Journal of Artificial Intelligence Research 30 (2007) 51 - 100 Submitted 03/07; published 09/07

Graph Abstraction in Real-time Heuristic Search

Vadim Bulitko BULITKO@UALBERTA.CA
Nathan Sturtevant NATHANST@CS.UALBERTA.CA
Jieshan Lu JIESHAN@CS.UALBERTA.CA
Timothy Yau THYAU@UALBERTA.CA
Department of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, CANADA

Abstract

Real-time heuristic search methods are used by situated agents in applications that require the amount of planning per move to be independent of the problem size. Such agents plan only a few actions at a time in a local search space and avoid getting trapped in local minima by improving their heuristic function over time. We extend a wide class of real-time search algorithms with automatically-built state abstraction and prove completeness and convergence of the resulting family of algorithms. We then analyze the impact of abstraction in an extensive empirical study in real-time pathfinding. Abstraction is found to improve efficiency by providing better trading offs between planning time, learning speed and other negatively correlated performance measures.

Keywords: learning real-time heuristic search, state abstraction, goal-directed navigation.

1. Introduction and Motivation

In this paper we study the problem of *agent-centered real-time heuristic search* (Koenig, 2001). The distinctive property of such search is that an agent must repeatedly plan and execute actions within a constant time interval that is independent of the size of the problem being solved. This restriction severely limits the range of applicable algorithms. For instance, static search algorithms (e.g., A* of Hart, Nilsson, & Raphael, 1968), re-planning algorithms (e.g., D* of Stenz, 1995), anytime algorithms (e.g., ARA* of Likhachev, Gordon, & Thrun, 2004) and anytime re-planning algorithms (e.g., AD* of Likhachev, Ferguson, Gordon, Stentz, & Thrun, 2005) cannot guarantee a constant bound on planning time per action. LRTA* provides such guarantees by planning only a few actions at a time and updating its heuristic function, but the solution quality can be poor during a lengthy convergence process (Korf, 1990; Ishida, 1992).

As a motivating example, consider navigation in gridworld maps in commercial computer games. In such games, an agent can be tasked to go to any location on the map from its current location. The agent must react quickly to the user's command regardless of the map's size and complexity. Consequently, game companies impose a time-per-action limit on their pathfinding algorithms. As an example, Bioware Corp., a major game company, limits planning time to 1-3 ms for all pathfinding units (and there can be many units planning simultaneously).

An additional challenge comes in the form of limited sensing in virtual reality trainers where Artificial Intelligence controlled characters may not have access to the entire map *a priori*, in order to avoid unrealistic behavior (Dini, van Lent, Carpenter, & Iyer, 2006). Such agents have to build an internal map model based on sensing a limited amount of the map around their position.

An efficient search agent would minimize the delay incurred while planning its actions, explore and learn the environment quickly, and always discover an optimal path to the goal. Unfortunately, these measures are negatively correlated (or *antagonistic*) in that optimizing performance for one measure results in worse performance for one of the others. For instance, reducing the amount of planning done before each action improves the agent's response time, but leads to slower learning due to lower-quality actions taken by the agent.

We propose to use graph abstraction to improve efficiency of search agents and make the following four contributions. First, we introduce a new algorithm, Path Refinement Learning Real-time Search (PR LRTS)[1], which enhances existing real-time heuristic search algorithms with automatically-built graph abstraction. PR LRTS learns its heuristic function in an abstract space thereby substantially accelerating learning. Actions in the abstract space are then refined to actions in the environment by the A* algorithm. This approach allows agents to generate actions in constant time, explore the environment quickly, and converge to near-optimal solutions. In this paper we use the previously published clique abstraction (Sturtevant & Buro, 2005). Our contributions specific to abstraction are three-fold. First, we introduce the initial clique building and the repair procedure in more detail than previously published. Second, we prove a worst-case bound on suboptimality of the path induced by abstraction. Third, we present the first application of state abstraction to real-time heuristic search.

The standard practice in the heuristic search literature is to promote new algorithms as trading a "small" amount of one performance measure for a "large" gain in another performance measure. For instance, state abstraction in non-real time heuristic search has been shown to trade "little" solution quality for a "substantial" reduction in running time (e.g., Holte, Mkadmi, Zimmer, & MacDonald, 1996; Botea, Müller, & Schaeffer, 2004). Unfortunately, it is not always clear whether the trade-offs are made optimally. As the second contribution, we demonstrate that PR LRTS outperforms a number of other algorithms with respect to two antagonistic measures (e.g., learning speed *and* amount of planning per action).

As the third contribution, we analyze effects of abstraction on search with respect to commonly used performance measures: solution suboptimality, amount of planning per action, total travel, total planning time, and memory footprint. Knowing the effects deepens our understanding of real-time heuristic search methods as well as guides a practitioner in selecting the most appropriate search algorithm configuration for her application. Fourth, we show theoretically that PR LRTS unifies and extends several well known existing heuristic search algorithms and satisfies the real-time operation, completeness, and convergence properties. This contribution can be viewed as a follow-up to previous unification and extension efforts (Bulitko & Lee, 2006).

The rest of the paper is organized as follows. We begin by formulating the problem of real-time heuristic search in Section 2. The new algorithm, PR LRTS, is described in Section 4. Empirical results follow in Section 5. Theoretical results are presented in Section 6. We then review existing agent-centered search algorithms as well as work on automatic graph abstraction in Section 7. The paper is concluded by a discussion of current limitations and future research.

2. Real-time Heuristic Search

The defining property of *real-time* heuristic search is that the amount of planning performed by an agent per action has a constant upper-bound that does not depend on the problem size. Low bounds are preferred in applications, as they guarantee the agent's fast response when presented with a new goal. A real-time search agent plans its next action by considering states in a local search space

1. An early version of this algorithm was published as a conference paper (Bulitko, Sturtevant, & Kazakevich, 2005).

```
1 $s_c \leftarrow s_0$
2 while $s_c \neq s_g$ do
3   sense the environment, update agent's model
4   compute partial path $p$ that originates in the current state $s_c$
5   execute $p$
6   update current state $s_c$
7 end while
```

Figure 1: Real-time heuristic search (a single trial).

surrounding its current position. A *heuristic function* (or simply *heuristic*) estimates the cumulative cost between a state and the goal, and is used by the agent to rank available actions and select the most promising one. This process is shown schematically in Figure 1. Agent's current state s_c is set to the initial state s_0 in line 1. As long as the goal s_g is not reached (line 2), the agent senses the environment around it (see Section 3 for details) and updates its model of the search graph it is operating on in line 3. Then it computes a (partial) path from its current state toward the goal state in line 4. The real-time property requires that lines 3 and 4 execute in a constant-bounded time regardless of problem size. This is accomplished by calling a real-time heuristic search algorithm in line 4. In this paper, we discuss three candidate algorithms: LRTA* in Section 2.1, LRTS in Section 2.2, and PR LRTS in Section 4. Each of them would be called from line 4. The agent then executes the path in line 5 and updates its current state in line 6.

A *trial* is defined as the agent's problem-solving experience while traveling from its start state to the goal state. Once the goal state is reached, the agent is teleported to the start state and the next trial begins. A *convergence process* is defined as the first sequence of trials until the agent no longer updates its heuristic function or its model of the search problem. The first trial without such updates is the *final trial* and the learning process is said to have converged.

2.1 Learning Real-time A* (LRTA*)

We first review the best known real-time heuristic search algorithm, Learning Real-Time A* (LRTA*) (Korf, 1990). The algorithm is shown in Figure 2. In line 1, a d-ply breadth-first search with duplicate detection is used to find frontier states precisely d actions away from the current state s. The standard path-max (Mero, 1984) technique is used to deal with possible inconsistencies in the heuristic function when computing $g + h$-values. The value of each state, $\hat{s}$, is the sum of the cost of a shortest path from s_c to $\hat{s}$, denoted by $g(s, \hat{s})$, and the estimated cost of a shortest path from $\hat{s}$ to s_g (i.e., the heuristic value $h(\hat{s}, s_g)$). The state that minimizes the sum is identified as s' in line 2. The heuristic value of the current state s is updated in line 3. Finally, a path of one action toward the most promising frontier state s' is returned in line 4.

```
path LRTA*($s_c, s_g, d$)
1 generate successor states of $s_c$ up to $d$ actions away
2 find state $s'$ with the lowest $g(s_c, s') + h(s', s_g)$
3 update $h(s_c, s_g)$ to $g(s_c, s') + h(s', s_g)$ if it is greater than the current $h$
4 return the first action along an optimal path from $s_c$ to $s'$
```

Figure 2: The LRTA* algorithm.

2.2 Learning Real-time Search (LRTS)

LRTS extends LRTA* in three ways: it puts a weight on the heuristic function, it uses the max-of-min learning rule, and it utilizes backtracking. We review these extensions in more detail in Section 7.2 and walk through LRTS operation below. LRTS has three control parameters: lookahead $d \in \mathbb{N}$, optimality weight $\gamma \in (0, 1]$, and learning quota $T \in [0, \infty]$. It operates as follows. In the current state s_c, the agent running LRTS conducts a full-width d-ply lookahead search (line 1 in Figure 3). At each ply, it finds the most promising state (line 2). Assuming that the initial heuristic h is admissible, it can safely increase $h(s_c)$ to the maximum among the f-values of promising states for all levels (line 3). If the total learning amount u (updated in line 4) exceeds the learning quota T, the agent backtracks to the previous state from which it planned (lines 5, 8). Otherwise, it returns a path of d moves between the current state s_c and the most promising state at level d (line 6). The learning amount u is reset to 0 when the agent is in the start state (i.e., at the beginning of each trial).

```
path LRTS($s_c, s_g, d, \gamma, T$)
1  generate successor states of $s_c$, $i$ actions away, $i = 1 \dots d$
2      on level $i$, find the state $s_i$ with the lowest $f(s_i) = \gamma \cdot g(s_c, s_i) + h(s_i, s_g)$
3  update $h(s_c, s_g)$ to $\max_{1 \le i \le d} f(s_i)$ if it is greater than the current $h$
4  increase the amount of learning $u$ by $\Delta h$
5  if $u \le T$ then
6      return a path of $d$ actions from $s_c$ to $s_d$
7  else
8      return a path of $d$ actions to backtrack to the previous state, set $u = T$
9  end if
```

Figure 3: The LRTS algorithm.

LRTS parameters have been previously studied at length (Bulitko & Lee, 2006). Here we summarize the trends. Higher lookahead d reduces convergence travel, convergence memory, and suboptimality. However, it increases the first-move lag. A lower heuristic weight γ leads to less optimal solutions and, generally speaking, reduces convergence travel and convergence memory. First-move lag is not influenced by γ. A lower learning quota T causes more backtracking and tends to reduce convergence travel and convergence memory; T does not affect the first-move lag.

2.3 Notation

Definition 2.1 A *search problem* is defined as a tuple (G, c, s_0, s_g, h_0) where $G = (S, E)$ is a directed weighted graph (henceforth *search graph*). S is a finite set of states (or vertices) and $E \subset S \times S$ is a finite set of edges between them. The edge weights are defined by the cost function $c : E \to (0, \infty)$ with $c(s_1, s_2)$ being the *travel cost* for the edge $e = (s_1, s_2)$. $s_0 \in S$ is the start state, $s_g \in S$ is the goal state, and $h_0 : S \to [0, \infty)$ is the initial heuristic function. We assume that $h_0(s_g) = 0$. Out-edges of a state are called *moves* or *actions*. The number of out-edges (i.e., out-degree of a state) is called the *branching factor* of a state.

Definition 2.2 A solution to a search problem is a *path* from the start state s_0 to the goal state s_g. The path is denoted by $(s_0, s_1, \dots, s_g)$ where each s_i is a valid state and there is a valid edge for each pair of states (s_i, s_{i+1}). The *travel cost* of a path is the sum of travel costs of its edges.

Definition 2.3 At all times, a search agent resides in a single search state $s_c \in S$ called the *current state*. The agent can change its current state only by executing actions and thus incurring travel cost. Initially, the current state coincides with the start state s_0. An agent is said to succeed when it makes its current state coincide with the goal state s_g.

We assume that the goal state is reachable from any state the agent can get to from its start state. This is needed for completeness in all real-time heuristic search algorithms. We also follow the standard practice in real-time heuristic search literature and assume that the environment is stationary and deterministic. Additionally, to support backtracking (Shue & Zamani, 1993; Shue, Li, & Zamani, 2001) (i.e., reversing agent's actions), we require that every action has a reverse action. This is needed only when backtracking is enabled in our algorithm.

Definition 2.4 The *travel cost* from state s_1 to state s_2 denoted by $\mathrm{dist}(s_1, s_2)$ is defined as the cost of a shortest path from s_1 to s_2. Throughout the paper, we will assume that dist satisfies the triangle inequality: $\forall s_1, s_2, s_3 \in S\,[\mathrm{dist}(s_1, s_3) \leq \mathrm{dist}(s_1, s_2) + \mathrm{dist}(s_2, s_3)]$. Then, for any state, s, $h^*(s)$ is defined as the minimal travel cost to the goal: $h^*(s) = \mathrm{dist}(s, s_g)$. A *heuristic function*, h, is an approximation of h^*. It *admissible* if it does not overestimate h^*: $\forall s \in S\,[h(s) \leq h^*(s)]$. The value of h in state s will be referred to as the heuristic value of state s. We assume that for any heuristic function $h(s_g) = 0$ which trivially holds for an admissible h.

In our experiments we break all ties between moves in a fixed fashion (e.g., always prefer the action "north", then "north east", then "east", etc.) which entails that the agent's behavior will be identical on all trials after the final trial. It does not necessarily mean that the entire search graph is explored or the learned heuristic is accurate for all states.

Definition 2.5 *Convergence travel* is the cumulative cost of all edges traversed by the agent during the convergence process. *Convergence planning* is the amount of all planning effort expended by the agent during the convergence process. The *first-move lag* is the amount of planning effort expended by the agent on the first move of its final trial. *Convergence memory* is measured as the total number of heuristic values stored during the convergence process. The standard practice in the real-time heuristic search literature (e.g., Korf, 1990; Shimbo & Ishida, 2003) is to store the heuristic values in a hash table. Hash table misses are handled by a procedurally specified initial heuristic h_0 (e.g., the Manhattan distance in grid-based pathfinding). Then convergence memory is the number of entries in the hash table after convergence. Finally, *suboptimality* is defined in percentage points as the final-trial solution cost excess relative to the shortest-path cost. For instance, if the agent incurred the travel cost of 120 and the shortest-path cost is 100, the suboptimality is 20%.

We measure planning effort in two ways. First, we report the number of states the algorithm "touched" (i.e., considered) during planning. This measure is called *edges traversed* (e.g., Holte et al., 1996, p. 325). Second, we report physical CPU time, measured on a 2.0GHz PowerPC G5 computer with gcc 4.0 under Mac OS 10.4.8. We measure convergence memory in terms of the number of heuristic values stored. This is meaningful because each heuristic value stored takes the same fixed amount of memory (i.e., `double` type of C++) in our implementation of each algorithm.

Definition 2.6 A search algorithm exhibits *real-time performance* on a heuristic search problem if its planning effort per move is constant-bounded and the constant is independent of the problem size (assuming a fixed maximum branching factor).

