Ratio	h-values	Total	Ratio
δLRTA*-2	984	2,078	2.1	873	1,834	2.1	7,409	18,074	2.4
δLRTA*-4	1,024	2,161	2.1	938	1,965	2.1	14,100	33,437	2.4
δLRTA*-10	1,066	2,247	2.1	999	2,087	2.1	22,266	51,663	2.3
FALCONS	765	1,529	2.0	764	1,528	2.0	13,657	27,314	2.0
LRTA*	1,105	1,105	1.0	1,020	1,020	1.0	28,013	28,013	1.0

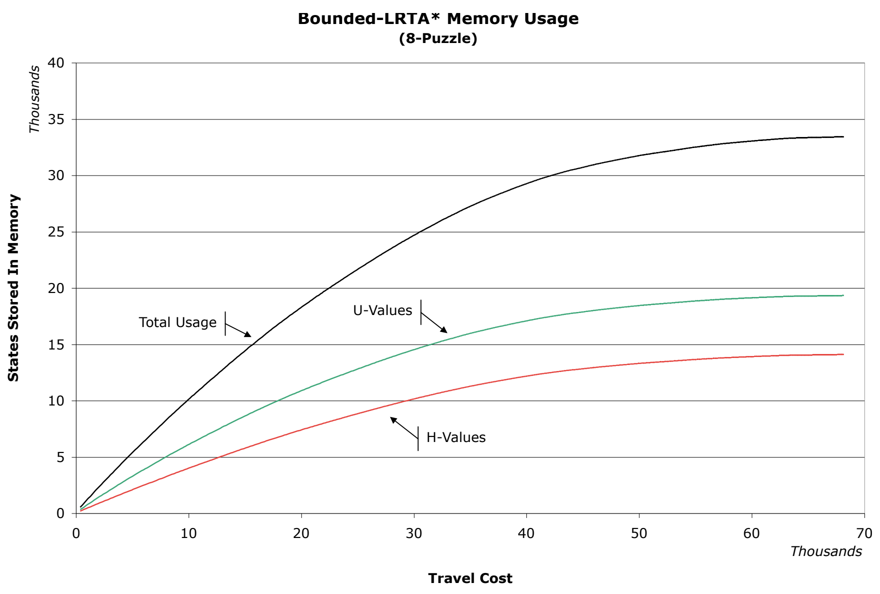


Figure 3.4: The memory usage of δLRTA* in the 8-puzzle domain
A comparison between the memory consumption of the δLRTA* method. The h-value and u-value usage is displayed separately along with the total memory usage.

parts. Although no explanation of this can be provided, one possible reason could be that the forward-propagation of upper-bounds that the algorithm performs (see Algorithm 3, lines 5 to 7), is forcing it to store updated upper-bound values for states that it never needs to visit. Although this increase in memory is only minimal for δLRTA* in the sliding-tile domain, this could have a more severe impact on memory usage when δLRTA* is run in domains with higher branching factors and less frequent transpositions (e.g. in Go).

In the past memory usage has not been considered a limiting factor in real-time search and rarely reported to any extent. This section showed that memory usage should however not be overlooked in circumstances where real-time algorithms are storing potentially large secondary data structures. Memory usage of real-time algorithms is clearly dependent on certain properties of the problem domains used, e.g. its branching factor, frequency of transpositions or the error-rate of its heuristic function. δLRTA* for example, was shown only to double its memory requirement in the pathfinding domains but in the 8-puzzle domain it uses more. This was partly explained as a result of the domain's branching factor coupled with low frequency of transpositions. As a result various domain factors must be carefully evaluated before storing additional state information.

3.7 Summary

This chapter presented empirical evaluation of the real-time algorithms discussed in the previous chapter. The chapter began by evaluating the most dominant factor of real-time search efficiency, travel cost, and ended with a discussion on their previously most overlooked one, namely memory usage.

Although the chapter primarily confirms previously reported findings it also presents several new interesting findings. For the first time, algorithms such as δLRTA* and FALCONS were empirically tested in a more realistic pathfinding environment such as the Baldur's Gate. There the δLRTA* algorithm exhibited significantly worse performance than FALCONS requiring close to 60% more travel cost. This contradicts the prior findings reported by Shimbo and Ishida. Also for the Gridworld the δLRTA* algorithm did not show as obvious travel cost improvements over FALCONS as previously reported. The FALCONS algorithm was for the first time contrasted with the performance of the SLA* and SLA*T algorithms. Although it did not match the performance of SLA*, it was shown to significantly improve upon the overall performance of the SLA*T variants in all test domains. A previously unobserved pathological problem inherent to SLA*T was illustrated and discussed. It was shown that the performance of SLA*T is highly dependent on domain specific values chosen for its T parameter. It can be argued that with highly tuned domain specific values of T SLA*T would do quite well, however finding these values is far from trivial.

The importance of loop elimination when comparing the solution quality of real-time algorithms was quantified. By eliminating loops, established algorithms are shown to provide significantly shorter and more stable solutions than previously reported. A new visual evaluation of the learning quality of intermediate algorithm trials was made possible by using the solution length as a function of the accumulated travel cost. By plotting this learning quality metric a better understanding of algorithm behavior was made possible. The number of trials to convergence was discussed as a way of enhancing the learning quality metric by giving an idea of how successful the algorithm is in distributing its learning process across consecutive trials. It was confirmed that, apart from SLA*, FALCONS converges in the fewest trials. However the performance of FALCONS during its earlier trials was show to be relatively poor compared to LRTA*'s. The five stability indices first used by Ishida and Shimbo where presented as a way of compressing the learning quality graphs into a single numerical value. Similar results, albeit less detailed, where shown to be readable from these indices as the learning quality graphs. In all domains the indices confirmed that δLRTA* is less prone to large fluctuations in its travel cost (indicated by its low ISE value) it however incurs more overall excess traveling due to its longer convergence process (indicated by its IAE value). Interesting to note is that in the Baldur's Gate domain δLRTA* incurs most of its excessive travel cost during the earlier trials (indicated by its ITSE value). Finally results contradicting prior research on the memory usage of δLRTA* were presented. They indicate that there is a correlation between the memory usage of δLRTA* in exponential domains such as the sliding-tile puzzle and the domain's branching factor. Higher branching factors coupled with restrictive bounds result in more excess memory usage.

A more detailed analysis of how these real-time algorithms perform in more varied and larger problem domains is needed. Further research is also needed to determine how different domain properties influence the properties and enhancements provided by the real-time algorithms.

Chapter 4

Back-Propagation

One of the main drawbacks of the LRTA* real-time heuristic search algorithm is slow convergence. Backtracking as introduced by SLA* is one way of speeding up the convergence, although at the cost of sacrificing first-trial performance. The backtracking mechanism of SLA* consists of back-propagating updated heuristic values to previously visited states while the algorithm retracts its steps. In this chapter these hitherto intertwined aspects are separated, and the benefits of each investigated independently. Two back-propagating search variants are presented that do value back-propagation without retracting their steps. Empirical evaluation shows that in some domains the value back-propagation is the key to improved efficiency while in others the retracting role is the main contributor.

4.1 Introduction

Learning Real-Time A* (Korf, 1990), or LRTA* for short, is probably the most widely known real-time search algorithm. A nice property of the algorithm is that it guarantees convergence to an optimal solution over repeated trials on the same problem instance (given an admissible heuristic). However, in practice, convergence to an optimal solution may be slow, both in terms of the number of trials required and the total traveling cost. Over the years researchers have proposed various enhancements aimed at overcoming this drawback. These improvements include: doing *deeper lookahead* (Russell & Wefald, 1991; Bulitko, 2004; Koenig, 2004), using *non-admissible heuristics*, although at the cost of forsaking optimality (Shimbo & Ishida, 2003; Bulitko, 2004), using more *elaborate successor-selection criteria* (Furcy & Koenig, 2000), or incorporating a *backtracking* mechanism (Shue & Zamani, 1993, 1999).

Backtracking affects the search in two ways. Firstly, backtracking algorithms may choose to retract to states visited previously on the current trial instead of continuing further along the current path, resulting in an exploration strategy that differs substantially from LRTA*'s. Secondly, during this retracting phase, changes in heuristic value estimates (h-values) get propagated further up the solution path (back-propagated). As these two aspects of backtracking have traditionally been closely intertwined in the real-time search literature, the importance of the role each plays is not clear. Back-propagation of learned values have previously been observed to improve the convergence of various real-time reinforcement learning algorithms (Lin, 1992; Sutton & Barto, 1998). Only recently has the propagation of heuristic values been studied in real-time search as a stand-alone procedure (Koenig, 2004; Hernández & Meseguer, 2005b, 2005a; Rayner et al., 2006), building in part on earlier observations by Russell and Wefald (Russell & Wefald, 1991) and the CRTA* and SLRTA* algorithms (Edelkamp & Eckerle, 1997).

It can be argued that in some application domains, value back-propagation cannot easily be separated from backtracking, since one must physically reside in a state to update its value. However, this argument does not apply to most agent-centered search tasks. For example, a typical application of real-time search is agent navigation in unknown environments. Besides the heuristic function used for guiding the search, the agent has initially only limited knowledge of its environment: knowing only the current state and its immediate successor states. However, as the agent proceeds with navigating the world it gradually learns more about its environment and builds an *internal model* of it. Because this model is kept in the agent's memory it is perfectly reasonable to assume that the agent may update the model as new information become available without having to physically travel to these states, for example to update the states' h-value. Consequently during the back-propagation, great care must be taken not to assume knowing successor information about states not visited before.

This chapter takes a closer look at the benefits of value back-propagation in real-time single-agent search. The main contributions of this chapter are:

1. New insights into the relative effectiveness of back-propagation versus backtracking in real-time search; in particular, in selected domains the effectiveness of backtracking algorithms is largely a side-effect of the heuristic value update, whereas in other domains their more elaborate successor-selection criterion is the primary contributor.