The objectives of a real-time search agent are to be complete (i.e., to arrive at a goal state on every trial), to converge (i.e., finish the learning process after a finite number of trials), and to minimize the five performance measures described above. In the rest of the paper we will discuss how existing and the new algorithms compare in terms of these objectives.

2.4 Application: Goal-directed Navigation

One of the motivating applications of heuristic search is *goal-directed navigation*, also known as *pathfinding*. It is a special case of the heuristic search problem formalized in the previous section where the search graph (S, E) is defined by a terrain map. Thus, the states/vertices correspond to geographical positions on a map, the edges describe passability or blocking, and the cost function represents the difficulty/time of traversing the terrain.

Real-time pathfinding is motivated primarily by time-sensitive robotics (e.g., Koenig & Simmons, 1998; Koenig, 1999; Kitano, Tadokoro, Noda, Matsubara, Takahashi, Shinjou, & Shimada, 1999; Koenig, Tovey, & Smirnov, 2003) and computer games. The latter include real-time strategy games (e.g., Blizzard Entertainment, 2002), first-person shooters (e.g., id Software, 1993), and role-playing games (e.g., BioWare Corp., 1998). In all of these, time plays a critical role since a number of agents can perform pathfinding simultaneously and gamers would like rapid response and fluid gameplay. As a result, pathfinding has become a major computational expense: in "Age of Empires II" (Ensemble Studios, 1999) it takes up to 60-70% of the simulation time (Pottinger, 2000).

In this paper, we follow the footsteps of Furcy and Koenig (2000), Shimbo and Ishida (2003), Koenig (2004), Botea et al. (2004), Hernández and Meseguer (2005a, 2005b), Sigmundarson and Björnsson (2006), Koenig and Likhachev (2006) and situate our empirical study in navigation on two-dimensional grid-based maps. The cells are square and each cell is connected to four cardinally (i.e., west, north, east, south) and four diagonally neighboring cells. Each cell can be occupied by an agent (i.e., free) or by a wall (i.e., blocked).

Each free grid cell constitutes a vertex/state in the search space S. If the agent can travel between any two free neighboring cells, s_1 and s_2, an edge (s_1, s_2) is added to the set of edges E. In this paper, we set the edge costs to 1 for cardinal moves and to $\sqrt{2}$ for diagonal moves. The cell initially occupied by the agent is s_0; the target cell is s_g. An example of converting a grid-based map to a search problem defined by (G, c, s_0, s_g, h_0) is shown in Figure 4. Note that we do not allow diagonal moves that "cut corners" and, thus, the state s_6 is not connected to states s_1, s_5, s_7, s_g. This is done because a non-zero size agent will not be able to pass through a zero-width bottleneck formed by two diagonally adjacent blocked cells. In the case when there is only one corner (e.g., between states s_5 and s_6 in Figure 4), allowing to cut it would lead to the actual travel distance exceeding $\sqrt{2}$ since a non-zero-width agent will have to walk around the corner.

Video games often feature *repeated* pathfinding experiences on the same map for two reasons: (i) there are units that commute between the same source and destination (e.g., resource collectors in real-time strategy games) and (ii) all ally units can share results of their learning (i.e., the heuristic function). Since a trial typically improves heuristic values of many states, even a single trial of a single unit can be of use to other units with different start states as long as they all share a goal state. This is often the case with state abstraction as an entire region of a map (e.g., a room in a role-playing game or the player's home base in a real-time strategy game) can be mapped into a single abstract state. Thus, single-trial learning experiences of multiple units can be approximated by multi-trial learning experience of a single unit. The latter is the scenario we study in this paper, in

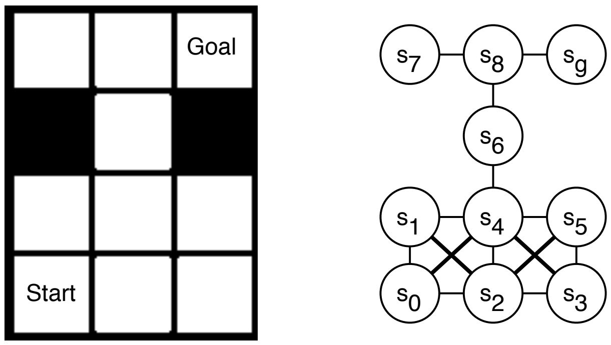

Figure 4: A 3×4 grid-based map (left) converted to a 9-state search graph (right). Thinner cardinal-direction edges have the cost of 1, thicker diagonal edges have the cost of $\sqrt{2}$.

line with Furcy and Koenig (2000), Shimbo and Ishida (2003), Sigmundarson and Björnsson (2006) and others.

3. Search Graph Discovery

In this paper, we do not require a search agent to know the problem in its entirety. Instead, a portion of the search problem in the neighborhood of the current state s_c is sensed by the agent at each time step. We assume that an agent can remember parts of the problem it has sensed so far. In other words, at all times an agent has an internal representation (model) of what the search space is like. The model is updated as the agent discovers the search graph (line 3 of Figure 1).

Let us illustrate the exploration process in goal-directed navigation. The terrain map is initially unknown to the agent. As it moves around the environment, grid cells whose coordinates are within a fixed visibility radius of the agent's current position are sensed. Formally, the agent situated in cell (x, y) can check the status (free/blocked) of any cell (x', y') if $|x - x'| \leq r$ and $|y - y'| \leq r$, where $r \in \mathbb{N}$ is a visibility radius. Thus, for any two visible cells (x', y') and (x'', y'') the agent can tell if there is an edge between them and its cost. This is similar to virtual sensors used by Thalmann, Noser, and Huang (1997).

One common approach is to assume a regular structure of the unknown part of the search space (Koenig et al., 2003; Koenig, 2004; Bulitko & Lee, 2006; Koenig & Likhachev, 2006). For instance, in grid-based pathfinding, the agent can assume that there are no obstacles in the gridworld until it senses otherwise (this is sometimes called the "free space assumption"). We demonstrate this in Figure 5, where the agent assumes that the space is obstacle-free (a) and builds its internal model accordingly (b). Exploration reveals obstacles in the environment (c) which cause the agent to update its model (d). We impose the restriction that a search agent never needs to add edges to its model during exploration and the weights of discovered edges never change. In other words, agent's initial model is optimistic and contains a superset of edges of the actual search graph. Adding edges or allowing arbitrary edge weight changes may require the agent to explore the environment explicitly. Combining exploration and exploitation effectively is an active research area (for early work, refer to Sutton, 1990) and is not addressed in this paper.

Map discovery is natural in robotics where sensors have limited ranges. In software domains, the agent can theoretically have access to an entire environment. Several types of arguments have been made to justify restricting an agent's senses in software domains. First, omniscient virtual

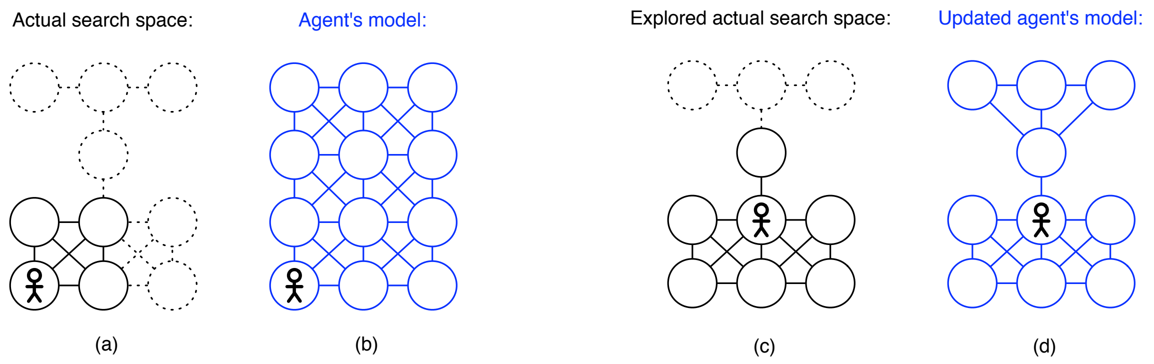

Figure 5: **(a):** initially only the part of the search space shown with solid lines is sensed by an agent (shown as a stick figure). The agent's model assumes a regular structure for the unknown part **(b)**. As the agent moves north-east, it senses an additional part of the search space **(c)** and updates its model correspondingly **(d)**.

humans tend to behave unrealistically and, thus, are less suitable for virtual reality trainers (Dini et al., 2006). Likewise, in commercial games, revealing the entire map to an AI player is viewed negatively as cheating. Second, it can be computationally expensive to sense (Orkin, 2006) and reason about an entire environment (Thalmann et al., 1997; Aylett & Luck, 2000). Consequently, localized sensing is used in large-scale multi-unit systems (Reynolds, 1987).

4. Path Refinement Learning Real-time Search (PR LRTS)

Real-time heuristic search algorithms plan using a small part of the search graph that surrounds an agent's current state. In order to avoid getting stuck in infinite loops, they update the heuristic function over time. This approach guarantees that each action is planned in a constant-bounded amount of time. The downside is slow convergence.

The central idea of PR LRTS is to address this downside by running real-time search on a smaller abstract search graph and then refining the produced abstract path into a ground-level path. The abstract graph is an image of the original graph under an abstraction operator. The operator maps a region of states in the original graph to a single abstract state in the abstract graph. When applied multiple times, a hierarchy of abstractions are formed. The hierarchy is a forest (a tree for each connected component of the search graph) and will be formalized in Section 4.2.

A variety of terminologies have been used in the literature for discussing the relationship between states at different levels of abstraction. In different contexts the abstract states have been referred to as clusters (Botea et al., 2004), sectors/regions (Sturtevant, 2007), and images (Holte et al., 1996). Because the abstraction is a forest, and in line with (Bacchus & Yang, 1994; Bulitko et al., 2005; Sturtevant & Buro, 2005), we sometimes call an image of an abstraction operator parent and its pre-image children. These terms are not to be confused with successor states in lookahead search. We first describe PR LRTS at an intuitive level and illustrate it with an example in Section 4.1. Then we give formal details in Section 4.2 and describe the abstraction operator in detail.

4.1 Path Refinement

PR LRTS computes paths at several levels of abstraction. First, a path is found through the most abstract search space (at level ℓ). Such an abstract path defines a region of the lower-level abstract

```
path **PR LRTS**($s_c, s_g$)
1  **if** level($s_c$) $> \ell$ **then**
2    **return** $\emptyset$
3  **end if**
4  $p$ =PR LRTS(parent($s_c$), parent($s_g$))
5  **if** $p \neq \emptyset$ **then**
6    $s_g$ = child(end($p$))
7  **end if**
8  $C = \{s \mid \text{parent}(s) \in p\}$
9  **switch**(algorithm[level($s_c$)])
10   A* : **return** A*($s_c, s_g, C$)
11   LRTS : **return** LRTS($s_c, s_g, C$)
12   pass-through : **return** $p$
13 **end switch**
```

Figure 6: The PR LRTS algorithm.

space that will be searched when refining the abstract path. This refinement proceeds incrementally until the level-0 (i.e., ground) search space is reached and a ground path is produced. In order to keep the amount of planning per move constant-bounded regardless of the ground space size, we need to have a real-time algorithm on the most abstract search graph. In this paper, we use LRTS at a *fixed* top level of abstraction (ℓ), and A* for refinement at the lower levels.[2] Some abstract levels can be left as "pass-through" to merely increase the amount of state aggregation; no processing is carried out at them. This design choice was motivated by experimentation (Section 5).

PR LRTS operates recursively as presented in Figure 6. In line 1 it checks if the states passed to it are above the top level of abstraction on which pathfinding is to occur. If so, an empty path is returned (line 2). Otherwise, the function calls itself recursively to compute a path between the abstract image of s_c (denoted by $\text{parent}(s_c)$ in line 4) and the abstract image of s_g. The returned path (if non-empty as checked in line 5) is used to derive the new destination in line 6. Specifically, the new destination s_g is a child of the end of the abstract path p.[3] In line 8, we compute the corridor C comprised of pre-images of the states on path p. The corridor C will be empty if the path p computed in line 4 was empty. Finally, we run the algorithm assigned to our current level of abstraction (i.e., the level of s_c and s_g) in lines 10 and 11. It will be the A* or the LRTS tasked to find either a full (in the case of A*) or a partial path (in the case of LRTS) from s_c to s_g limited to the set of states C. By convention, an empty corridor ($C = \emptyset$) allows A*/LRTS to search its entire graph. Note that no processing happens at a pass-through level (line 12).[4]

2. Because an agent explores its environment while moving about, we actually use a Local Repair A* instead of A*. It is described in Section 7.
3. While any child can be used, some choices may lead to better performance. Intuitively, the child chosen by $\text{child}(\text{end}(p))$ should be the "closest representative" of the abstract state $\text{end}(p)$ among $\text{children}(\text{end}(p))$. In pathfinding, we implement $\text{child}(s)$ to return the element of $\text{children}(s)$ that is geographically closest to the average coordinates of states in $\text{children}(s)$. Also, if the goal state s_g happens to be in the pre-image of $\text{end}(p)$ then we pick it as $\text{child}(\text{end}(p))$.
4. Also note that the functions child and parent handle pass-through levels. Specifically, in line 6, the state s_g will be computed by child at the first non-pass-through level below the level at which path p is computed. Likewise, in line 8, the states s forming the corridor C are at the first non-pass-through level (level i) below the level of path p (level j). Thus, $\text{parent}(s)$ will apply the abstraction mapping $j - i$ times so that $\text{parent}(s)$ and p are both at level j.

Our implementation of A* is standard (Hart et al., 1968) except it is run (line 10) on a subgraph defined by the corridor C (line 8). Our implementation of LRTS is taken from the literature (Bulitko & Lee, 2006) and is described in Section 2.2. Like A*, we run LRTS in the corridor C.

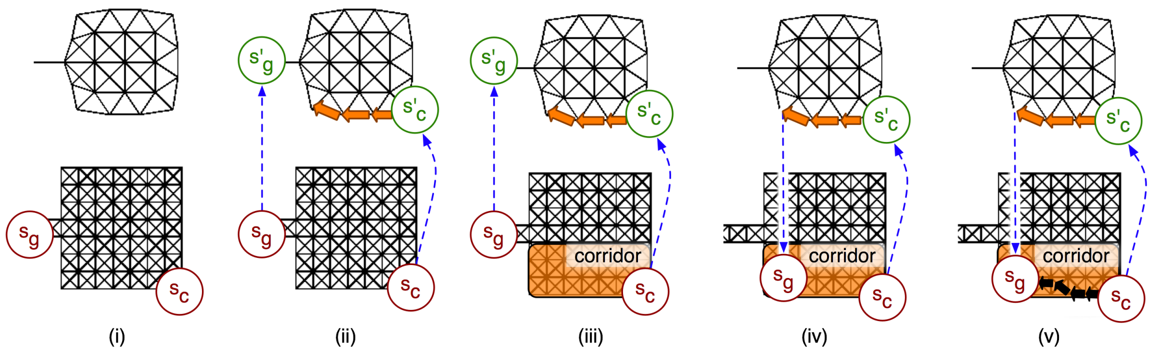

Figure 7: The path refinement process. The original graph (level 0) is shown at the bottom. The abstract graph (level 1) is shown at the top.