2. An algorithmic formulation of back-propagating LRTA* that assumes knowledge of only previously visited states; it outperforms LRTA* as well as its backtracking variants in selected application domains.

In the next section LRTA* is briefly explained and its most popular backtracking variants, using the opportunity to re-introduce the notation used throughout this thesis. The subsequent section provides a formulation of value back-propagating LRTA* variants that use information of only previously seen states. The result of evaluating these and other back-propagating and backtracking real-time search algorithms are reported in the empirical evaluation section. Finally the chapter is concluded and future work is briefly discussed.

4.2 Value Back-Propagation

SLA*'s backtracking mechanism serves two roles: firstly to back-propagate newly discovered information (in the form of updated h-values) as far back as possible and secondly

Algorithm 7 PBP-LRTA*

```
1:  $s \leftarrow$ initial start state $s_0$
2:  $solutionpath \leftarrow < empty >$
3:  **while** $s \notin S_g$ **do**
4:      $h'(s) \leftarrow min_{s' \in succ(s)}(c(s,s') + h(s'))$
5:      **if** $h'(s) > h(s)$ **then**
6:          update $h(s) \leftarrow h'(s)$
7:          **for all** states $s_b$ in LIFO order from the $solutionpath$ **do**
8:              $h'(s_b) \leftarrow min_{s' \in succ(s_b)}(c(s_b,s') + h(s'))$
9:              **if** $h(s_b) >= h'(s_b)$ **then**
10:                 break **for all loop**
11:             **end if**
12:             $h(s_b) \leftarrow h'(s_b)$
13:         **end for**
14:     **end if**
15:     push $s$ onto $solutionpath$
16:     $s \leftarrow argmin_{s' \in succ(s)}(c(s,s') + h(s'))$
17: **end while**
```

to offer the search the chance to reevaluate its previous actions given the new information. A natural question to ask is how important part each of the two roles plays in reducing the overall traveling cost. An algorithm that only performs the value back-propagation role

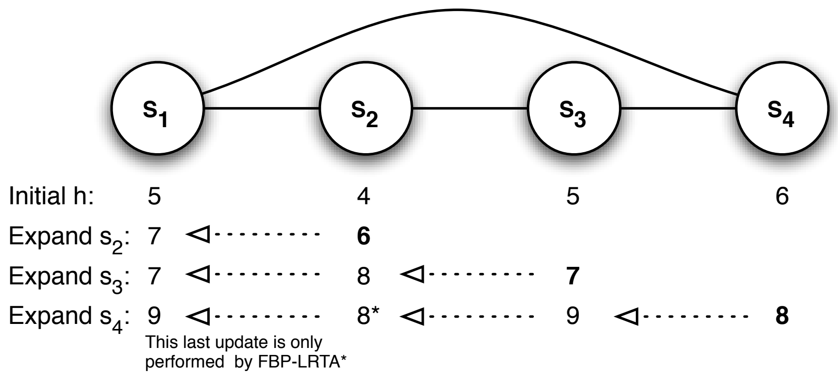

s1
s2
s3
s4
Initial h: 5 4 5 6
Expand s2: 7 ◁--------- 6
Expand s3: 7 ◁--------- 8 ◁--------- 7
Expand s4: 9 ◁--------- 8* ◁--------- 9 ◁--------- 8
This last update is only performed by FBP-LRTA*

Figure 4.1: Back-propagation stopping criteria example
The search starts in state s_1 and travels through to state s_4, the initial heuristic for each state is given in the first line below the image. Edges imply connectivity between states, all edges have unit-cost. (1) When state s_2 is first expanded its h-value is updated from 4 to 6. This update is then back-propagated and s_1 updated to 7. (2) Next s_3 is expanded, its h-value gets updated from 5 to 7 and back-propagation is again triggered. An important thing happens now when the back-propagation updates s_1 since the estimated h-value of s_4 determines that the h-value of s_1 becomes 7. (3) When the search finally expands state s_4 its h-value is updated to 8, the back-propagation phase then updates s_3 to 9. However s_2 does not require an update since it uses s_1 estimate and keeps its value of 8. Here our PBP-LRTA* terminates its back-propagation (the point is marked with an asterisks). However since s_1 h-value was based on a now outdated h-estimate of state s_4 it still needs updating. When running FBP-LRTA*, state s_1 however gets updated to a more correct value of 9.

of SLA*, not the backtracking itself, is illustrated as Algorithm 7. The algorithm, termed *partial back-propagating LRTA** (PBP-LRTA*), is identical to LRTA* except with additional code for back-propagating changes in heuristic values (lines 5 to 14). This code is invoked when a heuristic value is updated in the current state s. It back-propagates the heuristic values as far up the solution path as needed, before continuing exploration from state s in the same fashion as LRTA* does.

PBP-LRTA* stops the back-propagation once it reaches a state on the $solutionpath$ where the heuristic value does not improve. This may, however, be misguided in view of transpositions in the state space. It could be beneficial to continue the back-propagation further up the solution path even though no update takes place in a given state; an update might occur further up the path (Figure 4.1 shows an example illustrating this). An algorithm for doing full back-propagation is identical to PBP-LRTA* except that lines 9 to 11 are removed. We call such a search variant *full back-propagating LRTA**, or FBP-LRTA* for short. Since both PBP-LRTA* and FBP-LRTA* use the original LRTA* policies both for successor selection and heuristic value updating, they are expected to retain the same properties as LRTA* regarding completeness and optimality.

Although formulated very differently the FBP-LRTA* algorithm is closely related to the recently published LRTA*(∞) algorithm (Hernández & Meseguer, 2005b). For the special case of undirected state spaces both algorithms would update heuristic values in the same set of states. To retain the real-time requirement that FBP-LRTA* moves in constant time, the FBP-LRTA* algorithm could similarly be parameterized to be bounded by a constant.

4.3 Experimental Results

To investigate the relative importance of backtracking versus value back-propagation the PBP-LRTA* and FBP-LRTA* algorithms were empirically evaluated and contrasted with LRTA*, SLA* and SLA*T in three different domains. The domains and algorithm settings used for the experiments are the same as those previously described in Chapter 3.

Table 4.1 shows how the real-time search algorithms perform in the two path-finding domains. Reported for each algorithm are: their total travel cost, number of trials to convergence, first-trial travel cost, and solution length (with loops removed). Each number is the average over all test instances of the respective domain.

Both value back-propagation algorithms outperform LRTA* significantly in the pathfinding domains, converging faster in terms of both trials and travel cost. For example, FBP-LRTA* reduces the average number of trials to convergence on the Baldur's Gate maps by more than 100 (reduction of 62%). Its first-trial performance is also much better than LRTA*'s; an equally good solution is found on average using only a fraction of the search effort. Overall the FBP-LRTA* total traveling cost is roughly the same as SLA*'s, which is somewhat surprising because in the literature SLA* has been shown to consistently outperform LRTA*. The results indicate that the back-propagation of heuristic values, as opposed to backtracking, is largely responsible for improved performance

Table 4.1: Results from the pathfinding domains

Baldur's Gate Maps				
	Averaged Totals		First-trial	
	Travel Cost	Trials Conv.	Travel Cost	Sol. Len.
SLA*	17,374	1.81	17,308	71
FBP-LRTA*	19,695	63.40	508	89
SLA*T(100)	29,518	49.10	15,621	80
PBP-LRTA*	32,724	69.93	3,139	97
SLA*T(1000)	51,559	109.63	14,026	85
LRTA*	59,916	167.10	3,610	90
Gridworld with random obstacles				
	Averaged Totals		First-trial	
	Travel Cost	Trials Conv.	Travel Cost	Sol. Len.
FBP-LRTA*	8,325	35.32	389	102
SLA*	11,030	1.98	10,947	82
PBP-LRTA*	17,055	43.87	1,384	103
SLA*T(100)	17,705	46.67	9,223	91
SLA*T(1000)	24,495	69.90	8,404	97
LRTA*	29,760	90.67	2,237	102

Table 4.2: Results from the sliding-tile domains

8-puzzle				
	Averaged Totals		First-trial	
	Travel Cost	Trials Conv.	Travel Cost	Sol. Len.
SLA*	2,226	1.95	2,205	20
FBP-LRTA*	39,457	141.61	388	111
PBP-LRTA*	40,633	146.71	275	81
LRTA*	73,360	256.44	380	61
SLA*T(1000)	77,662	253.99	380	61
SLA*T(100)	149,646	202.77	651	41

in pathfinding domains. Furthermore, FBP-LRTA* achieves this, unlike SLA*, while keeping its real-time characteristics by amortizing the learning over many trials. The value back-propagation algorithms successfully combine the good properties of SLA* and LRTA*: SLA*'s short travel cost and fast convergence and LRTA*'s short first-trial delay and iterative solution approach.

Table 4.2 gives the same information for both sliding-tile puzzles. In the 8-puzzle domain SLA* is clearly superior to the other algorithms when evaluated by total travel cost. This result is consistent with what has been reported in the literature (Bulitko & Lee, 2006). In this domain backtracking, as opposed to only doing value back-propagation, is clearly advantageous. Also of interest is that FBP-LRTA* and PBP-LRTA* perform almost equally well, contrary to the pathfinding domains where FBP-LRTA* was superior. This can be explained by the fact that there are relatively few transpositions in the sliding-tile-puzzle domain. Thus, the benefit of continuing the back-propagation is minimal. Also, somewhat surprisingly the first-trial cost of PBP-LRTA* is superior to FBP-LRTA*. No solid explanation for this is available at this time. Results from the 15-puzzle domain show that

Table 4.3: Bounding the back-propgation phase of FBP-LRTA* and PBP-LRTA*.

Gridworld								
	Bound (k)	k=5	k=10	k=50	k=100	k=200	k=500	Unbounded
Travel Cost	FBP-LRTA*	16,492	14,493	12,070	11,305	10,536	9,499	8,325
	PBP-LRTA*							17,055
	LRTA*							29,760
Ave. Move Delay	FBP-LRTA*	0.0049	0.0077	0.0280	0.0488	0.0855	0.1726	0.4275
	PBP-LRTA*							0.0036
	LRTA*							0.0013
8-Puzzle								
	Bound (k)	k=5	k=10	k=50	k=100	k=200	k=500	Unbounded
Travel Cost	FBP-LRTA*	43,466	40,046	38,964	39,041	38,946	39,139	39,457
	PBP-LRTA*							40,633
	LRTA*							73,360
Ave. Move Delay	FBP-LRTA*	0.0146	0.0215	0.0746	0.1323	0.2198	0.4170	0.6077
	PBP-LRTA*							0.0110
	LRTA*							0.0061

the overall benefits of value back-propagation are relatively small. SLA* was the only algorithm that was successful for that puzzle size. It converged in 95 out of the 100 15-puzzle instances using a limit of 50 million states traveled, whereas the other algorithms all failed to converge even on a single problem.

4.4 Bounding the Back-Propagation

The evaluation of both the FBP-LRTA* and the PBP-LRTA* algorithms in this chapter has focused on showing how much the back-propagation of heuristic values can improve performance. However, to ensure that the algorithms retain their real-time property, of constant deliberation time, the back-propagation phase must be upper-bounded by a constant k. The k parameter defines the distance (in number of states) from the current state that values will be back-propagated. Only states on the current trial's solution path leading away from the current state are considered for updating. The implementation of a constant k into the back-propagation extensions previously presented in this chapter is relatively simple.

Table 4.3 presents data for the bounded versions of both the FBP-LRTA* and PBP-LRTA* algorithms. Both algorithms were run with 6 different values of k: 5, 10, 50, 100, 200 and 500 respectively, in addition to the unbounded version. The PBP-LRTA* algorithm never hit the lowest upper-bound limit, therefore only the value of the unbounded version of PBP-LRTA* is presented. Two metrics are presented for each testbed: the total average travel cost and the average move delay. Travel cost is measured as the number of states expanded and the move delay is measured in milliseconds. All times were taken on a cluster of Dual Intel P4 Xeon 3GHz CPU machines, the program was compiled using the Linux gcc 4.0 compiler. The results from the Baldur's Gate domain were similar to that from the Gridworld, therefore only the Gridworld is presented in Table 4.3. The only no-

Table 4.4: Total Time to Convergence
(All times are in milliseconds)

	8-puzzle	% of LRTA*	**Gridworld**	% of LRTA*	**Baldur's Gate**	% of LRTA*
LRTA*	**378.3**	100.0%	101.1	100.0%	1,148.6	100.0%
PBP-LRTA*	425.0	112.3%	**90.6**	89.7%	**810.6**	70.6%
FBP-LRTA*	12,682.4	3,352.6%	1,587.5	1,570.7%	9,545.1	831.0%

table difference is that the move-delay metrics for the unbounded FBP-LRTA* algorithm are slightly higher in the Baldur's Gate due to longer solution paths.

The data confirms that any additional back-propagation results in lower overall travel cost but at a cost of increased deliberation time. In the Gridworld the FBP-LRTA* algorithm steadily improves its performance over that of PBP-LRTA* as k is increased. The resulting increase in move delay is close to linear in the increase of k. Consequently back-propagating further results in lower overall travel costs while increasing the algorithms deliberation time. The 8-puzzle shows however that PBP-LRTA* which never back-propagates more than 2 states performs better overall than FBP-LRTA* with a $k = 5$ which is consistent with previous statements that back-propagation seems to be far less effective in this domain. Additionally the 8-puzzle exhibits much higher move delays than the pathfinding domains for the same values of k. This likely due to the search performing more heuristic updates in the 8-puzzle than in the Gridworld (saving and updating values requires more instructions than just comparing).

Table 4.4 shows the total time (in milliseconds) that it took the unbounded versions of the algorithms to converge. Here the advantage that back-propagation has in the pathfinding domains can be observed. There the PBP-LRTA* algorithm, which never reaches the lowest back-propagation limit of 5, has considerable advantage over the LRTA* algorithm, requiring 10%-30% less total time until convergence.

4.5 Summary

This chapter studied the effectiveness of backtracking versus back-propagation in selected single-agent search domains. The good performance of backtracking algorithms like SLA* has often been contributed to their more elaborate successor-selection criteria. This was shown not to be true in general. For example, in pathfinding domains the performance improvement is mainly due to the effects of back-propagating updated heuristics, not the backtracking. The FBP-LRTA* search variant exhibited the best overall performance of all the real-time search algorithms experimented with (not counting the SLA* algorithm). Furthermore, in this domain the back-propagation variants successfully combine the nice properties of SLA* and LRTA*: SLA*'s low travel cost and LRTA*'s short first-trial delay and iterative solution approach.

On the other hand, contrary to the pathfinding domains, back-propagation is much less effective in the sliding tile puzzle. It showed some benefits on the 8-puzzle, but results on the 15-puzzle show diminishing contribution. The different exploration criterion used by backtracking seems to be have far more impact than value updating. This poses an interesting research question of what properties of a problem domain favor backtracking versus value back-propagation. The complexity of the domain and the frequency of transpositions is suspected to be in part responsible, but based on the available evidence it is too premature to speculate much at this stage and this is therefore left for future research.

There is still additional work that needs to be done to understand the mutual and separate benefits of back-propagation and backtracking better. Such investigation opens up many new interesting questions. There is clearly scope for new search variants that better utilize the benefits of both approaches by adapting to different problem domains. The next chapter introduces one such variant, built on the FBP-LRTA* algorithm but attempting to augment it with an optional backtracking stage.

Chapter 5

Backtracking

This chapter introduces a new real-time search variant based on the FBP-LRTA* algorithm introduced in the previous chapter. The new algorithm augments its value back-propagation phase with an optional backtracking mechanism. The new algorithm is a hybrid method between the LRTA* and SLA* algorithms. The new method is able to make better decisions than the current SLA*T hybrid, regarding whether backtracking is more efficient than continuing forward from the current state. The new hybrid method successfully improves the poor performance of FBP-LRTA* in the sliding-tile puzzle. However the backtracking phase is shown to have counter-productive effects on the algorithms performance in the pathfinding domains.

5.1 Introduction

Backtracking was first introduced to the field of real-time search with the SLA* algorithm (Shue & Zamani, 1993). The SLA* algorithm and its SLA*T variant are the only strictly optimal seeking real-time methods that include a backtracking extension. However, as previously witnessed, they both have serious real-time flaws: SLA* with its extremely poor first-trial performance and SLA*T's parameterization being highly problem dependent.

As shown in the previous chapter, to facilitate a change in the successor selection then backtracking must be coincided with value back-propagation. However the chapter showed how back-propagation can be done independently of backtracking. This chapter on the other hand, shows how an independent back-propagation technique can be used to enable a more informed backtracking. Therefore the new backtracking algorithm introduced in this chapter will augment the previously described FBP-LRTA* algorithm with an optional backtracking phase. This new algorithm is able to make more intelligent choices regarding whether and then how far it should backtrack. By separating these two phases the algorithm improves the stability of the search process from that of the other backtracking variants and boosts the performance of FBP-LRTA* in exponential domains such as the sliding-tile puzzle.

The main contributions of this chapter are:

1. New insights into how information learned through back-propagation can be utilized to make better backtracking choices.
2. An algorithmic formulation of backtracking FBP-LRTA*, that clearly separates its value back-propagation and bactracking
3. An improved way of amortizing backtracking over consecutive trials; eliminating the need for a user-definable control parameter.

Section 5.2 provides a formulation of the enhanced backtracking LRTA* variant. Its evaluation results are contrasted with other common backtracking real-time search algorithms in Section 5.3. Finally the chapter is concluded and future work discussed in Section 5.5.

5.2 Enhanced Backtracking

Algorithm 8 EB-LRTA*

```
1: $s \leftarrow$ initial start state $s_0$
2: $solutionpath \leftarrow < empty >$
3: while $s \notin S_g$ do
4:     $S_{bt} \leftarrow \emptyset$
5:     $h'(s) \leftarrow min_{s' \in succ(s)}(c(s,s') + h(s'))$
6:     if $h'(s) > h(s)$ then
7:         update $h(s) \leftarrow h'(s)$
8:         $s_{prev} \leftarrow s$
9:         for all states $s_b$ taken in LIFO order from the $solutionpath$ do
10:            $s_{curr} \leftarrow argmin_{s' \in succ(s_b)}(c(s_b,s') + h(s'))$
11:            $h(s_b) \leftarrow h(s_{curr}) + c(s_b, s_{curr})$
12:            if $s_{prev} \in succ(s_b)$ and $s_{prev} \neq s_{curr}$ then
13:                $S_{bt} \leftarrow S_{bt} \cup \{s_b\}$
14:            end if
15:            $s_{prev} \leftarrow s_b$
16:        end for
17:    end if
18:    push $s$ onto $solutionpath$
19:    $S_{bt} \leftarrow S_{bt} \cup \{argmin_{s' \in succ(s)}(c(s,s') + h(s'))\}$
20:    $s \leftarrow argmin_{s' \in S_{bt}}(h(s'))$ – breaking ties towards the lowest $c(s,s')$
21: end while
```

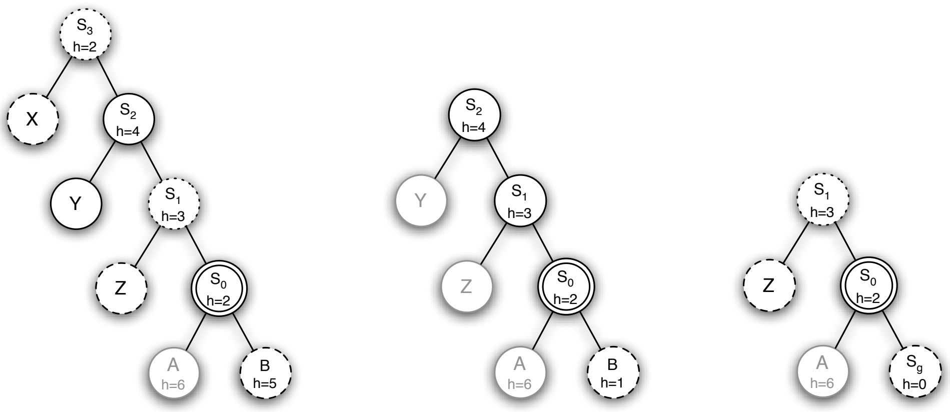
The new backtracking version of the FBP-LRTA* algorithm is presented as Algorithm 8. The algorithm, termed Enhanced Backtracking LRTA* (EB-LRTA* for short), uses the value back-propagation method of FBP-LRTA* but enhances it with an optional backtracking phase. The backtracking phase of the algorithm requires, as SLA*, a bi-directional state space (i.e. that all actions are retractable) however the back-propagation phase only requires that all reachable states from the initial start state s_0 lie on a path to one of its goal states (i.e. that the algorithm never enters dead-ends).