For illustration purposes, consider an example in Figure 7. In the example $\ell = 1$, so only one level of abstraction (shown at the top) is used in addition to the ground level (shown at the bottom). s_c is the current state while s_g is the destination state. LRTS is assigned to level 1 while A* is assigned to level 0. Subfigure (i) shows the ground state space below and one level of abstraction above. The agent must plan a path from s_c to s_g located at the ground level. First, the abstract parents of s_c and s_g, $\text{parent}(s_c) = s'_c$ and $\text{parent}(s_g) = s'_g$, are located. Then LRTS with $d = 3$ plans three steps in the abstract space (ii). A corridor C at the ground level comprised of children of the abstract path is then built (iii). A child representing the end of the abstract path is set as the new destination s_g (iv). Finally, A* is run within the corridor to find a path from s_c to the new destination s_g (v).

While an agent is executing a path computed by PR LRTS, new areas of the search graph may be seen. This causes updates to the abstraction hierarchy that the agent maintains. PR LRTS clears and recomputes its abstract paths upon discovering new areas of the search graph. Also, if a ground path proves invalid (e.g., runs into a newly discovered obstacle), the execution stops and PR LRTS replans from the current state using the updated abstraction hierarchy.

Graph discovery can lead to arbitrary updates to the abstract search graphs an agent maintains. In our implementation, LRTS operating on an abstract graph resets its heuristic function if its abstract search graph is updated in any way. On the other hand, updates to the ground-level graph are limited to state and edge removals (Section 3). Consequently, the heuristic learned at the ground level remains admissible and there is no need to reset it upon updates.

4.2 Automatic Graph Abstraction

We use the term *abstraction operator* (or *abstraction*, for short) to mean a graph homomorphism in line with Holte et al. (1996). Namely, abstraction is a many-to-one function that maps (*abstracts*) one or more states to a single abstract state. Adjacent vertices are mapped to adjacent or identical vertices (Property 5 below). Given such a graph homomorphism function and a model of a search problem, a PR LRTS agent builds ℓ additional abstract search graphs, collectively called an *abstrac-*

tion hierarchy, as follows. It first applies the graph homomorphism to the search graph of the model (called *ground-level* graph). The result is an abstract search graph at level 1. The process is then repeated until an abstract search graph at level ℓ is computed. Any homomorphic abstraction can be used as long as the resulting hierarchy of abstract graphs satisfies several key properties. In the following we introduce the properties informally and illustrate them with an example. In Appendix B we formalize them.

Property 1 Every abstract graph is a search graph in the sense of Definition 2.1 in Section 2.

Property 2 Every state has a unique abstract parent (except states at the top level of abstraction).

Property 3 Every state at any abstract level, has at least one child state below.

Property 4 Given a heuristic search problem, the number of children of any abstract state is upper-bounded by a constant independent of the number of states in the ground-level graph.

A corollary of this property is that the number of ground-level states that abstract into a single state at any *fixed* level of abstraction is also constant-bounded with a constant independent of the ground-level graph size.

Property 5 (Graph homomorphism) Every edge in the search graph at a level of abstraction has either a corresponding edge at the level above or the states connected by the edge abstract into a single abstract state.

Property 6 If an abstract edge exists between two states then there is an edge between at least some child of one state and some child of the other.

Property 7 Any two children of an abstract state are connected through a path whose states are all children of the abstract state.

Property 8 The abstraction hierarchy is consistent with agent's model of its search problem at all times. That is, properties 1 through 7 are satisfied with respect to the agent's model.

In this paper, we use a clique-based abstraction mechanism (Sturtevant & Buro, 2005). It operates by finding fully connected components (cliques) in the search graph and mapping each to a single abstract state. This method of building abstractions is favored in recent analysis by Sturtevant and Jansen (2007) and earlier analysis by Holte et al. (1996, Section 5.2) who showed that reduction of search effort due to abstraction is maximized by minimizing edge diameter of the set of children and maximizing its size. For any clique, its edge diameter (i.e., the maximum number of edges between any two elements) is one while the number of states in a clique is maximized.

We present the clique-based abstraction mechanism by developing several stages of a hand-traceable example. We then illustrate how each of the properties introduced above is satisfied in the example. A formal introduction of the clique abstraction technique complete with pseudocode is found in Appendix A. We review other ways of building abstraction in Section 7.3. Note that while general clique computation is NP-complete, finding cliques in two-dimensional grid-based search graphs can be done efficiently (Appendix A).

A single application of the abstraction procedure is illustrated in Figure 8. Cliques of size four are first located in the graph, meaning that states s_0, s_1, s_2 and s_4 are abstracted into s'_1. There are no cliques of size three which are not already abstracted in the first step, so cliques of size two will be abstracted next. This includes s_5 and s_3 which are abstracted into s'_2, and s_7 and s_8 which are abstracted into s'_3. Because s_g has degree 1, we add it to s'_3; however s_6 has degree two, so it is abstracted into its own parent, s'_4. Adding degree 1 states to their neighbors reduces the number of resulting abstract states but increases the edge diameter of the set of children (it becomes 2 for the set $\{s_7, s_8, s_g\}$). This is a minor detail of our abstraction that happens to be effective in grid-based pathfinding. One can use "pure" clique abstraction as well.

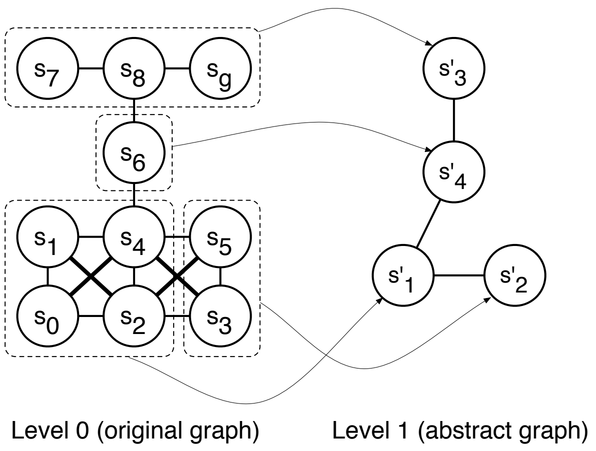

Figure 8: Clique abstraction: the original search graph from Figure 4 shown on the left is abstracted into the search graph on the right.

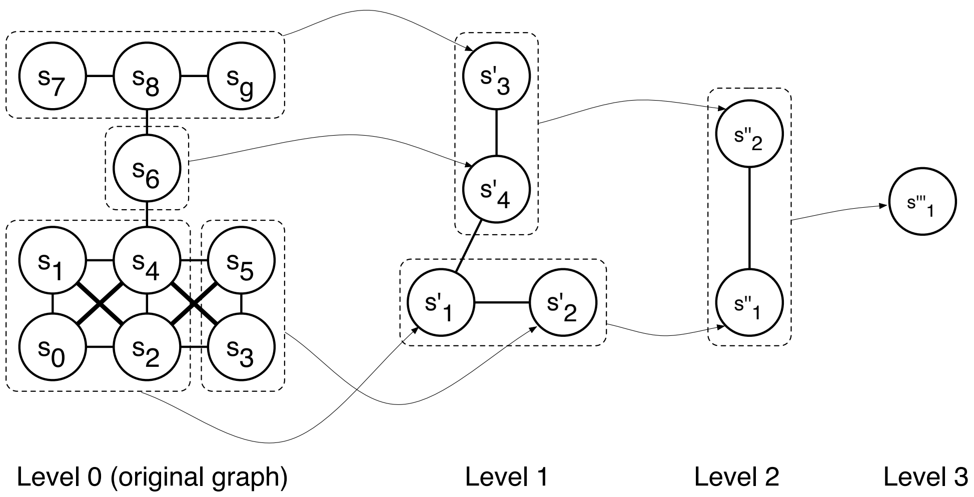

Figure 9: Three iterations of the clique abstraction procedure.

The abstraction process can be successively applied until a single abstract state for each connected component of the original graph remains (Figure 9, Level 3). We can now illustrate the abstraction properties with Figure 9. Property 1 requires that each of the four abstraction levels in Figure 9 is a search graph in the sense of Definition 2.1 in Section 2. Property 2 requires each state at levels 0, 1, and 2 to have a unique parent at the level above. Property 3 requires that each state at levels 1, 2, and 3 has a non-empty set of children at the level below. Property 4 places an upper bound on the number of children each abstract state can have. In this example the bound is 4. Properties 5 and 6 require that, for any two abstract states connected by a path, their parents and any of their children are also connected. Consider, for instance, abstract states s'_2 and s'_3. They are connected at level 1 as there is an abstract path $p = (s'_2, s'_1, s'_4, s'_3)$ between them. Thus, any child of s'_2 is also connected to any child of s'_3 at level 0. For instance, s_3 is connected to s_7. Property 7 requires that all children of node s'_1 are connected by an internal path within s'_1. (s'_0, s'_1, s'_2, s'_4) form a clique, so this property is satisfied.

The costs of abstract edges (e.g., edge (s'_1, s'_2) in Figure 8) can be defined in an arbitrary way as long as the resulting search graph satisfies properties in Section 2. However, for better performance, a low-cost abstract path should be an abstraction of a low-cost ground path. In this paper we experiment with grid-based navigation and, correspondingly, define the cost of edge (s'_1, s'_2) as the Euclidean distance between the average coordinates of $\text{children}(s'_1)$ and $\text{children}(s'_2)$ (Figure 10).

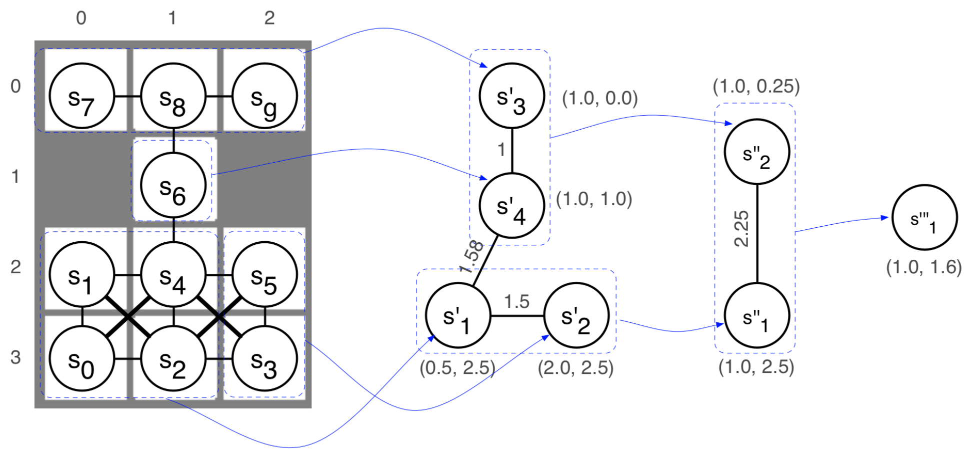

Figure 10: Coordinates and edge costs for all levels of the abstraction hierarchy. At the grid level (leftmost illustration) vertex coordinates are given through column/row labels. Ground edges cost 1 (cardinal) $\sqrt{2}$ (diagonal). Abstract states are labeled with (x, y). Abstract edges are labeled with their approximate cost.

In practice, Property 8 is satisfied by repairing an agent's abstraction hierarchy upon updates to the agent's model. To illustrate, imagine an agent just discovered that discrepancies between the terrain elevation in state s_4 and s_6 (Figure 8) prevent it from being able to traverse the edge (s_4, s_6). It will then update its model by removing the edge. Additionally, degree-one state s_6 will join the clique $\{s_7, s_8\}$. At this point, an agent's abstraction hierarchy needs to be repaired. This is accomplished by replacing abstract states s'_4 and s'_3 with a single abstract state s'_5. The edges (s'_1, s'_4) and (s'_4, s'_3) will be removed. If more than one level of abstraction is used then the repair has to be propagated to higher levels as well. The repair mechanism is presented in detail in Appendix A.2. We will prove in Section 6 that the PR LRTS algorithm can operate with any abstraction mechanism that satisfies the properties listed above.

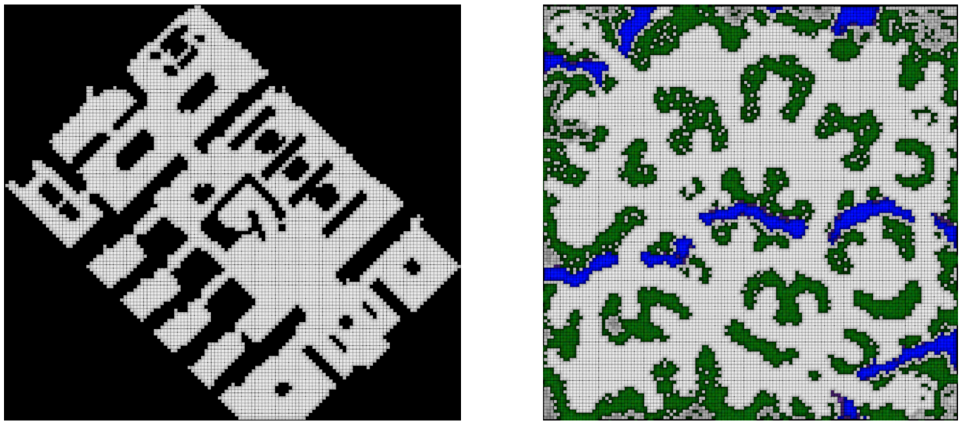

Figure 11: Two of the the six maps used in the experiments.

5. Empirical Study

Empirical evaluation of the effects that state abstraction has on learning real-time heuristic search is presented in four parts. In Section 5.1, we introduce the concept of trading off antagonistic measures and demonstrate that PR LRTS makes such trade-offs efficiently. This is due to the use of abstraction and, consequently, we investigate the effects of abstraction independently of LRTS control parameters in Section 5.2. We then study how PR LRTS performance scales with problem size (Section 5.3). Finally, we examine the interplay between the effects of abstraction and the LRTS control parameters. As it is the most domain-specific study we present these details in Appendix F.

The experimental setup is as follows. We used 3000 problems randomly generated over three maps modeled after environments from a role-playing game (BioWare Corp., 1998) and three maps modeled after battlefields from a real-time strategy game (Blizzard Entertainment, 2002). The six maps had 5672, 5852, 6176, 7629, 9749, and 18841 states on grids from 139×148 to 193×193. Two maps are in Figure 11. The other four maps are shown in Appendix C. The 3000 problems were uniformly distributed across five buckets where each bucket represents a range of optimal solution costs. The first 600 problems had the optimal path cost in the $[50, 60)$ range, the next 600 problems fell into the $[60, 70)$ bucket and so on until the last 600 problems that were in the $[90, 100)$ bucket.

We experimented with various assignments of algorithms (A*, LRTS, none) to levels of abstraction. Through experimentation, we found that keeping LRTS at the top, A* at the bottom level and leaving intermediate levels pass-through yielded the best results in our testbed. In the following, we present results of 160 PR LRTS configurations, denoted as LRTS$_\ell(d, \gamma, T)$, where ℓ is the level of abstraction at which LRTS with control parameters d, γ, T operates, with A* running at the bottom level. With $\ell = 0$, we run LRTS at the ground level and do not run A* at all. The LRTS parameter space was as follows: $\ell \in \{0, 1, 2, 3, 4\}$, lookahead depth $d \in \{1, 3, 5, 9\}$, optimality weight $\gamma \in \{0.2, 0.4, 0.8, 1.0\}$, and learning quota $T \in \{100.0, \infty\}$. Two visibility radii were used: 10 and 1000. In our analysis, we will focus on the case of visibility radius of 10, in line with the previous publications in the area (Bulitko et al., 2005; Bulitko & Lee, 2006). Experiments with the visibility radius of 1000 yielded similar results. As a point of reference, we ran a single non-real-time algorithm: A*. The algorithms were implemented within the Hierarchical Open Graph framework (Sturtevant, 2005) in C++ and run on a cluster, with an aggregate of 1.7 years of Intel Xeon 3.0GHz CPU.

5.1 Antagonistic Performance Measure Trade-off

From a practitioner's viewpoint, this section can be viewed as a parameter selection guide. We start by finding sets of parameters to optimize performance of PR LRTS along a single measure. Then we will consider optimizing pairs of antagonistic performance measures.

The research on optimizing single performance measures within LRTS was published by Bulitko and Lee (2006). In Table 1 we extend the results to include A* and PR LRTS. The best algorithms for a single performance measure are A* and LRTA* and do not use abstraction. The only exception is convergence planning which is an interplay between planning per move and convergence travel.

Table 1: Optimizing a single performance measure.

Measure	The best algorithm
convergence travel	A*
first-move lag (states touched)	LRTA*
conv. memory	A*
suboptimality	A* or LRTA*
conv. planning (states touched)	LRTS$_3(d = 1, \gamma = 0.2$ or $0.4, \infty)$ or LRTS$_2(d = 1, \gamma = 0.2$ or $0.4, \infty)$

The power of abstraction comes when we attempt to optimize two negatively correlated (or antagonistic) performance measures simultaneously. Consider, for instance, convergence travel and first-move lag. In order to lower convergence travel, the agent needs to select better actions. This is done by increasing the amount of planning per move which, in turn, increases its first-move lag. As these measures are negatively correlated, performance along one measure can be traded for performance along the other. Thus, we are interested in algorithms that make such trade-offs efficiently. In order to make our analysis more specific, we first introduce the concept of dominance with respect to a set of parameterized algorithms.

Definition 5.1 Algorithm A is said to *dominate* algorithm B with respect to performance measures x and y on a set of problems P if A's average performance measured in both x and y is not worse than B's: $\operatorname{avg}_P x(A)$ is not worse than $\operatorname{avg}_P x(B)$ and $\operatorname{avg}_P y(A)$ is not worse than $\operatorname{avg}_P y(B)$. Algorithm C is called *dominated* in a set of algorithms if the set contains another algorithm that dominates it.

The definition is illustrated in Figure 12 where non-dominated algorithms are shown as solid circles and dominated algorithms are shown as hollow circles. Intuitively, non-dominated algorithms make the trade-off between performance measures x and y most efficiently among all algorithms in a set. They should be considered in practice when one wants to optimize performance in both measures. Dominated algorithms can be safely excluded from consideration regardless of the relative importance of measures x and y in a particular application.

Non-dominated algorithms for ten pairs of antagonistic measures are summarized in Table 2. A* and weighted version of Korf's LRTA* are extreme cases of the performance measures: A* minimizes convergence travel and uses no heuristic memory; LRTA* minimizes first-move lag. All non-dominated algorithms between them are PR LRTS with abstraction. In other words, abstraction and path-refinement improve efficiency of trading off antagonistic performance measures. Figure 13 shows a dominance plot for convergence planning against first-move lag. PR LRTS forms a frontier

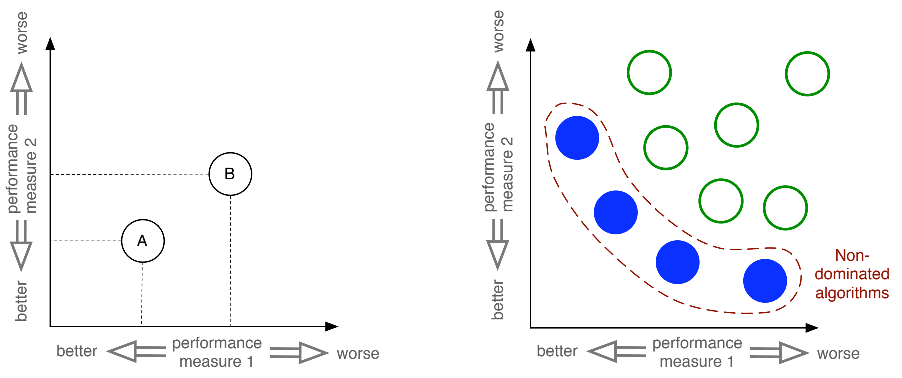

Figure 12: **Left:** algorithm A dominates algorithm B (left). **Right:** all non-dominated algorithms are shown as solid circles, dominated algorithms are shown as hollow circles.

of non-dominated algorithms (the rightmost non-dominated point is a weighted LRTA* which has an extremely low first-move lag). Plots for other combinations are in Appendix D.

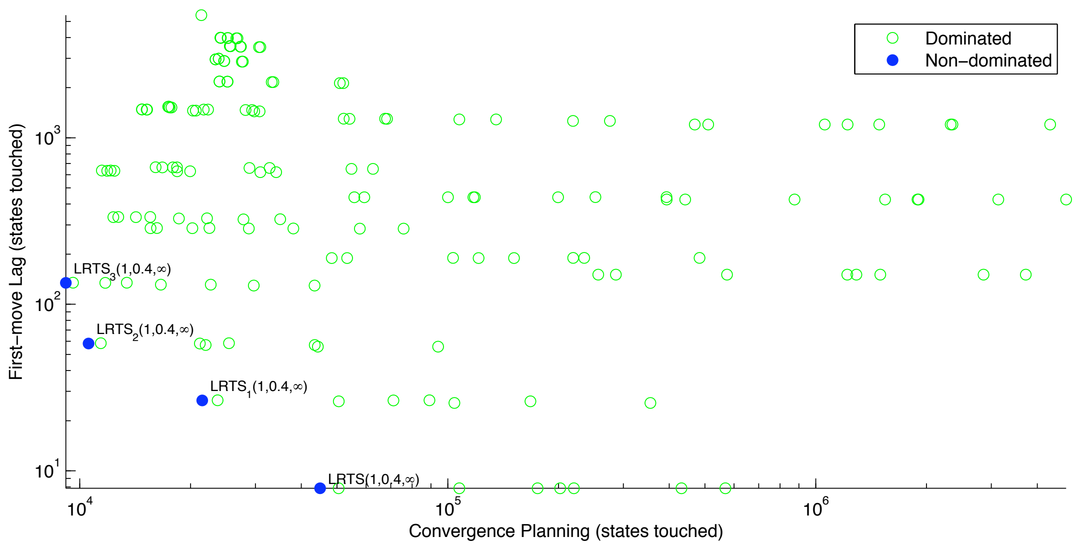

Figure 13: Dominance for convergence planning against first-move lag.

The dominance analysis above is done with respect to performance measures averaged over a benchmark set of problems. Dominance analysis at the level of *individual* problems is found in Appendix E and shows similar trends.

5.2 Effects of Abstraction on Individual Performance Measures

In this section we study effects of abstraction on individual performance measures. We arbitrarily choose three diverse LRTS parameter combinations of lookahead d, optimality weight γ, and learning quota T: $(1, 1.0, \infty), (3, 0.2, 100), (9, 0.4, \infty)$. The plots are in Figure 14, a qualitative summary is in Table 3, and an analysis of the trends is below.

Convergence planning decreases with abstraction level. This is because the increase of planning per move at higher abstraction levels is overcompensated for by the decrease in convergence travel. The exact shape of the curves is due to an interplay between these two measures.

Table 2: Trading off antagonistic performance measures.

Measure 1	**Measure 2**	**Non-dominated algorithms (extreme cases are in *italic*)**
first-move lag (states touched)	conv. travel	*A**, *LRTA**$(\gamma = 0.4)$, LRTS$_{1\ldots4}(d \in \{1,3,5,9\}, \gamma \in \{0.2,0.4,0.8\}, T \in \{100,\infty\})$
first-move lag (states touched)	conv. memory	*A**, *LRTA**$(\gamma = 0.2)$, LRTS$_{1\ldots4}(d \in \{1,3,5,9\}, \gamma \in \{0.2,0.4\}, T \in \{100,\infty\})$
first-move lag (states touched)	conv. plan. (states touched)	LRTS$_{1\ldots3}(d = 1, \gamma = 0.4, T = \infty)$, *LRTA**$(\gamma = 0.4)$
first-move lag (time)	conv. travel	*A**, *LRTA**$(\gamma = 0.4)$, LRTS$_{1\ldots4}(d \in \{1,3,5,9\}, \gamma \in \{0.2,0.4\}, T \in \{100,\infty\})$
first-move lag (time)	conv. memory	*A**, *LRTA**$(\gamma \in \{0.2,0.4\})$, LRTS$_{1\ldots4}(d \in \{1,3,5\}, \gamma \in \{0.2,0.4\}, T \in \{100,\infty\})$
first-move lag (time)	conv. plan. (time)	*A**, *LRTA**$(\gamma = 0.4)$, LRTS$_{1\ldots3}(d \in \{1,3\}, \gamma \in \{0.2,0.4\}, T \in \{100,\infty\})$
suboptimality	conv. plan. (states touched)	*A**, LRTS$_{2\ldots3}(d \in \{1,3,5\}, \gamma \in \{0.2,0.4,0.8\}, T \in \{100,\infty\})$
suboptimality	conv. plan. (time)	*A**
suboptimality	conv. travel	*A**
suboptimality	conv. memory	*A**

First-move lag increases with the abstraction level. This is due to the fact that the corridor at the ground level induced by the abstract path of length d computed by LRTS at the abstract level increases with the abstraction level. There are two additional factors affecting shape of the curves. First, the average out-degree of abstract states varies with abstraction level. Second, boundaries of abstract graphs can often be seen with lookahead of $d = 9$ at higher abstraction level.

Convergence memory decreases with abstraction level as the learning algorithm (LRTS) operates on smaller abstract maps and incurs smaller travel cost. In practice, the amount of learning in LRTS tends to correlate tightly with its travel. For instance, for LRTS$_2(3, 0.8, \infty)$ the correlation between convergence memory and convergence travel is empirically measured at 0.9544 with the confidence of 99%.

Suboptimality increases with abstraction. The increase is due to the fact that each abstraction level progressively simplifies the ground graph topology and, while the abstract path is guaranteed to be refinable into a ground path, it may lead the agent away from the shortest solution. An illustrative example is given in Appendix B.1 where a refinement of a complete abstract path is 221% longer than the optimal ground path. Derivation of a theoretical upper bound on suboptimality due to abstraction is found in Appendix B.2. A second mechanism explains why suboptimality rises faster with shallower LRTS searches. Specifically, A* at the ground level refines an abstract d-step path by finding a ground-level solution from the current state to a ground representative of the end of the abstract path. This solution is guaranteed to be optimal within the corridor and does not necessarily have to pass geographically closely to intermediate states of the abstract path. Thus, giving A* a corridor induced by a longer abstract path liberates it from having to plot a path to possibly far-off intermediate states on the abstract path. This phenomenon is illustrated in Appendix B.1 where "feeding" A* the abstract path in small fragments results in more suboptimality than giving it the "big picture" – the abstract path in its entirety. A third factor affecting the suboptimality curves

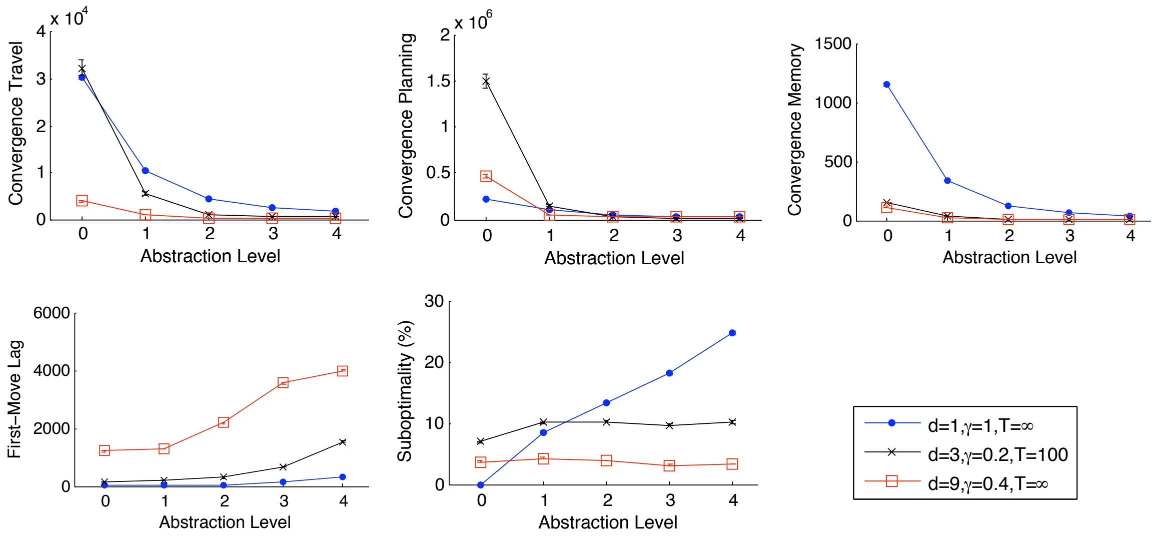

Figure 14: Effects of abstraction in PR LRTS. Error bars indicate the standard errors and are too small to see for most data points.

in the figure is the optimality weight γ. Setting γ to lower values leads to higher suboptimality independently of abstraction (Bulitko & Lee, 2006).

Convergence travel decreases with abstraction as the bottom level search is constrained within a narrow corridor induced by an abstract path. The decrease is most noticeable for shallow lookahead searches ($d = 1$ and $d = 3$). Algorithms using lookahead of $d = 9$ have low convergence travel even without abstraction, and the convergence travel is lower bounded by double the optimal solution cost (one optimal solution for the first trial at which the map is discovered and one for the final trial with no map discovery or heuristic learning). Consequently, abstraction has diminished gains for deeper lookahead ($d = 9$), although this effect would disappear on larger maps.

Table 3: Qualitative effects of abstraction: general trends.

measure / parameter	$0 \to \ell \to \infty$
first-move lag	$\uparrow$
convergence planning	$\downarrow$
convergence memory	$\downarrow$
suboptimality	$\uparrow$
convergence travel	$\downarrow$

Given that higher abstraction reduces convergence travel, one may ask how this compares to reducing convergence travel of non-abstract algorithms by simply terminating their convergence process before the final trial. In Figure 15 we compare four algorithms on a single problem: A*, non-abstract LRTS$(3, 0.4, \infty)$ and two abstract versions: LRTS$_2(3, 0.4, \infty)$ and LRTS$_4(3, 0.4, \infty)$.