The algorithm has two main facets:

1. When an heuristic update occurs it back-propagates the update through all the states on its current $solutionpath$. (lines 9 to 16).
2. While back-propagating, the algorithm stores information on states where updated heuristic values cause a different successor selection than before (lines 12 to 14).

When an h-value update occurs in state s (line 7) the algorithm starts its back-propagation phase. It cycles through all traversed states during the current trial updating under-estimated h-values. The algorithm additionally checks if the updated heuristic value causes a change in the earlier successor selection (line 12). If such a change occurs the state is stored in S_{bt} as a backtrack candidate (line 13).

The algorithm is now given a choice of backtracking to a previous state (chosen from the backtrack set S_{bt}) or continuing forward from its current state. Then the algorithm greedily selects which state to move to from this set based on the state's heuristic value



(a) Available Backtracking Options

(b) No Forced Backtracking

(c) No Unnecessary Backtracking

Figure 5.1: Trade-off when selecting a backtrack state

The figures above each show a part of an EB-LRTA* state-tree after the back-propagation phase has concluded. In all the figures S_0 is the current state, dotted border indicate a previously visited state where a heuristic update caused a change in its successor selection, a dashed state indicates a current minimum successor.

In Figure 5.1(a) state S_2 did not get updated during the back-propagation phase. In this case the SLA* algorithm would backtrack to S_2 where it would stop and continue its LRTA* search from there on. EB-LRTA* could however choose to backtrack to state S_3 because it has the lowest heuristic value.

In Figure 5.1(b) none of the S_n states changed their successor selection choices after updating. Therefore EB-LRTA* is not offered any backtracking states and it immediately continues forward to its successor B. Here SLA* would be forced to backtrack all the way up to state S_2 and then revisit state S_1 and S_0.

Figure 5.1(c) demonstrates a case where EB-LRTA* decides to continue forward despite having detected possible backtrack states. In this case the goal state is the next current successor and has the lowest heuristic value. Therefore EB-LRTA* will decide to move forward although it is presented with S_1 as a possible backtrack state.

Table 5.1: Results from the sliding-tile domains

8-puzzle				
	Averaged Totals		First-trial	
	Travel Cost	Trials Conv.	Travel Cost	Sol. Len.
SLA*	2,226	2.0	2,205	20
EB-LRTA*	26,300	107.1	413	58
FBP-LRTA*	39,457	141.6	388	111
LRTA*	73,360	256.4	380	61
SLA*T(1,000)	77,662	254.0	380	61
15-puzzle				
	Averaged Totals		First-trial	
	Travel Cost	Trials Conv.	Travel Cost	Sol. Len.
SLA*	29,705,920	2.0	29,705,867	52
FBP-LRTA*	>50,000,000	n/a	3,914	687
EB-LRTA*	>50,000,000	n/a	4,725	235
LRTA*	>50,000,000	n/a	22,779	813
SLA*T(1,000)	>50,000,000	n/a	1,228,778	209

(line 20) breaking ties toward the state closest to its current residing state. If tied, the algorithm favors the lowest cost successor of the current state. The actions performed when backtracking are all counted towards the algorithms travel cost. The algorithm, by adding the current state's lowest cost successor to the S_{bt} set or favoring it when breaking ties, tempers the aggressiveness of the search's backtracking, by allowing the search to continue forward from its current state instead of forcing it to backtrack. This policy is illustrated as Figure 5.1. For the same reasons that SLA*'s backtracking phase is not applicable in directed domains, the EB-LRTA* algorithm is not fully applicable there either. On the other hand, contrary to SLA*, when the EB-LRTA* algorithm is unable to backtrack it is still able to back-propagate heuristic information, reverting completely to its FBP-LRTA* ancestor.

Intuitively backtracking is only needed when different choices are present from its previously traversed path. By only considering states for backtracking where back-propagation resulted in a successor selection change, EB-LRTA* only considers backtracking when faced with different choices. Otherwise the search reverts to its default LRTA* successor selection policy. Not only does this result in a more effective backtracking strategy but also eliminates the need for an user-definable backtracking control parameter such as SLA*T's T parameter. Consequently EB-LRTA* forms a better combination of backtracking and value back-propagation than previous hybrid methods such as the SLA*T algorithm.

5.3 Experimental Results

The EB-LRTA* algorithm was empirically evaluated and contrasted with FBP-LRTA*, LRTA*, SLA* and SLA*T (with T value of 100, 1,000 and 10,000; For each domain the T value that performed the best was chosen for comparison). All algorithm settings and domains used for the experiments are otherwise the same as in Chapter 3.

Table 5.1 reports the algorithm's performance in the sliding-tile domain. For each algorithm its total travel cost, number of trials to convergence, first-trial travel cost, and solution length (with loops removed) is reported. Each number is the average over all

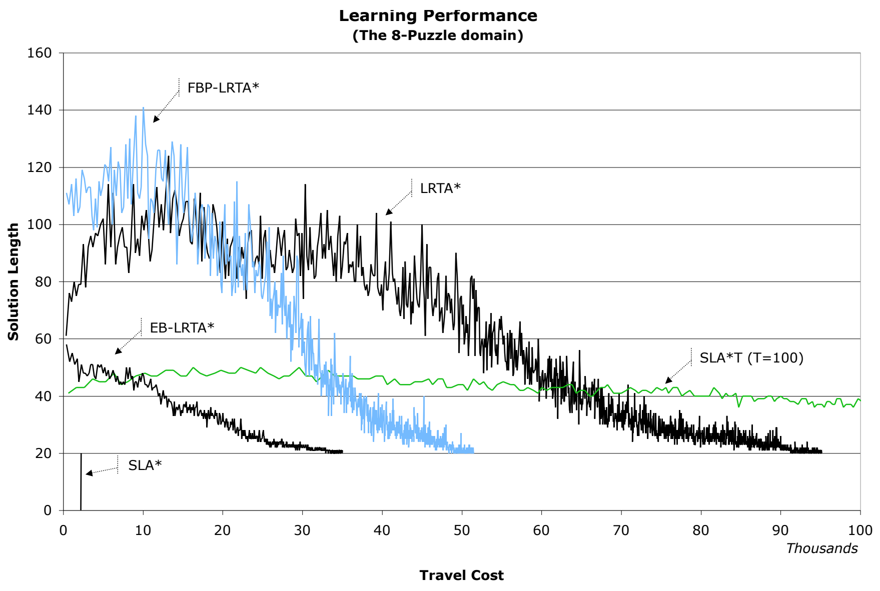


Figure 5.2: Learning Performance of EB-LRTA* compared to FBP-LRTA*, LRTA*, SLA* and SLA*T(100) in the 8-piece version of the sliding-tile puzzle.

test instances of the respective domain. Since none of the comparison algorithms, except SLA*, converged on any of the 15-puzzle problems within a travel cost of 50 million states only the first-trial statistics are provided for that domain.

The data shows that when solving the 8-puzzle EB-LRTA* significantly improves the performance of the FBP-LRTA* algorithm, having both less total travel cost and fewer trials to convergence. During its first-trial EB-LRTA* has a marginally higher travel cost than FBP-LRTA*, but produces much better solutions. This is even more evident in the 15-puzzle, where EB-LRTA* finds a first-trial solution that is only a third of that found by FBP-LRTA*. EB-LRTA* achives this while only requiring a relatively small overhead in travel cost.

Figure 5.2 shows the learning performance of the EB-LRTA* algorithm in the 8-puzzle domain. The graph demonstrates clearly EB-LRTA*'s lower travel cost requirement. More importantly, the graph shows EB-LRTA*'s excellent real-time nature. Namely how quick it is to produce a relatively good first-trial solution and then effectively utilizes additional travel to further improve its solution. EB-LRTA* short trial delay is also evident in this graph, illustrated by the compactness of the plotted line. Similarly Figure 5.3 shows EB-LRTA* learning performance in the 15-puzzle domain. Although each algorithm was restricted to only 500 trials, the results give a good indication of their overall performance. The graph in the figure shows that EB-LRTA* has a shorter trial delay than the other algorithms (closely comparable to LRTA*) and is able to produce a fair first-trial solution relatively fast. Its short trial delay is clearly evident since it has performed close to 350 trials (out of 500) before SLA*T(1,000) is able to return an initial solution.

Table 5.2 shows the same information as presented Table 5.1 only now for the pathfinding domains. In these domains, FBP-LRTA* previously exhibited a much better performance than any of the other algorithms, both during its first-trial and in total. Table 5.2 shows however that the backtracking phase of EB-LRTA* is counterproductive to the search's

Table 5.2: Results from the pathfinding domains

Baldur's Gate Maps				
	Averaged Totals		First-trial	
	Travel Cost	Trials Conv.	Travel Cost	Sol. Len.
SLA*	17,374	1.8	17,308	71
FBP-LRTA*	19,695	63.4	508	89
EB-LRTA*	26,669	62.7	3,158	89
SLA*T(100)	29,518	49.1	15,621	80
LRTA*	59,916	167.1	3,610	90
Gridworld with random obstacles				
	Averaged Totals		First-trial	
	Travel Cost	Trials Conv.	Travel Cost	Sol. Len.
FBP-LRTA*	8,325	35.3	389	102
SLA*	11,030	2.0	10,947	82
EB-LRTA*	11,332	35.6	1,146	97
SLA*T(100)	17,705	46.7	9,223	91
LRTA*	29,760	90.7	2,237	102

success when compared to FBP-LRTA*, since EB-LRTA* increases both its total and first-trial travel cost. As expected, value back-propagation is by far the best search extension for pathfinding domains.