The left plot in the figure demonstrates a well-known fact that convergence of learning heuristic search algorithms is non-monotinic (e.g. Shimbo & Ishida, 2003). The right plot shows the cost of the shortest solution as a function of cumulative travel. We prefer algorithms that find shorter solutions after traveling as little as possible. In the plot, the abstract algorithms perform better (lower

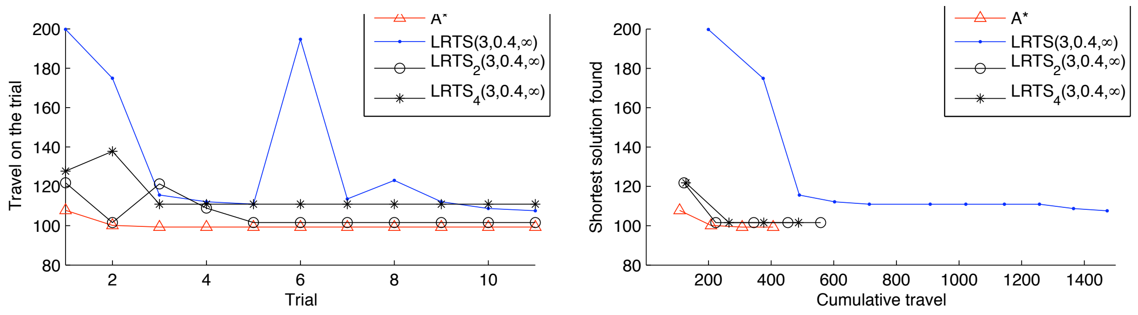

Figure 15: Convergence process at the level of individual trials.

curves) and are preferred. In other words, for this single problem, it is better to run abstract algorithms than non-abstract algorithms regardless of how early the convergence process is terminated. We observe that this is not the case for all problems and for all assignments of d, γ, T we tried. In certain cases, prematurely terminating convergence process of a non-abstract algorithm can indeed be beneficial. Future research will investigate to what extent one can automatically select the best algorithm and a number of trials to run it for.

5.3 Effects of Abstraction: Scaling up with Problem Size

In this section we investigate the effects of abstraction as the problem size increases. We measure the size of the problem as the cost of a shortest path between the start and the goal position (henceforth *optimal solution cost*).

Figures 16 and 17 show five performance measures plotted as bucket averages. For each data point, we use the middle of the bucket (e.g., $55, 65, \dots$) as the horizontal coordinate. The error bars indicate standard errors. Overall, the results demonstrate that abstraction enables PR LRTS to be applied to larger problems by significantly dampening the increase in convergence travel, convergence planning, and convergence memory. These advantages come at the price of suboptimality and first-move lag. The former clearly increases with abstraction when lookahead is small (Figure 16) and is virtually bucket-independent. The lookahead of $d = 9$ (Figure 17) draws the curves together as deeper lookahead diminishes effects of abstraction on suboptimality (cf. Figure 36).

First-move lag is virtually bucket-independent except in the case of $d = 9$ and abstraction levels of 3 and 4 (Figure 17). There, first-move lag is capped for problems in lower buckets as the goal is seen from the start state at these higher levels of abstraction. Consequently, LRTS computes an abstract path that is shorter than nine moves. This leads to a smaller corridor and less work for A* when refining the path. Consequently, the first-move lag is reduced. As the problems become larger, LRTS has room to compute a full nine-move abstract path and the first-move lag increases. For abstraction level 3 this phenomenon takes place up to bucket 85 where seeing the goal state from the start state is not frequent enough to make an impact. This does not happens with abstraction level 4 as proximity of the abstract goal continues to cut the search short even for the largest problems.

Finally, we observe a minute decrease in first-move lag for larger problems. This appears to be due to the fact that problems in the higher buckets tend to have their start state located in a cluttered region of the map (so that the optimal solution cost is necessarily higher). Walls reduce the number of states touched by the agent on its first move and reduce the first-move lag.

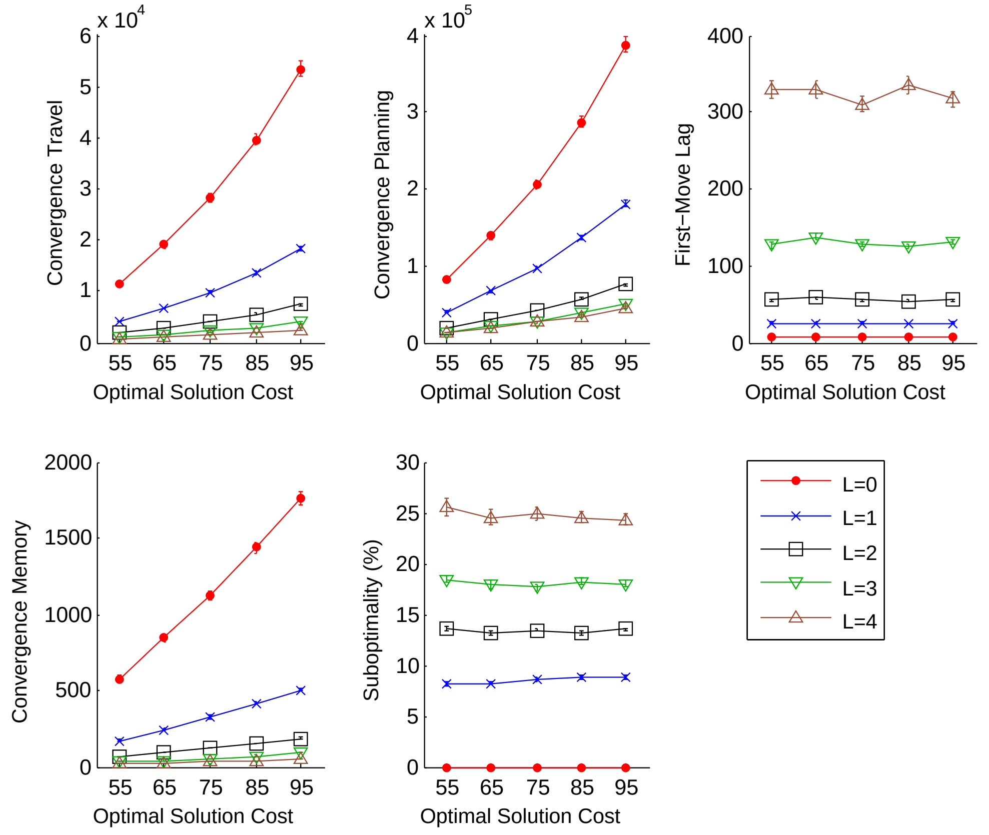

Figure 16: Scaling up. Each curve shows bucketed means for LRTS$_L(1, 1.0, \infty)$. Error bars indicate standard errors and are too small to see for most data points.

6. Theoretical Analysis

PR LRTS subsumes several known algorithms when no abstraction is used. Clearly, LRTS (Bulitko & Lee, 2006) is a special case of PR LRTS when no abstraction is used. LRTS itself subsumes and generalizes several real-time search algorithms including LRTA* (Korf, 1990), weighted LRTA* (Shimbo & Ishida, 2003), γ-Trap (Bulitko, 2004) and SLA*/SLA*T (Shue & Zamani, 1993; Shue et al., 2001).

Theorem 6.1 (real-time operation) For any heuristic search problem, LRTS$_\ell(d, \gamma, T)$ the amount of planning per any action is constant-bounded. The constant depends on the constant control parameters $d \in \mathbb{N}, \gamma \in (0, 1], T \in [0, \infty]$ but is independent of the problem's number of states.

We first prove an auxiliary lemma.

Lemma 6.1 (downward refinement property) For any abstract path $p = (s_a, \ldots, s_b)$, any two children of its ends are connected by a path lying entirely in the corridor induced by p. This means that any abstract path can be refined within the corridor formed by its children. Formally:

$$\begin{aligned} \forall 1 \leq k \leq \ell \; \forall p = (s_a, \ldots, s_b) \; [p \subset (S(k), E(k)) \implies \\ \forall s'_a \in \text{children}(s_1) \; \forall s'_b \in \text{children}(s_m) \\ \exists p' = (s'_a, \ldots, s'_b) \; [p' \subset (S(k-1), E(k-1)) \;\; \& \;\; \forall s' \in p' \; [s' \in \cup_{s \in p} \text{children}(s)]]]. \end{aligned} \quad (6.1)$$

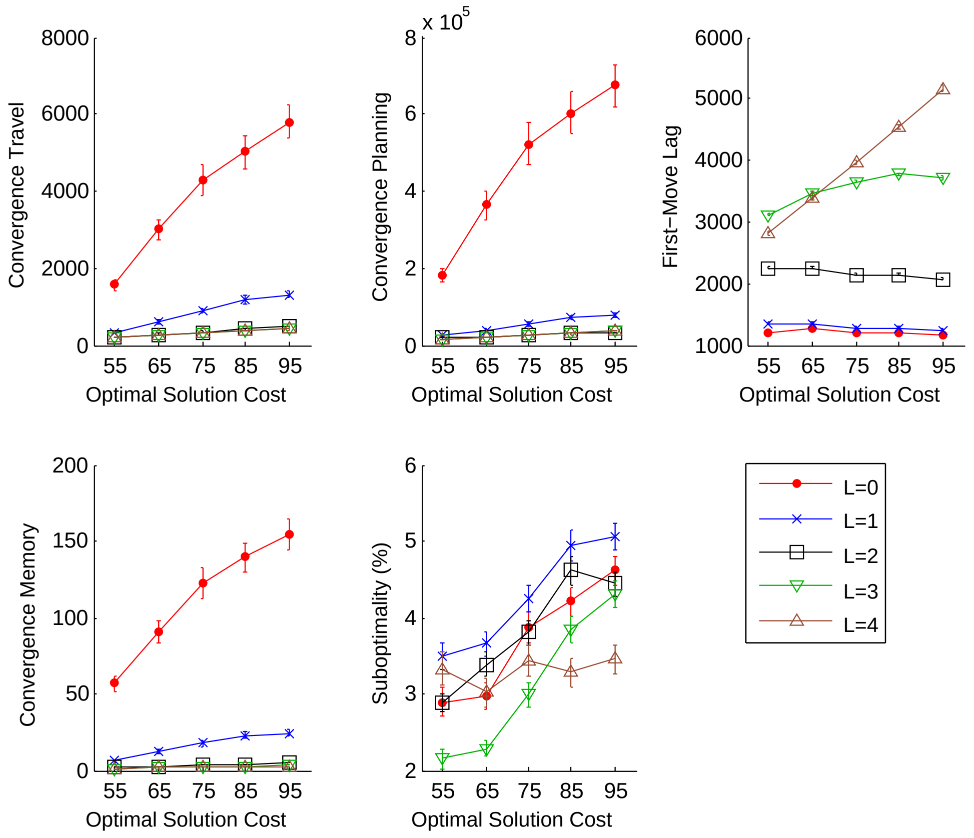

Figure 17: Scaling up. Each curve shows bucketed means for LRTS$_L(9, 0.4, \infty)$. Error bars indicate standard errors and are too small to see for most data points.

Proof. The proof is by induction on the number of edges in the abstract path. The base case is 0. This means that any two children of a single abstract state are connected by a path that lies entirely in the set of children of the abstract state. This holds due to Property 7.

Suppose the statement holds for all abstract paths of length j. We will now show that then it holds for all abstract paths of length $j+1$. Consider an arbitrary abstract path $p \subset (S(k), E(k)), k > 0$ that has $j+1$ edges. Then we can represent p as $(s_1, \ldots, s_{j+1}, s_{j+2})$. Consider arbitrary children $s'_1 \in \text{children}(s_1)$ and $s'_{j+2} \in \text{children}(s_{j+2})$. We need to show that there is path $p' \subset (S(k-1), E(k-1))$ between them that lies entirely in the union of children of all states on p (let us denote it by C_p). Let s'_{j+1} be an arbitrary child of state s_{j+1}. Since s_1 and s_{j+1} are only j edges apart, by inductive supposition, there is a path between s'_1 and s'_{j+1} that lies entirely in C_p. All that is left is to show is that s'_{j+1} and s'_{j+2} are connected within C_p. If s'_{j+1} and s'_{j+2} have the same parent, Property 7 guarantees they can be connected. If they have different parents, then Property 6 provides the same guarantee. Either way, the induction step is completed. □

We can now prove Theorem 6.1.

Proof. At the abstract level ℓ, LRTS(d, γ, T) considers no more than b^d abstract states by the algorithm design (cf. Section 2.2), here b is maximum degree of any state. As assumed earlier in the paper, the maximum degree of any state does not depend on the number of states. The resulting abstract path of no longer than d abstract edges induces a corridor at the ground level. The corridor consists of all ground-level states that abstract to abstract states on the path. The size of the corridor is upper-bounded by the number of edges in the path (at most d) multiplied by the maximum number of ground-level children of any abstract state at level ℓ. The latter is upper-bounded by a constant independent of the number of ground-level states due to Property 4. A* running in a corridor of constant-bounded size takes a constant-bounded time. Finally, abstraction repair is $O(\ell)$ and ℓ is independent of graph size (Appendix A.2). $\square$

Completeness is defined as the ability to reach the goal state on every trial. We prove completeness for LRTS$_\ell(d, \gamma, T)$ based on the following reasoning. Recall that LRTS$_\ell(d, \gamma, T)$ uses the LRTS algorithm to build an abstract path at level ℓ. It then uses a corridor-restricted A* at the ground level to refine the abstract path into a sequence of ground-level edges. Due to Property 7 of Section 4.2, A* will always be able to find a path between the ground-level states s_c and s_g that lie within the corridor C by the time execution gets to line 9 in Figure 6. Due to the exploration process, the agent's model of the search graph may be different from what the graph actually is in reality. Consequently, the path found by A* may contain a ground-level edge that the agent believes to exist but in reality does not. The following lemma demonstrates that such an execution failure is possible only a finite number of times for a given search graph:

Lemma 6.2 There are only a finite number of path execution failures on each trial.

Proof. By contradiction: suppose there are an infinite number of such failures. Each failure is due to a discovery of at least one new blocked edge or vertex in the ground-level graph. Then there will be infinitely many blocked edges or vertices in a finite graph. $\square$

A direct corollary to this lemma is that for any trial, there will be a moment of time after which no graph discoveries are made on that trial. Therefore, executing A*'s path will indeed allow the agent to follow its abstract path on the actual map.

Lemma 6.3 LRTS is complete on an abstract graph.

Proof. First, we show that any abstract graph satisfies the properties under which LRTS is shown to be complete (Theorem 7.5, Bulitko & Lee, 2006). That is, the abstract graph is finite, each action is reversible, there are no self-loops, all actions have a positive cost, and the goal state is reachable from every state. The graph also has to be stationary and deterministically traversible (p.122, Bulitko & Lee, 2006). Due to abstraction mechanism requirements in Section 4.2, the properties listed above are satisfied by the clique abstraction mechanism as long as the ground-level graph satisfies these properties as well (which we require in Section 2). Thus, LRTS running on an abstract graph as if it were a ground graph is complete.

In PR LRTS, however, LRTS on an abstract graph does not execute its own actions. Instead, its current (abstract) state is computed as abstract parent of the agent's current ground-level state. Therefore, a critical question is whether the agent is able to find a ground-level path from its current state to the ground-level state corresponding to the end of its abstract path as computed in line 6 of Figure 6. The failure to do so would mean that the corridor computed in line 8 of Figure 6 and used to refine the path does not contain a ground-level path from s_c to s_g. Due to the downward refinement property (Lemma 6.1), this can only be due to graph discovery.