The stability of the travel cost incurred by EB-LRTA* during its convergence process is presented in Table 5.3. This table lists the convergence stability indices formerly described in Section 3.5. They confirm that EB-LRTA* significantly improves the travel cost stability of FBP-LRTA* in the 8-puzzle domain. While both the IAE and ISE indices for EB-LRTA* indicate a much milder over-shots in travel cost its SOD value shows how effectively EB-LRTA* reduces wandering. Although EB-LRTA* shows an overall improved performance over the comparison algorithms in the pathfinding domains, its backtracking phase clearly causes an increase in excess travel cost from that of FBP-LRTA*.

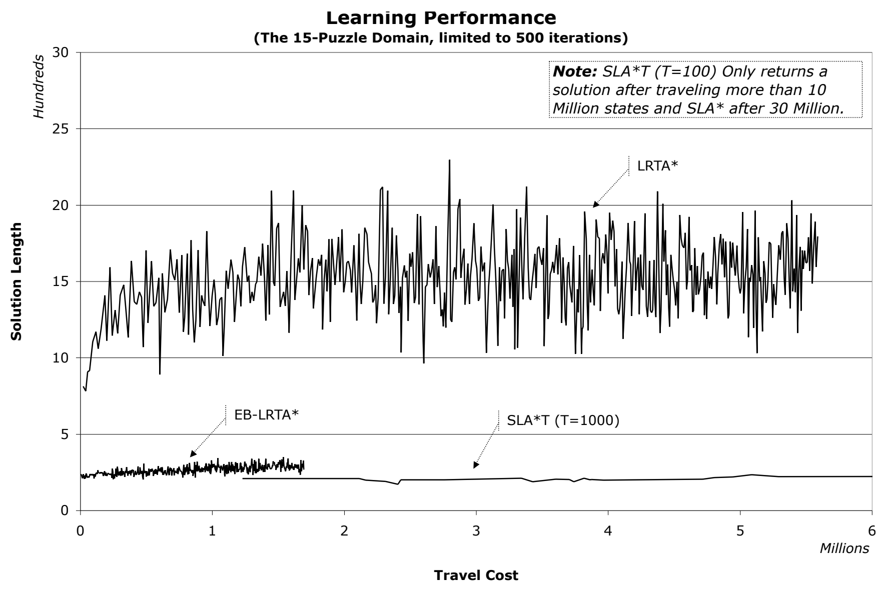


Figure 5.3: Learning Performance of EB-LRTA* compared to LRTA* and SLA*T(1,000) in the 15-puzzle. The algorithms were limited to only 500 trials each (due to difference in travel cost per trial the algorithms incur different total travel costs during the 500 trials). FBP-LRTA* was excluded to improve readability, its performance nearly identical to that of LRTA*.

Table 5.3: Stability of Travel Cost During Convergence for EB-LRTA*

		IAE (x10^3)	ISE (x10^9)	ITAE (x10^6)	ITSE (x10^9)	SOD (x10^3)
	FBP-LRTA*	**8.1**	**0.018**	2.103	**1.138**	**3.4**
	EB-LRTA*	15.4	0.342	2.447	4.382	6.2
Baldur's Gate	LRTA*	28.9	0.524	12.555	22.414	10.1
	SLA*	17.2	7.324	**0.017**	7.324	0.0
	SLA*T(100)	24.0	7.624	0.530	28.140	4.8
	FBP-LRTA*	**4.5**	**0.005**	0.112	**0.067**	2.0
	EB-LRTA*	7.4	0.032	0.183	0.246	3.0
Gridworld	LRTA*	19.8	0.108	1.081	1.986	7.9
	SLA*	10.9	1.304	**0.011**	1.304	0.0
	SLA*T(100)	13.6	1.265	0.114	1.401	**1.8**
	FBP-LRTA*	36.1	0.019	4.778	2.181	17.6
	EB-LRTA*	23.7	**0.010**	2.435	0.875	**10.7**
8-puzzle	LRTA*	67.2	0.036	15.622	7.096	32.7
	SLA*	**2.2**	0.013	**0.002**	**0.013**	0.0
	SLA*T(1,000)	71.6	0.068	16.061	12.026	37.3

When developing the original idea behind the EB-LRTA* algorithm it was intended to greedily select a state from the backtrack set (see Algorithm 8, line 20) using the sum of the backtrack state's heuristic value and the cost of backtracking to that state from the current state or $argmin_{s' \in S_{bt}}(h(s') + c(s, s'))$. However in environments that have uniform state-transition costs, this cost function results in no backtracking actions being performed. Adapting the EB-LRTA* algorithm to domains with variable state-transition costs is left for future work.

Experimentation in using alternative tie-breaking rules during the backtracking phase, resulted in no significant performance improvements. When breaking ties towards the backtrack state furthest away from the current state, e.g. the highest $c(s, s')$ (see Algorithm 8 line 20) no noteworthy improvements could be observed, neither in total travel cost nor in convergence stability. Primarily the experiments indicate that backtracking further away from the current state results in a reduction in trials required until convergence and somewhat shorter solutions. This however comes at the cost of a significant increase in trial delay (particularly during the first-trial).

5.4 Bounding EB-LRTA*'s Back-Propagation

As previously discussed in Chapter 4, the back-propagation phase of the FBP-LRTA* algorithm must be bounded to ensure constant deliberation time. Since EB-LRTA* builds on the FBP-LRTA* algorithm, similar back-propagation bounds can be put in place.

Table 5.4 presents data for the bounded versions of EB-LRTA*. The algorithm was run with 6 different values of *k*: 5, 10, 50, 100, 200 and 500 respectively, in addition to the unbounded version. Two metrics are presented for each testbed: the total average travel cost and the average move delay. Travel cost is measured as the number of states expanded and the move delay is measured in milliseconds. The experiments were run on a Dual Intel P4 Xeon 3GHz CPU machine and the code was compiled using the Linux gcc 4.0 compiler. The results from the Baldur's Gate domain are omitted from Table 5.4, since they provide similar information as the Gridworld. The only significant difference

Table 5.4: Bounding the back-propgation phase of EB-LRTA*.

Gridworld								
	Bound (k)	k=5	k=10	k=50	k=100	k=200	k=500	Unbounded
Travel Cost	FBP-LRTA*	16,492	14,493	12,070	11,305	10,536	9,499	8,325
	EB-LRTA*	15,043	12,567	11,334	11,333	11,333	11,333	11,333
	LRTA*							29,760
Ave. Move Delay	FBP-LRTA*	0.0049	0.0077	0.0280	0.0488	0.0855	0.1726	0.4275
	EB-LRTA*	0.0048	0.0073	0.0248	0.0401	0.0503	0.0552	0.0507
	LRTA*							0.0013
8-Puzzle								
	Bound (k)	k=5	k=10	k=50	k=100	k=200	k=500	Unbounded
Travel Cost	FBP-LRTA*	43,466	40,046	38,964	39,041	38,946	39,139	39,457
	EB-LRTA*	29,937	26,896	26,310	26,310	26,300	26,300	26,300
	LRTA*							73,360
Ave. Move Delay	FBP-LRTA*	0.0146	0.0215	0.0746	0.1323	0.2198	0.4170	0.6077
	EB-LRTA*	0.0098	0.0158	0.0464	0.0570	0.0586	0.0605	0.0565
	LRTA*							0.0061

is that the move-delay for the unbounded FBP-LRTA* algorithm is higher in the Baldur's Gate due to longer solution paths.

EB-LRTA* shows diminishing travel cost improvements beyond a bound of $k = 10$ in both the Gridworld and the 8-puzzle domain. EB-LRTA* has a noticeably good first-move delay and average total move-delay in both domains compared to FBP-LRTA*. While FBP-LRTA* has a move-delay of roughly a factor of 100 compared to LRTA*, the move-delay of the EB-LRTA* compared to LRTA* is only around a factor of 10. This difference is most obvious in the 8-puzzle domain, though it is also clearly noticeable in both pathfinding domains. A reason for this is that while FBP-LRTA*'s $solutionpath$ strictly grows in length as the search travels through the problem space, resulting in increasingly longer back-propagation, the EB-LRTA* algorithm regularly removes states from its $solutionpath$ decreasing its length, thus requiring shorter back-propagation than FBP-LRTA*. Interesting to note is that Table 5.4 indicates that the EB-LRTA* algorithm has no noticeable travel cost improvements beyond a bound of $k = 10$. Also, when $k >= 50$ the algorithm does not exhibit any significant increase in either its first-move delay or its average move delay.

Even when bounded by low k-values the EB-LRTA* algorithm maintains most of the efficiency of its unbounded counterpart. This makes the EB-LRTA* a good practical real-time algorithm.

5.5 Summary

This chapter demonstrated the performance improvement gained through backtracking in certain domains. It built on the findings on value back-propagation obtained from Chapter 4 to illustrate how backtracking can be used to improve both the overall and first-trial performance for problems where pure value back-propagation is less effective.

The EB-LRTA* algorithm was introduced which extends the FBP-LRTA* algorithm by augmenting it with an optional backtracking step. By clearly separating the value back-propagation and action backtracking the algorithm is able to intelligently decide to back-track only when it is likely to yield a better result.