According to Lemma 6.2, after a finite number of failures, the A* algorithm operating at the ground level is guaranteed to find path to reach the end of the abstract path computed by LRTS. Thus, LRTS has the effective ability to "execute" its own abstract actions. Putting these results together, we conclude that for any valid d, γ, T parameters, LRTS on the abstract graph finds its goal on every trial. □

The two lemmas lead directly to the following statement.

Theorem 6.2 LRTS$_\ell(d, \gamma, T)$ is complete.

Proof. This follows directly from Lemma 6.2 and Lemma 6.3. □

Theorem 6.3 LRTS$_\ell(d, \gamma, T)$ with fixed tie-breaking converges to its final solution after a finite number of trials. On all subsequent trials, it does not update its search graph model or the heuristic and follows the same path.

Proof. Follows from Lemma 6.2 and Theorem 7.6 of (Bulitko & Lee, 2006) in the same way Lemma 6.3 and Theorem 6.2 were proved above. □

Theoretical results on suboptimality are found in Appendix B.2

7. Related Research

Existing heuristic search methods for situated methods can be divided into two categories: full search and real-time search. Full-search algorithms form an entire solution given their current knowledge of the search graph. In contrast, real-time search plans only a small segment (frequently just the first action) of their solution and executes it right away. Due to the local nature of planning, real-time search algorithms need to update the heuristic function to avoid getting stuck in local minima of their heuristic function.

7.1 Full Search

A common full-search algorithm is a version of A* (Hart et al., 1968) called Local Repair A* (Stout, 1996). In it, a full search is conducted from agent's current state to the goal state under the free space assumption. The agent then executes the computed path until either the destination is reached or the path becomes invalid (e.g., a previously unknown wall blocks the way). In the latter case, the agent replans from its current position to the goal. Local Repair A* suffers from two problems. First, it searches for a shortest solution and, for a general search problem, may end up expanding a number of states exponential in the solution cost due to inaccuracies in the heuristic function (Pearl, 1984). Second, re-planning episodes do not re-use results of previous search.

The first problem is addressed by suboptimal versions of A* which are frequently implemented via weighting the heuristic function (Pohl, 1970, 1973). Such a weighted A* (WA*) usually finds a longer solution in less time. Once a suboptimal solution is found, it can be improved upon by conducting additional searches. This can be done by re-using the open list between successive searches (Hansen, Zilberstein, & Danilchenko, 1997; Likhachev et al., 2004; Hansen & Zhou, 2007) or by re-running A* in a tunnel induced by a suboptimal solution (Furcy, 2006). In the later case, beam search with backtracking can be used in place of weighted A* (Furcy & Koenig, 2005).

The second problem is addressed by incremental search methods such as D* (Stenz, 1995), D* Lite (Koenig & Likhachev, 2002a) and LPA* (Koenig & Likhachev, 2002b). These algorithms reuse some information from the previous search, thus speeding up subsequent replanning episodes.

In all of these algorithms, a full path has to be computed before the first move can be executed by the agent. Consequently, the planning time per move is not constant-bounded and increases with the problem size. Thus, agent-centered full search is not real-time.

7.2 Learning Real-time Search

Since the seminal work on LRTA* (Korf, 1990), research in the field of learning real-time heuristic search has flourished resulting in over twenty algorithms with numerous variations. Most of them can be described by the following four attributes:

The **local search space** is the set of states whose heuristic values are accessed in the planning stage. The two common choices are full-width limited-depth lookahead (Korf, 1990; Shimbo & Ishida, 2003; Shue & Zamani, 1993; Shue et al., 2001; Furcy & Koenig, 2000; Hernández & Meseguer, 2005a, 2005b; Sigmundarson & Björnsson, 2006; Rayner, Davison, Bulitko, Anderson, & Lu, 2007) and A*-shaped lookahead (Koenig, 2004; Koenig & Likhachev, 2006). Additional choices are decision-theoretic based shaping (Russell & Wefald, 1991) and dynamic lookahead depth-selection (Bulitko, 2004; Luštrek & Bulitko, 2006).

The **local learning space** is the set of states whose heuristic values are updated. Common choices are: the current state only (Korf, 1990; Shimbo & Ishida, 2003; Shue & Zamani, 1993; Shue et al., 2001; Furcy & Koenig, 2000; Bulitko, 2004), all states within the local search space (Koenig, 2004; Koenig & Likhachev, 2006) and previously visited states and their neighbors (Hernández & Meseguer, 2005a, 2005b; Sigmundarson & Björnsson, 2006; Rayner et al., 2007).

A **learning rule** is used to update the heuristic values of the states in the learning space. The common choices are dynamic programming or mini-min (Korf, 1990; Shue & Zamani, 1993; Shue et al., 2001; Hernández & Meseguer, 2005a, 2005b; Sigmundarson & Björnsson, 2006; Rayner et al., 2007), their weighted versions (Shimbo & Ishida, 2003), max of mins (Bulitko, 2004), modified Dijkstra's algorithm (Koenig, 2004), and updates with respect to the shortest path from the current state to the best-looking state on the frontier of the local search space (Koenig & Likhachev, 2006). Additionally, several algorithms learn more than one heuristic function (Russell & Wefald, 1991; Furcy & Koenig, 2000; Shimbo & Ishida, 2003).

Control strategy decides on the move following the planning and learning phases. Commonly used strategies include: the first move of an optimal path to the most promising frontier state (Korf, 1990; Furcy & Koenig, 2000; Hernández & Meseguer, 2005a, 2005b), the entire path (Bulitko, 2004), and backtracking moves (Shue & Zamani, 1993; Shue et al., 2001; Bulitko, 2004; Sigmundarson & Björnsson, 2006).

Given the multitude of proposed algorithms, unification efforts have been undertaken. In particular, Bulitko and Lee (2006) suggested a framework, called Learning Real Time Search (LRTS), to combine and extend LRTA* (Korf, 1990), weighted LRTA* (Shimbo & Ishida, 2003), SLA* (Shue & Zamani, 1993), SLA*T (Shue et al., 2001), and to a large extent, γ-Trap (Bulitko, 2004). In the dimensions described above, LRTS operates as follows. It uses a full-width fixed-depth local search space with transposition tables to prune duplicate states. LRTS uses a max of mins learning rule to update the heuristic value of the current state (its local learning space). The control strategy moves the agent to the most promising frontier state if the cumulative volume of heuristic function updates on a trial is under a user-specified quota or backtracks to its previous state otherwise (Section 2.2).

Within LRTS, the unification of several algorithms was accomplished through implementing several methods for local search space selection, the learning rule, and the control strategy. Each of these methods can be engaged at run-time via user-specified parameters. The resulting parameter space contained all the original algorithms plus numerous new combinations, enabling tuning performance according to a specific problem and objective function of a particular application. As a demonstration, Bulitko et al. (2005) tuned LRTS for ten maps from the computer game "Baldur's Gate" (BioWare Corp., 1998) and achieved a convergence speed that is two orders of magnitude faster than LRTA*, while finding paths within 3% of optimal. At the same time, LRTS was about five times faster on the first move than incremental A*. Despite the improvements, LRTS and other real-time search algorithms converge more slowly than A* and, visually, may behave unintelligently by repeatedly revisiting dead-ends and corners.

7.3 State Abstraction

The idea of abstraction has been previously applied to full search methods. In particular, HPA* and PRA* (Botea et al., 2004; Sturtevant & Buro, 2005) use abstraction to speed up A* search: instead of running A* on the lowest-level graph, they instead run A* on a smaller abstract graph. PRA* computes an abstract path and then refines it in a similar manner to PR LRTS. However, PRA* dynamically chooses which abstract level to use, and computes a path at each intermediate level (i.e., it does not have pass-through levels). PRA* also widens its corridors to decrease suboptimality at the cost of lower speed.

HPA* abstracts a map using large regions, and selects connection points (gates) between neighboring regions. For all gates of a region, optimal paths between all gates are pre-computed off-line using A* and are stored in a table. This means that refining an abstract path (i.e., a sequence of region gates) can be done simply by concatenating stored optimal paths. Smoothing is applied as a post-processing step to decrease suboptimality of the resulting path.

Both of these algorithms are based on the ideas presented by Holte et al. (1996), who used an abstraction mechanism in a similar manner to our use of the clique abstraction. Their method, the STAR abstraction, can also be described as a radius abstraction. That is, a state is selected, and is aggregated together with all states in a fixed radius of the original state. Holte et al. (1996)'s work did not initially gain wide acclaim, because, at the time, there was little interest in problems which were small enough to fit in memory. Motivating applications, such as pathfinding in computer games, have resulted in a resurgence of interest in such techniques.

This class of algorithms first plan an abstract path, which is then refined into a traversable path. Another approach is to build an abstraction which can be directly used for planning in the real-world. This includes methods like framed quad-trees (Yahja, Stentz, Singh, & Brummit, 1998), which efficiently represent sparse maps. Quad-trees are a multi-resolution representation, as some areas of the map are represented at high-resolution, and others are represented at lower resolution. This abstraction differs from abstractions like the clique abstraction in that it can only be applied once; further applications would not produce lower resolution maps, although the clique abstraction could be applied to the graph implied by the framed quad-tree representation.

One other common use of abstraction is to provide better heuristics. Holte, Perez, Zimmer, and MacDonald (1995) used the result of an abstract search to provide a more accurate heuristic for low-level search and performed no path refinement. Similarly, pattern databases are abstractions which are built and solved off-line. The abstract solution costs are stored and then used during search as a heuristic function (Culberson & Schaeffer, 1996; Felner, Zahavi, Schaeffer, & Holte, 2005).

PR LRTS presented in this paper is the first *real-time* heuristic search algorithm to use automatically-built state abstraction. Path-refinement algorithms listed above conduct a full-search and therefore cannot guarantee constant-bounded planning time for all agent's moves.

8. Limitations and Future Work

The results presented in this paper open several directions for future research. First, PR LRTS is able to operate with a wide class of homomorphic graph abstraction techniques. Thus, it would be of interest to investigate the extent to which effects of graph abstraction on real-time search presented in this paper are specific to the clique abstraction mechanism and the pathfinding domain. Recent work has shown that the clique abstraction has parameters that are well-tuned to minimize work done in traditional path planning (Sturtevant & Jansen, 2007). Our experiments in pathfinding have suggested that the clique abstraction is well-suited to map abstraction because it represents key properties of the underlying space well. In particular, the branching factor stays roughly constant at higher levels of abstraction. On an empty map, for instance, the number of nodes at each level of abstraction will be reduced by a factor of four by the clique abstraction, but the branching factor of every state will stay the same. (Corner states will have 3 neighbors, edge states will have 5 neighbors, and middle states will have 8 neighbors.) This may not be the case in other domains. For instance, in the sliding tile puzzle the maximum branching factor of abstract states quickly increases with abstraction level. As a result, the corridor derived from an abstract path in PR LRTS becomes excessively wide and does not effectively constrain A* search at the ground level. We conjecture that algorithms which use homomorphic abstractions will only be effective in a domain if the abstraction preserves the average, minimum, and maximum branching factor from the original problem at each level of abstraction. Clique abstraction, then is likely to work well in three-dimensional pathfinding, while problem-specific mechanisms would be needed for permutation-type puzzles. It is an area of open research to provide such an abstraction.

Second, PR LRTS uses an abstract solution to restrict its search in the original ground-level graph. It is interesting to combine this with a complementary approach of using the cost of an optimal solution to an abstract problem as a heuristic estimate for the original search graph in the context of real-time search. In particular, we are looking at effective ways of propagating heuristic values from higher to lower levels of the abstraction hierarchy.

Third, state aggregation is just one way of generalizing learning. Future research will consider combining it with function approximation for the heuristic function, as is commonly practiced in large-scale applications of reinforcement learning.

Fourth, we are presently investigating applications of PR LRTS to dynamic environments. In particular, we are studying the extent to which savings in memory gained by learning at a higher abstraction level will afford application of PR LRTS to moving target search. An existing algorithm (Ishida & Korf, 1991) requires learning a number of heuristic values quadratic in the size of the map. This is prohibitive in the case of commercial game maps.

Finally, we are presently extending the graph abstraction method presented in this paper to stochastic environments formulated as Markov Decision Processes.

9. Conclusions

Situated agents in real-time environments are expected to act quickly while efficiently learning an initially unknown environment. Response time and learning speed are antagonistic performance measures as more planning leads to better actions and, consequently, faster convergence but longer response time. Full search algorithms, such as local repair A*, converge quickly but do not have a constant-bounded planning time per move. Real-time heuristic search algorithms have constant-bounded planning times per move, but learn slowly.

In this paper, we attempted to combine the best of both approaches and suggest a hybrid algorithm, PR LRTS, that learns a heuristic function in a smaller abstract space and uses corridor-restricted A* to generate a partial ground-level path. In a large-scale empirical study, PR LRTS was found to dominate virtually all tested algorithms that do not use abstraction with respect to several performance measure pairs. The combination of learning and planning brings real-time performance to much larger search spaces, substantially benefiting applications such as pathfinding in robotics and video games.

Acknowledgments

Funding was provided by the National Science and Engineering Research Council of Canada, the Alberta Ingenuity Centre for Machine Learning, the Informatics Circle of Research Excellence, and the University of Alberta. We appreciate input from David Thue, Rich Korf, Rob Holte, Csaba Szepesvári, David Furcy, and Adi Botea. Special thanks go to Jonathan Schaeffer and the anonymous reviewers whose detailed comments greatly improved this paper. This research has been enabled by the use of WestGrid computing resources, which are funded in part by the Canada Foundation for Innovation, Alberta Innovation and Science, BC Advanced Education, and the participating research institutions. In particular, we would like to acknowledge help from Roman Baranowski, Masao Fujinaga, and Doug Phillips.

Appendix A. Clique Abstraction

Below we will describe the clique abstraction mechanism in several stages. First, we present the algorithm for building an initial abstraction hierarchy using the free space assumption. Then we describe the repair procedure that updates the abstract graphs as an agent explores the environment. Finally, we consider suboptimality of solution caused by abstraction on examples and derive a worst-case upper bound.

A.1 Building Initial Abstraction Hierarchy

The pseudo-code for building an initial clique abstraction is in Figure 18. The $\mathrm{abstract}$ procedure (lines 5 and 14) takes a set of states at some level i and maps them to a single abstract state at level $i + 1$. This involves creating a new abstract state and storing parent and child links. If, in line 20, a new abstract edge is added where one already exists, we do not add an extra edge but increase a count associated with the edge. Such counts are used to facilitate abstraction repair as described in the next section.

In general, clique-finding is an NP-complete problem (Garey & Johnson, 1990). However, in eight-connected two-dimensional grid-based search graphs the largest possible clique size is 4.

```
graph **CliqueAbstraction**(graph $g$)
  initialize graph $g' \leftarrow \emptyset$
  for $i$ = 4...2
    for each unabstracted state $s$ in $g$
      if $s$ is part of some $i$-clique $c$
        $g' \leftarrow g' \cup \{\text{abstract}(c)\}$
      end if
    end for each
  end for

  for each unabstracted state $s$ in $g$
    if degree($s$) = 1
      set $\text{parent}(s)$ to $\text{parent}(\text{neighbor}(s))$
    else
      $g' \leftarrow g' \cup \{\text{abstract}(n)\}$
    end if
  end for each

  for each edge $e$ = ($v_1$, $v_2$)
    if $\text{parent}(v_1) \neq \text{parent}(v_2)$
      $g' \leftarrow g' \cup \{(\text{parent}(v_1), \text{parent}(v_2))\}$
    end if
  end for each
return $g'$
```

Figure 18: Building the initial clique abstraction.

Because the degree of each state is also constant-bounded (as required in Section 2) the time per clique constant (i.e., $\binom{8}{3}$ state accesses to check eight neighbors and find 3 which form a 4-clique together with the current state). Thus, the total running time for doing a single clique abstraction is $O(|S|)$, where $|S|$ is the number of states in the original search graph. If the abstraction procedure reduces the graph size by at least a constant factor greater than one, the total cost of abstracting a graph will also be $O(|S|)$ as the cost of each additional abstraction step is reduced exponentially.

A.2 Repairing Abstraction Hierarchy

As the agent explores its environment, it may find some edges or states blocked. In such cases, it will remove the corresponding states and edges from its model and will need to propagate the changes to all abstract graphs in the abstraction hierarchy. We will demonstrate the repair code in Figure 19 with an example that also shows how the repair procedure can have amortized constant-time cost.

In Figure 20 we remove edges from the Level 0 graph at the bottom left of the figure. The right side of the figure shows the full abstract graph after all edges have been removed. At each level we show a portion of the abstract graph and assume that there are more states and edges in the graph. They are shown schematically in Level 0 in gray.

At Level 0 there are four states marked A which form an abstract state at level 1. This is also true for the states marked A'. A and A' are joined by an edge at level 1, which is abstracted from

```
RemoveEdge(edge $e$, level $l$, graph $g$)
 1   decrease edge count of $e$
 2   if child edge count of $e \neq 0$ return end if
 3   $e = (v_1, v_2)$
 4   remove $e$ from $g[l]$
 5   if parent($v_1$) = parent($v_2$)
 6     AddToRepairQ(parent($v_1$))
 7   else
 8     RemoveEdge((parent($v_1$), parent($v_2$)), $l + 1$, $g$)
 9   end if
RemoveState(state $s$, level $l$, graph $g$)
10   for each edge $e$ incident to $s$
11     RemoveEdge($e$, $l$, $g$)
12   end if
13   remove $s$ from $g[l]$
14   AddToRepairQ(parent($s$))
HandleRepairQ()
15   while RepairQ not empty
16     remove state $s$ at lowest level $l$ of $g$
17     if abstraction properties do not hold in $s$
18       AddToRepairQ(parent($s$))
19       split state $s$ into $s_1 \ldots s_n$ so that abstraction properties holds in $s_i$
20       for $i = 1 \ldots n$ either:
21         1. Merge $s_i$ into existing abstract state
22         2. Extract $s_i$ into new abstract state
23       end for
24     end if
25   end while
```

Figure 19: Repairing abstraction hierarchy.

four edges at level 0. When we remove these edges from the level 0 graph using the RemoveEdge() procedure from Figure 19, the first three removals simply decrement the count associated with the abstract edge between A and A' (line 1-2). The fourth removal, however, will result in removing the edge between A and A' (line 4). This removal will be recursively propagated (line 8) into the abstraction hierarchy, but will not change the abstract graph at level 2, because the edge count will again be decremented.

There are 22 edges which must be removed to perform the full split between the top and bottom states at level 0 in Figure 20. Removing the first edge between E and E' at level 2 requires the removal of 10 underlying edges at level 0 which correspond to 4 edges from level 1 (all edges between A, B, A' and B').

State repair first occurs when we remove the edge from E to E'. In this case E and E' have the same parent, so G is added to the repair queue (line 5 and 6). The repair queue is processed after each set of removal operations. Once the edge from E to E' is removed, children of G at level 3 no longer form a clique. Thus, G must be split into two states H and H'. Initially these states will

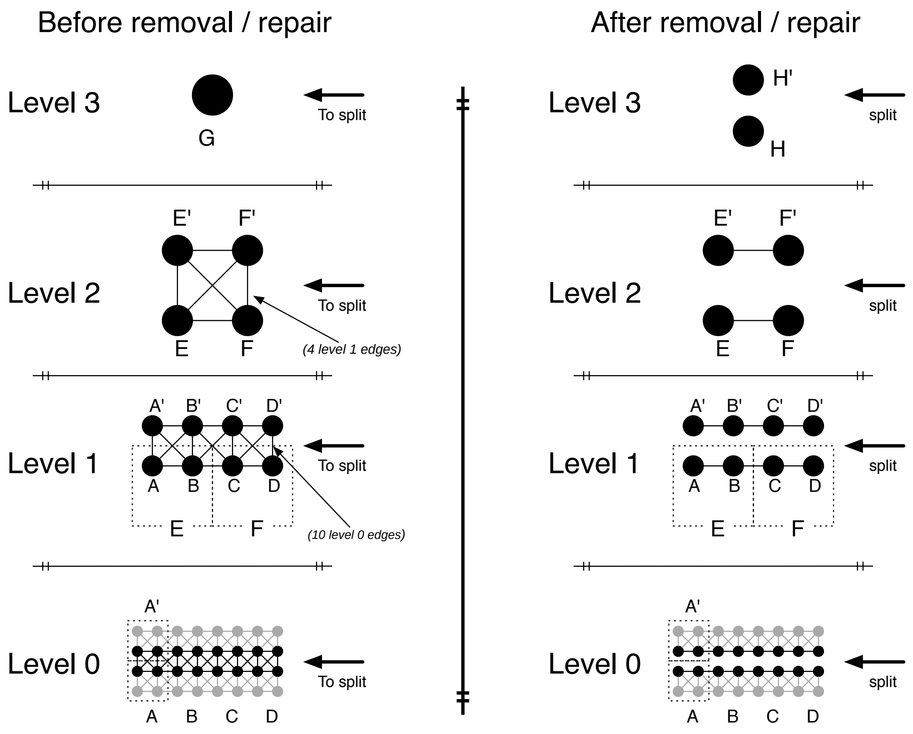

Figure 20: An example of repairing the clique abstraction.

have an edge between them, but this edge will be removed when the last of the edges from level 0 are removed. The repair code can work with many abstraction mechanisms. Specifically, the check that an abstract state's children still form a clique (line 17) can be changed to check for the corresponding property of a non-clique abstraction.

For this example, the amortized cost of abstraction repair is constant. Imagine an agent traversing the graph at level 0 from left to right and discovering a wall splitting the top and bottom rows of the states (as shown by "To split" label in the figure). At each step more of the graph is sensed by the agent and edges will be removed from level 0 graph. Removing the three edges (A, A'), (A, B'), and (B, A') at level 1 requires removing six edges at level 0. Similarly, removing the three edges (E, E'), (E, F'), and (F, E') requires removing 12 edges at level 0. In general, an agent traveling at level 0 must move twice as far (or remove twice as many states) before repair is required at an additional level of abstraction. Thus, the number of abstraction levels repaired when traversing n ground edges, is:

$$\frac{n}{2}1 + \frac{n}{4}2 + \frac{n}{8}3 + \frac{n}{16}4 + \cdots + \frac{n}{n}\log_2(n) = O(n).$$

Consequently, in this example, the amortized repair cost per edge traveled is $O(1)$. In general, the worst-case complexity of repair is $O(\ell)$ and, in PR LRTS, ℓ is a constant independent of graph size. This is because repairs are propagated only up the abstraction hierarchy (line 18 in Figure 19).

Appendix B. Abstraction Properties

Abstraction properties were informally introduced and illustrated with an example in Section 4.2. The appendix makes the presentation mathematically precise. In this section, variables i, k, m run natural numbers including 0.

Property 1. An agent using abstraction maintains a hierarchy of ℓ abstract search graphs in addition to its model of the environment. Each of the abstract graphs is a search graph in the sense of Section 2. In the following we denote the abstract search graph at level $i, 1 \le i \le \ell$ by $(G(i), c(i), s_0(i), s_g(i), h_0(i))$. As before, $G(i) = (S(i), E(i))$.

Property 2. Each state s in the search graph at level $n < \ell$ has a unique "parent" state s' in level $n+1$ abstract search graph. More formally:

$$\forall s \in S(k), k < \ell \; \exists! s' \in S(k+1) \left[\text{parent}(s) = s'\right]. \tag{B.1}$$

Property 3. Each state s in search graph at level m, for $0 < m \le \ell$, has at least one "child" state s' at level $m-1$. The notation $\text{children}(s)$ represents the set of children of state s. Thus, $s' \in \text{children}(s)$:

$$\forall s \in S(k), k > 0 \; \exists s' \in S(k-1) \left[s' \in \text{children}(s)\right]. \tag{B.2}$$

Property 4. Given a heuristic search problem $\mathcal{S}$, for any instance of that problem, the number of children of any abstract state is upper-bounded by a constant independent of the number of states:

$$\forall \mathcal{S}, \mathcal{S} \text{ is a search problem } \exists m \; \forall ((S,E), c, s_0, s_g, h_0) \in \mathcal{S} \; \forall i, 0 < i \le \ell \; \forall s \in S(i) \\ \left[|\text{children}(s)| < m\right]. \tag{B.3}$$

Property 5. **(Graph homomorphism)** Every edge $(s_1, s_2) \in E(k), k < n$ has either a corresponding abstract edge at level $k+1$ or s_1 and s_2 abstract into the same state:

$$\forall s_1, s_2 \in S(k), k < \ell \\ \left[(s_1, s_2) \in E(k) \implies (\text{parent}(s_1), \text{parent}(s_2)) \in E(k+1) \vee \text{parent}(s_1) = \text{parent}(s_2)\right]. \tag{B.4}$$

Property 6. If an edge exists between abstract states s_1 and s_2 then there is an edge between some child of s_1 and some child of s_2:

$$\forall s_1, s_2 \in S(k), k > 0 \\ \left[(s_1, s_2) \in E(k) \implies \exists s_1' \in \text{children}(s_1) \; \exists s_2' \in \text{children}(s_2) \; (s_1', s_2') \in E(k-1)\right]. \tag{B.5}$$

For the last property, we need the following definition.

Definition B.1 A *path* p in space $(S(k), E(k))$, $0 \le k \le \ell$ is defined as an ordered sequence of states from $S(k)$ whose any two sequential states constitute a valid edge in $E(k)$. Formally, p is a path in $(S(k), E(k))$ if and only if:

$$\exists s_1, \dots, s_m \in S(k) \left[p = (s_1, \dots, s_m) \;\; \& \;\; \forall i, 1 \le i \le m \left[(s_i, s_{i+1}) \in E(k) \right] \right]. \tag{B.6}$$

We use the notation $p \subset (S(k), E(k))$ to indicate that both the vertices and the edges of the path p are in the sets $S(k)$, $E(k)$ respectively. The notation $s \in p$ indicates that state s is on the path p.

Property 7 Any two children of an abstract state are connected through a path whose states are all children of the abstract state:

$$\forall s \in S(k), 0 < k \le \ell \; \forall s_1', s_2' \in \text{children}(s) \; \exists p = (s_1', \dots, s_2') \subset (S(k-1), E(k-1)). \tag{B.7}$$

B.1 Abstraction-induced Suboptimality: Examples

Abstraction can cause suboptimality. In Figure 21 left, we are refining an abstract path. Solid arrows indicate the abstract path. Ground-level path is shown with thinner dashed arrows. The agent's position is shown with "A" and the goal's position is "G". The white cells form the corridor induced by the abstract path. An optimal path is shown on the right.

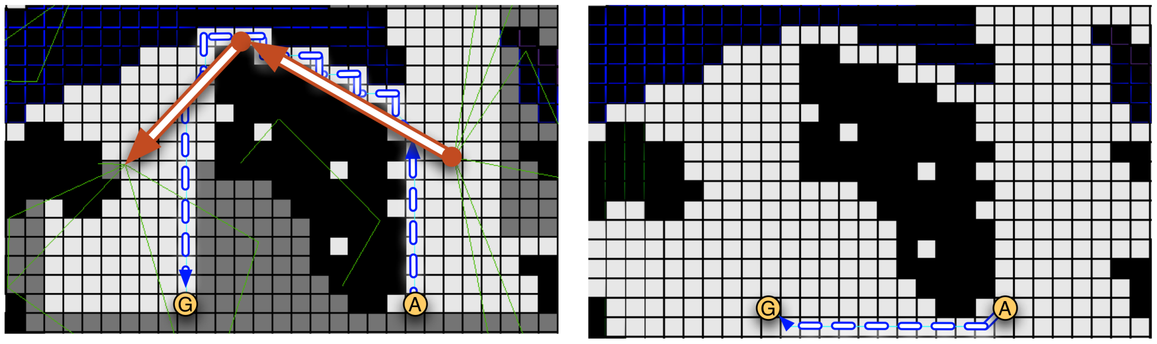

Figure 21: Abstraction causes suboptimality.

Partial path refinement can increase suboptimality. Refining an entire abstract path (Figure 22, left) can yield shorter paths than refining a segment of an abstract path (Figure 22, right). Solid arrows indicate the abstract path. The ground-level path is shown with thinner dashed arrows. The agent's position is shown with "A" and the goal's position is "G".

B.2 Abstraction-induced Suboptimality: An Upper Bound

There are two factors that contribute to the suboptimality of paths returned by PR LRTS. The first factor is the parameters chosen for LRTS, which can be weighted to allow suboptimality. This effect has been analyzed in the literature (Bulitko & Lee, 2006). Here we analyze the suboptimality that can be introduced by the abstraction. For simplicity of analysis we consider a uniform abstraction where at each level k states are abstracted into a parent at the next level of abstraction. This assumption simplifies our analysis and also enables application of this analysis to non-clique abstraction mechanism that maintain this property. Before proving our result, we introduce two simple lemmas:

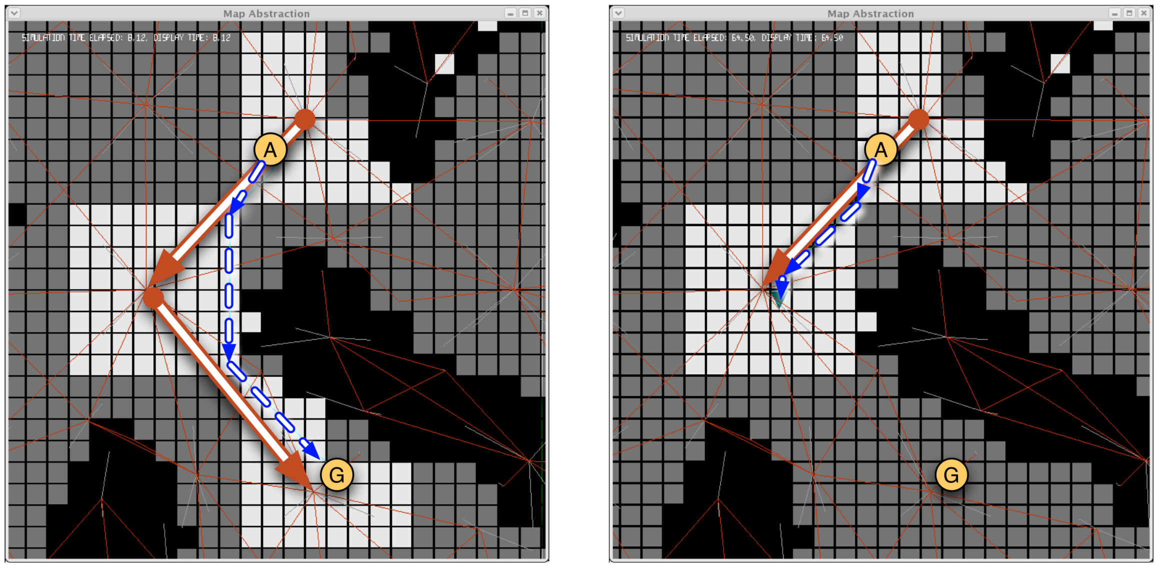

Figure 22: Partial path refinement increases suboptimality.