The algorithm was shown to perform very well in the sliding-tile domain, significantly improving upon the performance of FBP-LRTA* in that domain. The algorithm achieved an excellent overall first-trial and total performance, requiring lower travel cost and showing more stable convergence process. On the other hand, the results from the pathfinding domains show that while back-propagation does very well, the introduction of a backtracking step can have a detrimental impact on the algorithm performance. This somewhat indicates that backtracking and value back-propagation are, in some domains, counter-productive to each other. Although not conclusive, there is a strong indication that the structure of the problem domain determines which of them is appropriate. Therefore while augmenting value back-propagation with backtracking for exponential domains such as the sliding-tile puzzle is clearly beneficial, the same is not true for domains such as the pathfinding domains that have a high amount of state transpositions.

Compared to the SLA*T algorithm, EB-LRTA* shows both better performance and real-time nature and forms a more successful hybrid algorithm of the greedy LRTA* successor selection and the backtracking of SLA*. Future work include a more detailed evaluation of alternative tie-breaking policies during the backtracking phase and a more refined control of how and when backtracking is performed. The performance of EB-LRTA* in variable cost environments is also largely unknown and how different state transitional costs affect the selection of backtrack states. There is still much work to be done in extracting information from the backtracking and back-propagation search extensions. Such research opens up many new interesting questions regarding what information other real-time search extension have to offer each other.

Chapter 6

Conclusions

This chapter summarizes the main contributions of this thesis and describes several avenues for future work.

6.1 Conclusion

The main contributions of the research presented in this thesis are threefold: extensive comparison of optimal seeking real-time algorithms, the separation of value back-propagation from backtracking, and the introduction of the enhanced backtracking real-time search algorithm. This section summarizes the individual contributions of each of these three parts.

The comparison research presented in Chapter 3 sheds new light on the real-time nature of several popular algorithms and their relative performance in a diverse set of testbeds. It unified earlier comparison research on disjoint set of real-time algorithms (Shimbo & Ishida, 2003; Bulitko & Lee, 2006). The comparisons were based on a unified implementation of both the algorithms themselves and in a varied set of testbeds. All the algorithms were tested using the same initial problem instances. The research while primarily confirming earlier results also shed new light on various unknown algorithm properties. The chapter introduced a new simple visual method of evaluating the learning performance of real-time algorithms during their convergence process. This method of evaluation makes it easier to judge the real-time nature of algorithms and the stability and effectiveness of their convergence process. The chapter also quantified the effect that loop elimination has on the reported solution quality of real-time algorithms and advocated for loop removal to ensure fair comparisons.

Chapter 4 demonstrates that the good performance of backtracking algorithms, such as SLA*, in certain domains is not primarily due to their different successor selection policy. In-fact the value back-propagation phase contributes significantly more in certain domain types (such as the pathfinding domains used in this thesis). A simple extension to the LRTA* algorithm that only performs value back-propagation was introduced. An important design element of the new algorithm is that it does not assume knowledge of unexplored successor states during updating of heuristic values. The algorithm, termed FBP-LRTA*, was empirically evaluated and its performance contrasted with other real-time algorithms. FBP-LRTA* exhibited excellent performance in the pathfinding domains and demonstrated that back-propagation clearly outperforms backtracking in those domains. However, while improving the performance of LRTA*, FBP-LRTA* did not show the same effectiveness in the sliding-tile domain.

EB-LRTA* is the first real-time algorithm that utilizes backtracking but clearly separates it from its back-propagation of heuristic values. The algorithm is based on the FBP-LRTA* algorithm. The separation of back-propagation and backtracking enables EB-LRTA* to do more informed backtracking. It significantly improves upon the overall performance of LRTA* and the SLA*T algorithms in all the test domains. More importantly it was more successful in improving the trial delay (especially the first-trial) than previous backtracking algorithms. EB-LRTA* was intended to improve the relative poor performance of the FBP-LRTA* algorithm in domains such as the sliding-tile puzzle. While successfully attaining its goal the additional backtracking phase however degrades the performance of the FBP-LRTA* algorithm in the pathfinding domains. This finding shows that combining backtracking and back-propagation are to some extent counter-productive to each other.

6.2 Future Work

During the experimental investigation of this thesis, some interesting phenomena were observed that need to be addressed. Below possible avenues for future work are listed.

An interesting research topic would be to expand the comparison research of Chapter 3 onto more varied and complex testbeds. Especially experimenting with real-time algorithms in testbeds that have considerably higher branching factors and different state-space properties.

To complement the comparisons done in Chapter 3 a similar performance comparison is needed for the real-time algorithms that sacrifice the optimality of their final solutions. A detailed evaluation of the performance improvements that these algorithm gain by sacrificing optimality compared to their optimal seeking counterparts. For algorithms such as ϵLRTA* and γ-Trap a study of the most effective parameter values is needed as well as contrasting the performance of the sub-optimal real-time algorithms against the comparisons presented in this thesis. Experiments with ϵLRTA* not included in this thesis, indicated that if any sizable reduction in resource consumption is to be gained the sub-optimality sacrifice had to be considerable. An evaluation is needed of the relative gain of sacrificing optimality opposed to the evaluation of the optimal algorithms presented in Chapter 3. Further research is also needed of other ways to sacrifice optimality in the hopes of an improved real-time performance.

The efficiency of back-propagation in the pathfinding domains contrasted with its relative poor performance in the sliding-tile puzzle raise further questions. Identifying what domain properties are causing back-propagation to excel and backtracking to fail (and *vice versa*) is important to the development of new and more improved algorithm extensions.

The current knowledge of the FBP-LRTA* and the EB-LRTA* algorithms behavior is solely based on empirical testing. Although both algorithms are strongly believed to retain all of LRTA* properties related to convergence and completeness, currently no theoretical proofs can be provided. Formulation of such theoretical work could be advantageous for further understanding the properties of these algorithms.

Bibliography

Björnsson, Y., Enzenberger, M., Holte, R., Schaeffer, J., & Yap, P. (2003). Comparison of different abstractions for pathfinding on maps. *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 03)*, 1511-1512.

Bulitko, V. (2003). Lookahead pathologies and meta-level control in real-time heuristic search. In *Proceedings of the 15th euromicro conference on real-time systems* (pp. 13–16). Porto, Portugal.

Bulitko, V. (2004). *Learning for adaptive real-time search* (Tech. Rep.). Computer Science Research Repository (CoRR).

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

Bulitko, V., Li, L., Greiner, R., & Levner, I. (2003). Lookahead pathologies for single agent search. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), poster section* (pp. 1531–1533). Acapulco, Mexico.

Dechter, R., & Frost, D. (1998). *Backtracking algorithms for constraint satisfaction problems; A Survey* (Tech. Rep.). School of Information and Computer Science, University of California, Irvine, CA 92697-3425.

Dechter, R., & Pearl, J. (1985). Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM*, *32*(3), 505-536.

Dorf, R. C. (1988). *Modern control systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Edelkamp, S., & Eckerle, J. (1997). New strategies in learning real time heuristic search. In *On-line Search: Papers from AAAI Workshop, Providence, RI* (pp. 30–35). AAAI Press.

Furcy, D., & Koenig, S. (2000). Speeding up the convergence of real-time search. In *Proceedings of the national conference on artificial intelligence (AAAI/IAAI)* (pp. 891–897).

Furcy, D., & Koenig, S. (2001). Combining two fast-learning real-time search algorithms yields even faster learning. In *Proceedings of the Sixth European Conference on Planning (ECP-01), Toledo, Spain, 2001.*

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

Hart, P. E., Nilsson, N. J., & Raphael, B. (1972). Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, *37*, 28–29.

Hart, T., & Edwards, D. (1963). *The alpha-beta heuristic* (Tech. Rep.). Cambridge, MA, USA.

Hernández, C., & Meseguer, P. (2005a). Improving convergence of LRTA*(k). In *In Proceedings of Workshop on Planning and Learning in A Priori Unknown or Dynamic Domains IJCAI-05.*

Hernández, C., & Meseguer, P. (2005b). LRTA*(k). In *In Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05.*

Ishida, T. (1997). *Realtime search for learning autonomous agents.* Kluwer Academic Publishers.

Ishida, T., & Shimbo, M. (1996). Improving the learning efficiencies of realtime search. In *National conference on artificial intelligence (AAAI-96) (AAAI/IAAI)* (Vol. 1, pp. 305–310).

Koenig, S. (2001). Agent-centered search. *Artificial Intelligence Magazine*, *22*(4), 109-132.

Koenig, S. (2004). A comparison of fast search methods for real-time situated agents. In *Aamas '04: Proceedings of the third international joint conference on autonomous agents and multiagent systems* (pp. 864–871). Washington, DC, USA: IEEE Computer Society.

Korf, R. E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109.

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

Lin, L.-J. (1992, May). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*(3-4), 293–321.

Luštrek, M., & Bulitko, V. (2006). Lookahead pathology in real-time path-finding. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search.* Boston, Massachusetts.

Pemberton, J., & Korf, R. (1992). *Making locally optimal decisions on graphs with cycles* (Tech. Rep.). Computer Science Department, University of California at Los Angeles.

Rayner, D. C., Davison, K., Bulitko, V., & Lu, J. (2006). Prioritized-LRTA*: Speeding up learning via prioritized updates. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search.* Boston, Massachusetts.

Russell, S., & Wefald, E. (1991). *Do the right thing: studies in limited rationality.* Cambridge, MA, USA: MIT Press.

Shannon, C. E. (1950, March). Programming a computer for playing chess. *Philosophical Magazine*, *41*, 256–275.

Shimbo, M., & Ishida, T. (2003, May). Controlling the learning process of real-time heuristic search. *Artificial Intelligence Archive*, *146*(1), 1–41.

Shue, L.-Y., & Zamani, R. (1993). An admissible heuristic search algorithm. In J. Komorowski & Z. W. Ras (Eds.), *Methodologies for Intelligent Systems: In Proceedings of the 7th International Symposium (ISMIS-93)* (pp. 69–75). Berlin, Heidelberg: Springer.

Shue, L.-Y., & Zamani, R. (1999, Sep). An intelligent search method for project scheduling problems. *Journal of Intelligent Manufacturing*, *10*, 279–288.

Sigmundarson, S., & Björnsson, Y. (2006, July). Value Back-Propagation vs. Backtracking in Real-Time Search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search.* Boston, Massachusetts, USA: AAAI Press.

Sutton, R., & Barto, A. (1998). *Reinforcement Learning: An Introduction.* MIT Press.

Wikipedia. (2006). *N-puzzle — wikipedia, the free encyclopedia.* `http://en.wikipedia.org/w/index.php?title=N-puzzle&oldid=41723281`. ([Online; accessed 8-March-2006])

Appendix A

Test Domains

Researchers in the field of Artificial Intelligence have used a vide variety of problem domains to test and showcase their research. In most cases these domains have been closely related to each researcher's interest or research area, some are even specially designed to highlight certain nice algorithm properties. Few examples include: *the sliding-tile puzzle* (Korf, 1990; Furcy & Koenig, 2000), *Gridworld with random obstacles* (Shimbo & Ishida, 2003), *wireless network routing* (Bulitko & Lee, 2006), *robot navigation simulator* (Ishida, 1997) and *commercial RTS*[1] *game-maps* (Björnsson et al., 2003; Bulitko & Lee, 2006).

A few of these problem domains were selected to use for comparison in this thesis. They are described in more detail below. These domains can be categorized into two main groups:

1. Domains that have become well established in the artificial literature.
2. Domains that emulate a realistic real-world domain.

All domains use the four-way (quartic) heuristic function and action generation. Several popular test domains (including the sliding-tile puzzle and several pathfinding domains) have been shown to exhibit inherent pathological problems (Bulitko, 2003; Bulitko, Li, Greiner, & Levner, 2003; Luštrek & Bulitko, 2006). These problems have been shown to skew results when using certain lookahead depths. In light of this research the lookahead depth of all algorithms experimented with in this thesis have a fixed value of one to reduce the impact of these domain specific problems. This is similar to what has been done by previously referenced researchers.

A.1 Sliding-Tile Puzzle

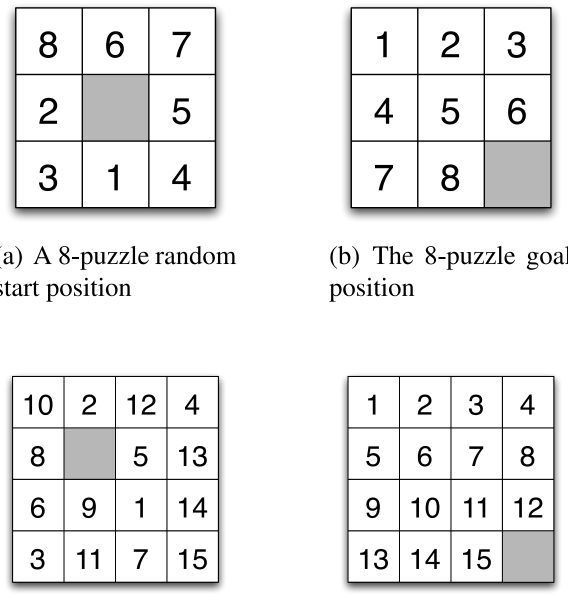
The sliding-tile puzzle, also known as the *n*-puzzle, has been a popular test-bed for Artificial Intelligence research for years. The puzzle and its rules are simple:

[1] Real-Time Strategic

> [The] sliding-tile puzzle consists of a grid of numbered squares with one square missing, and the labels on the squares jumbled up. [...]The goal of the puzzle is to un-jumble the squares by only making moves which slide squares into the empty space, in turn revealing another empty space in the position of the moved piece. (Wikipedia, 2006)

Examples of sliding-tile puzzle configurations are shown in Figure A.1. In real-time search research this testbed was first used by Korf (Korf, 1990) where it was set up with the 15-puzzle configuration for demonstration. Since then researchers have used this testbed (both 8 and 15 piece versions) extensively throughout their experiments. The puzzle provides a good testbed since it has an exponential state space and thus poses a rather hard problem to solve for many common off-line searches. This is especially true for the larger versions of this puzzle, e.g. the 15, 24, 99 piece puzzles. Also, unlike most pathfinding domains, the sliding-tile puzzle has a relatively low frequency of state transpositions (Koenig, 2001).

Korf presented 100 optimally solved instances of the 15-puzzle used for his evaluation (Korf, 1985). In this thesis the same 100 instances are used for all 15-puzzle experiments. Since such a set is not available for the 8-puzzle, a set of 100 random solvable instances of the 8-puzzle were generated. These 100 8-puzzle instances are available for download[2]. The 8-puzzle instances generated for the experiments in this thesis all have solutions of



(a) A 8-puzzle random start position (b) The 8-puzzle goal position

(c) A 15-puzzle random start position (d) The 15-puzzle goal position

Figure A.1: Examples of the 8- and 15 piece Sliding-Tile Puzzle. Each puzzle consists of a set of numbered sliding tiles and a blank location (indicated by the shaded square) into which the horizontally or vertically adjacent tiles may move. The goal location shown is the one used throughout this thesis.

[2] http://nemendur.ru.is/sverrirs01/thesis/

variable length. The shortest 8-puzzle solution is 6 moves and the longest 28. On average the solution length of the 100 instances is 20 moves.

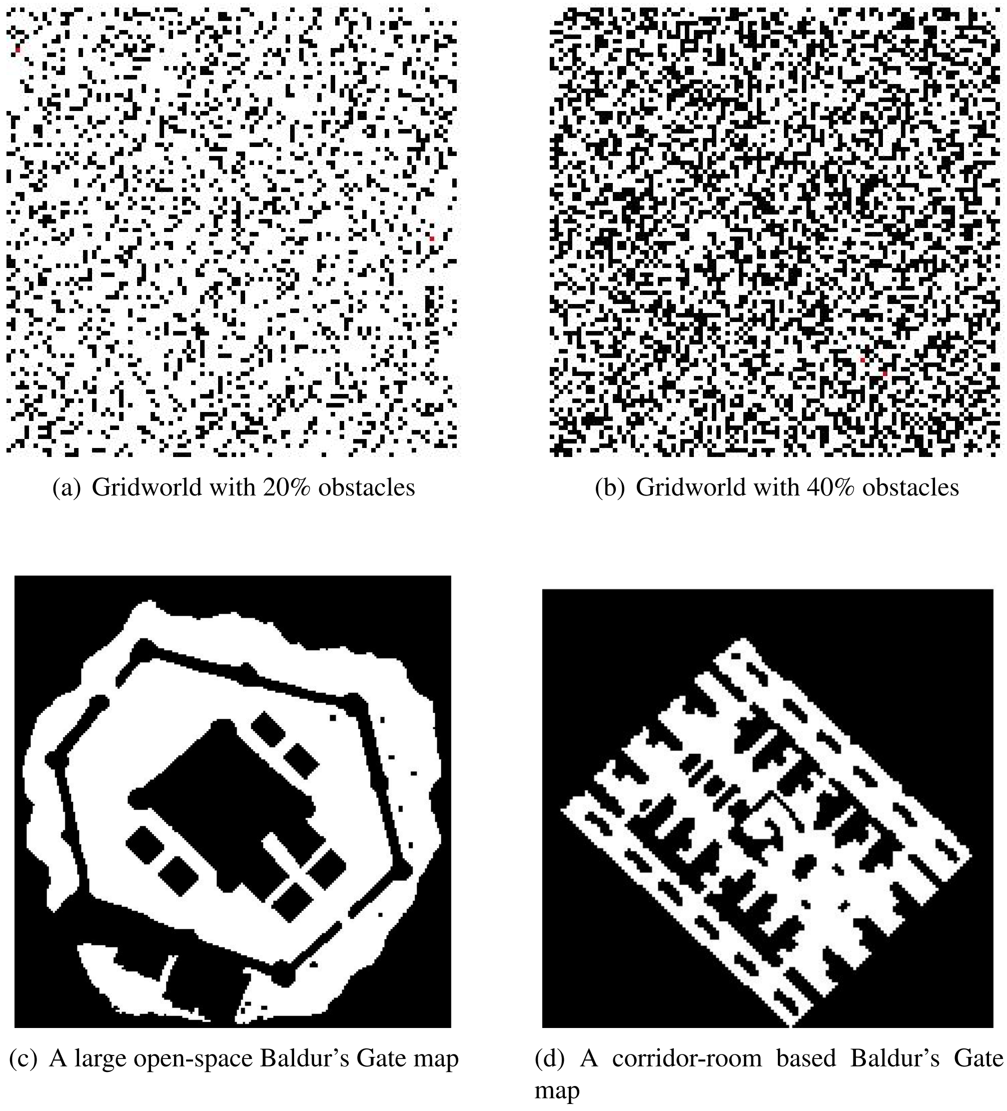
A.2 Random Gridworld

Shimbo and Ishida (Shimbo & Ishida, 2003) used a simple gridworld pathfinding domain in their research. The domain can be configured with a variable amount of obstacles (obstacle ratio) which is user controllable. It was noted by the authors that a certain range of obstacle ratio is preferred to ensure that the problem poses an interesting research problem. For comparisons in this thesis obstacles of three different ratios, 20%, 30% and 40% were randomly generated for each grid. Empirically this has shown to result in neither overly simple paths, i.e. a straight line (ratio lower than 10%) nor in unsolvable problem instances (ratio higher than 50%). The length of the paths also varied between maps (shortest being 5 moves and the longest 327 with a mean of 82 moves).

The Gridworld domain using this random obstacle generation has however an inherent problem relating to the random distribution of the obstacles. A true random function exhibits an equal distribution of its values across the domain. This fact makes clustering of obstacles unlikely, resulting in the forming of relatively straight paths from the start to the goal. Therefore this random obstacle generation does not generate realistic domains, namely systematic room structures, which makes the Gridworld a particularly poor domain to use. The Gridworld is however a part of the experimental section to provide comparisons with previously reported work. Figures A.2(a) and A.2(b) show two examples of the maps used.

A.3 Baldur's Gate Game-Maps

The third domain used is also a pathfinding domain, but now using eight maps taken from the commercial computer game Baldur's Gate I. These maps were deliberately chosen to provide a more realistic evaluation of the search performance in pathfinding domains. Each game map has a logical room structure, hallways, corridors, doorways and large open spaces. For each map 400 random start-goal state pairs were generated. The resulting 3,200 paths had an average length of 71 moves, with the shortest being only 2 moves and the longest 407. This same set of maps was first used by Björnsson in 2003 (Björnsson et al., 2003) and then later as a part of Bulitko's 2005 research (Bulitko & Lee, 2006). Two of the eight Baldur's Gate game maps are shown in Figures A.2(c) and A.2(d).



(a) Gridworld with 20% obstacles

(b) Gridworld with 40% obstacles

(c) A large open-space Baldur's Gate map

(d) A corridor-room based Baldur's Gate map

Figure A.2: A sample of the maps used for the pathfinding domains. Above is a sample of the Gridworld domain, below two of the eight Baldur's Gate maps used. The black areas represent obstacles in both map types.

Appendix B

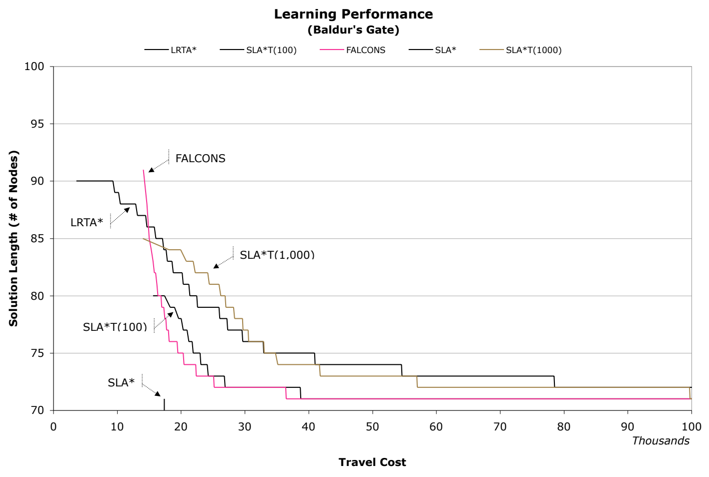
Experimental Data

B.1 Learning Quality

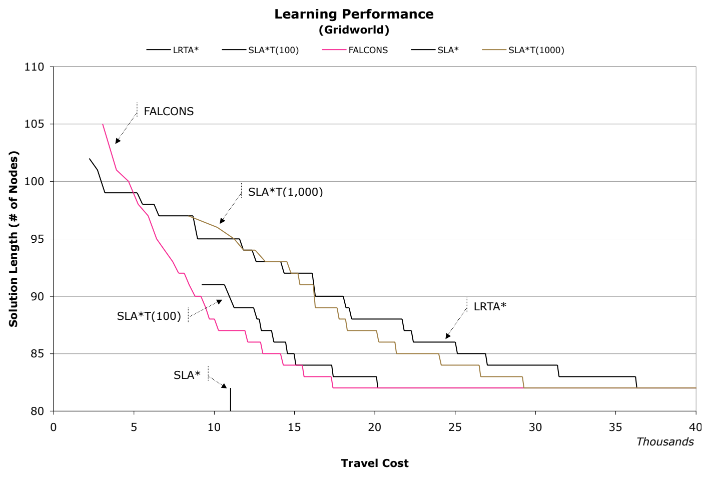
The graphs in Figures B.1 and B.2 are similar to the graph in Figure 3.2. To improve readability the *minimum true solution cost* is plotted on the y-axis instead of the algorithms true solution cost. This has no effect on the accuracy of the graphs.

Figure B.1 shows graphs for the learning performance of LRTA*, FALCONS, SLA* and the SLA*T variants in all domains. Overall FALCONS has the fastest convergence rate of any of the iterative algorithms, however it has a considerably worse first-trial performance than LRTA* in all domains. In both pathfinding domains the SLA*T algorithms perform exceptionally bad compared to their SLA* predecessor reporting first-trial travel costs close to that of SLA* while finding worse solutions.

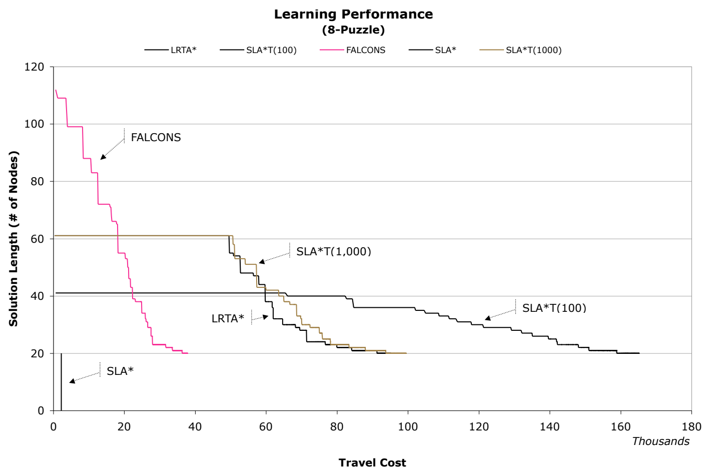
The graphs for δLRTA* are shown in Figure B.2. The performance of the δLRTA* algorithm is only marginally better than LRTA* for the pathfinding domains. However in the 8-puzzle (Figure B.2(c)) δLRTA* performs significantly better than LRTA* and a clear distinction between δ values is apparent (apart from their identical first-trial performance). The performance of δLRTA*(1) in the Baldur's Gate domain (Figure B.2(a)) compared to the 8-puzzle is somewhat interesting. When δLRTA* is run with $\delta = 1$ it converges to significantly inferior solutions in the Baldur's Gate domain, however it behaves quite differently in the 8-puzzle domain. Running δLRTA*(1) in the 8-puzzle returns far better solutions than δLRTA*(2) for more than 3 quarters of their total total travel cost. However as expected, eventually δLRTA*(1) performance worsens as it converges to sub-optimal solutions.



(a) Baldur's Gate Domain

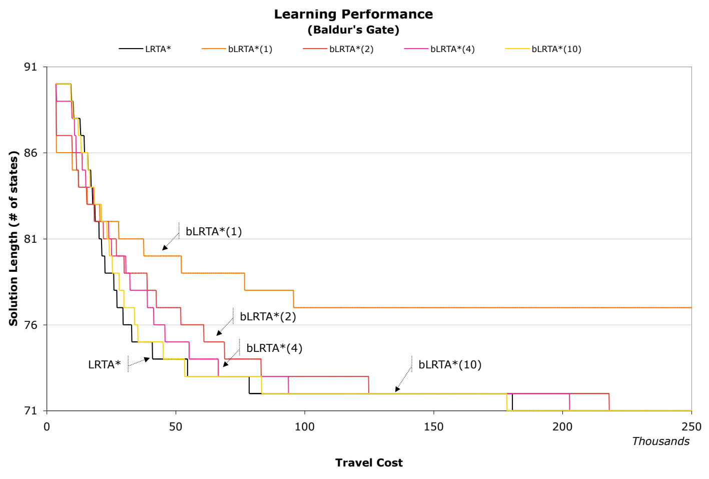


(b) Gridworld Domain

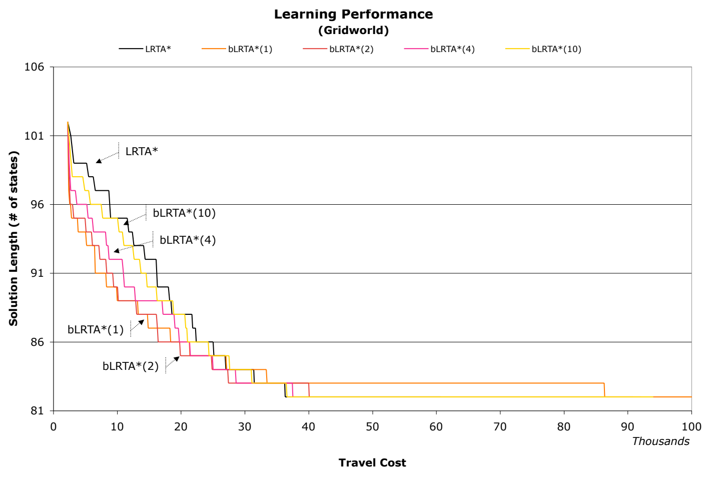


(c) 8-puzzle

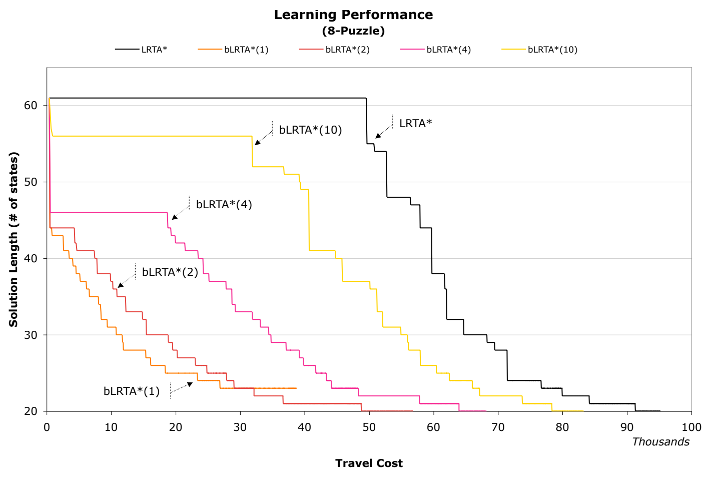
Figure B.1: FALCONS, SLA*, SLA*T and LRTA*'s learning quality in the test domains



(a) Baldur's Gate Domain



(b) Gridworld Domain



(c) 8-puzzle

Figure B.2: Bounded LRTA*'s learning quality in the test domains

Appendix C

A Maze Problem For Your Enjoyment