Lemma B.1 Suppose all abstract edges have the same cost. If a lowest-cost path p between states A and B has j edges then a lowest-cost abstract path between their abstract parents A' and B' has at most j abstract edges.

Proof. We prove this by contradiction. Suppose the lowest-cost abstract path q between A' and B' has $m > j$ edges. Then consider the abstract images of all states on p. They either have an abstract edge between them or coincide due to Property 5. Thus they form an abstract path p' between A' and B' and due to Property 2 it has no more than j edges. Since by the assumption of our theorem all abstract edges have the same cost, the lowest-cost path q between A' and B' must have a higher cost than path p' between A' and B' (Figure 23, right). This results in a contradiction. □

Lemma B.2 Any path created by refining an abstract edge at level ℓ cannot be longer than $O(k^\ell)$ at level 0.

Proof. This is demonstrated in the right portion of Figure 24. We assume that every abstract state has exactly k children. So, at level ℓ of the abstraction any state A cannot have more than k^ℓ children. Assuming that a path cannot visit a single node more than once, the refined path through A can therefore have no more than $O(k^\ell)$ edges. □

We can now present our main result:

Theorem B.1 Assume that every abstract state has k children and the ground level edge costs are in $[1, e]$. At level ℓ of an abstraction, the cost of a path created by refining an abstract path from level ℓ to level 0 (the original space) is at most $O(ek^\ell)$ times more costly than the optimal path if all abstract edges happen to have uniform cost and $O(e^2k^{2\ell})$ if all abstract edges have costs in $[1, ek^\ell]$ (from Lemma B.2).

Proof. First, we deal with the case where all edges in the *abstract* graph have uniform cost. Consider two level-0 states A and B that abstract into level-ℓ states A', B' (left side of Figure 23). If a lowest-cost path p between A and B has j edges then a lowest-cost abstract path between A' and B' has at most j abstract edges by Lemma B.1.

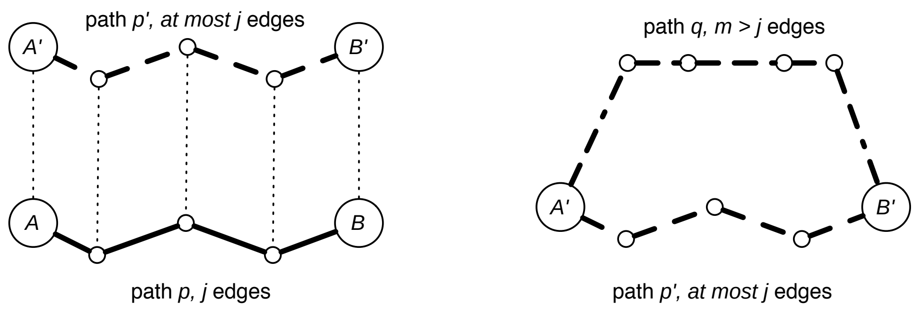

Figure 23: Proving that any lowest-cost abstract path will have no more than j edges.

Suppose the agent is in state A and is seeking to go to state B. The agent first computes a lowest-cost abstract path between A' and B'. In the worst case, the abstract path will have j edges. Suppose there are two abstract paths between A' and B': t_1' and t_2' as shown in Figure 24, left. They both have j edges and, due to the uniform abstract edge cost assumption, the same cost. In the worst case scenario, t_1' is refined into a lowest-cost path t_1 between A and B while t_2' is refined into the highest-cost path t_2 between A and B. By analyzing the cost ratio of t_1 and t_2 we will arrive at the upper bound of the theorem.

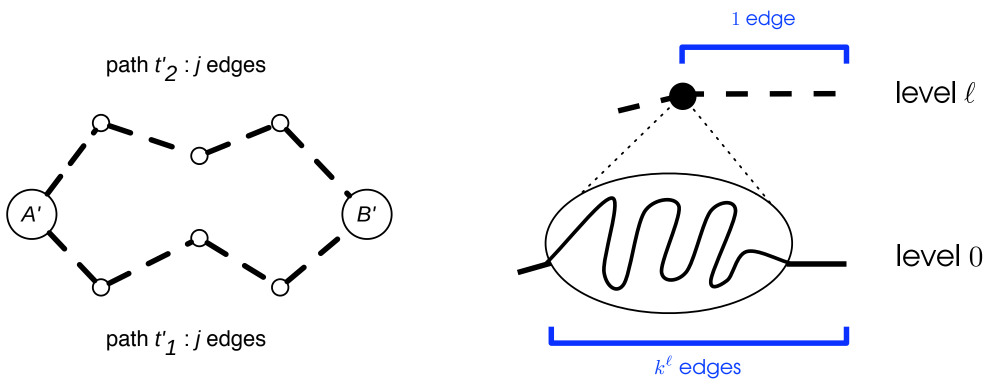

Figure 24: Paths t_1', t_2' are the same cost yet refine into the shortest and longest paths.

Due to Lemma B.2, one abstract edge at level ℓ can be refined into at most k^ℓ level-0 edges. In the worst case, t_1 has j edges while t_2 has $jk^\ell - 1$ edges (the result of abstracting ℓ levels of k^ℓ states into a single state). Furthermore, all edges on t_1 have a cost of 1 leading to the total cost of t_1 being j. All edges on t_2 have cost e leading to the total cost of t_2 being $e(jk^\ell - 1)$. Thus, the ratio of t_2 and t_1 costs is no higher than ek^ℓ which proves the first statement of the theorem.

In the case when abstract edges have non-uniform costs in $[1, ek^\ell]$, we again consider two abstract paths t_1' and t_2' from A' to B'. Now they can both have cost ejk^ℓ, which is the highest possible cost of a level-ℓ image of a cost-j ground path. On path t_1', the abstract cost might be overestimated so that there are j abstract edges, each of cost ek^ℓ which refine into the level-0 path t_1 of j edges of cost 1 each. Thus, the total cost of t_1 is j which is the lowest possible cost between A and B. Path t_2' has the same cost as t_1' but with ejk^ℓ abstract edges, each of cost 1. Since each of these abstract edges can be refined into at most k^ℓ edges at level 0, path t_2' is refined into the path t_2 of no more than $ejk^\ell \cdot k^\ell$ edges, each of which has cost e. Consequently, the total cost of t_2 is $e \cdot ejk^\ell \cdot k^\ell = je^2k^{2\ell}$. Thus, the ratio between the costs of t_1 and t_2 is $e^2k^{2\ell}$. $\square$

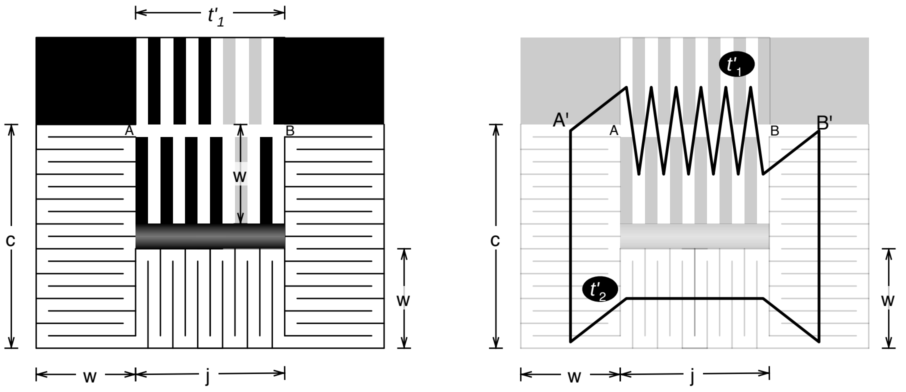

Figure 25: A grid-based example achieving the worst case suboptimality.

The worst-case upper bound is tight, and occurs when we both severely underestimate the cost of a suboptimal path and overestimate the cost of the optimal path. In Figure 25 we show how this can happen in practice; the level-0 map is shown on the left. The lowest-cost path (t_1) between states A and B is a straight line and has the cost j. All corridors have width 1. The length of the corridors, w, is chosen so that at level ℓ of the abstraction, all states in a corridor will abstract together into a single state. In this map there are only cliques of size two (i.e., $k = 2$ in Theorem B.1).

The right part of Figure 25 shows level-ℓ abstract graph in thick lines and the original map in light gray. In this abstraction, the path t'_2 between A' and B' (the abstract parents of A and B) goes through the lower part of the map. The path t'_2 has the abstract cost $2c + j + w$ but its abstract edges refine to w ground-level edges each. Thus, the total cost of its refined path t_2 is $w \times (2c + j) = 2cw + jw$. Path t'_1 is an abstract image of t_1 and has the abstract cost $w = k^\ell$ for each of its j edges, leading to the total abstract cost of $jk^\ell = jw$. It is shown in the right side of the figure as a highly zigzagged path.

We can now choose c so that t'_2 costs just as much as t'_1. Then the agent can have the bad luck of choosing to refine t'_2. To make this a certainty, we can make the cost of t'_1 slightly higher than the cost of t'_2. This is accomplished by setting $2c + j + w \approx jw$. From here, $2c \approx jw - j - w$ and $c \approx jw = jk^\ell = j2^\ell$. As a result, the agent chooses to refine t'_2 into t_2, which has cost $2cw + jw = 2j2^\ell 2^\ell + j2^\ell = O(j2^{2\ell})$. The ratio between this and the cost of t_1 is $2^{2\ell}$, which corresponds to the bound of the theorem for $k = 2, e = 1$.

Our experimental results demonstrate that such large suboptimality does not occur in practice. As an illustration, consider a histogram of suboptimality values for the 3000 problems and all parametrizations of PR LRTS in Figure 26.

If suboptimality does become a practical concern, one can use ideas from HPA* (Botea et al., 2004), where optimal path costs within regions were pre-computed and cached. Such pre-computation will help prevent the severe over- and under-estimation of abstract path costs which was assumed by the worst-case analysis in Theorem B.1.

Appendix C. Maps Used in the Empirical Study

The four additional maps are shown in Figure 27.

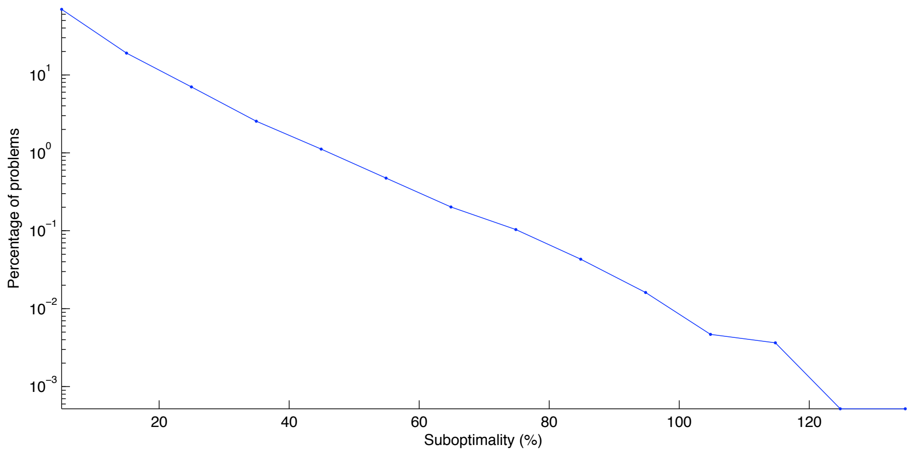

Figure 26: Histogram of suboptimality in our experiments.

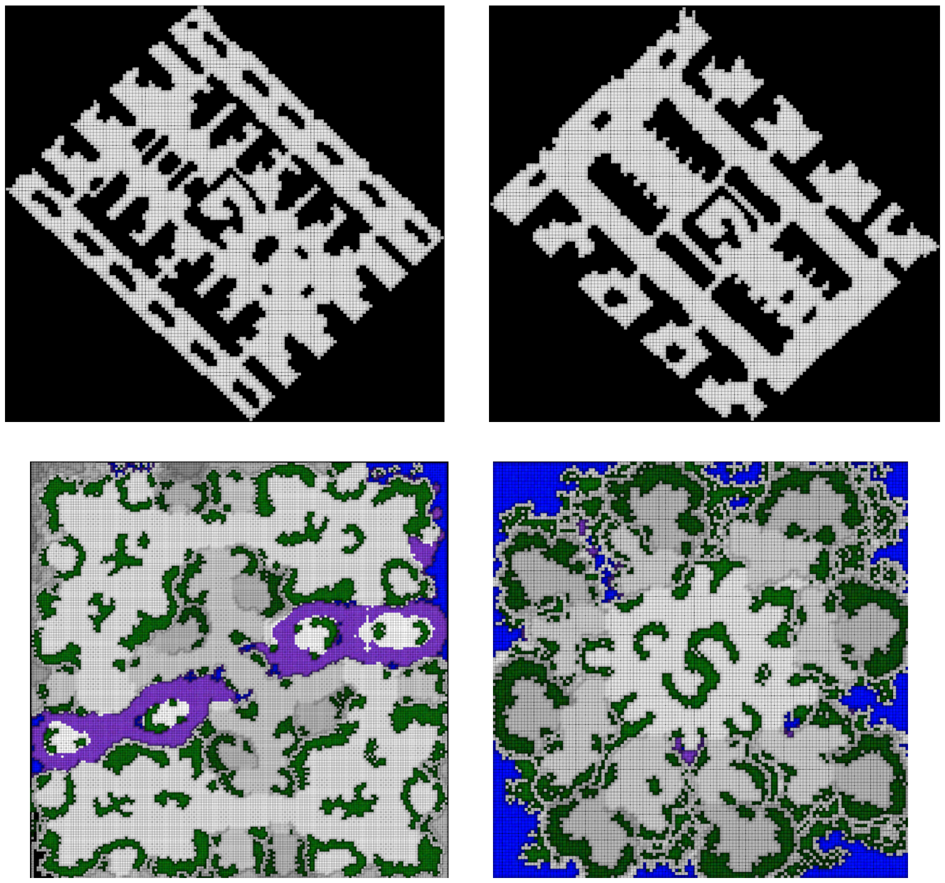

Figure 27: The four additional maps used in the experiments.

Appendix D. Dominance on Average: Plots

Six plots corresponding to entries in Table 2 are shown in Figures 28, 29.

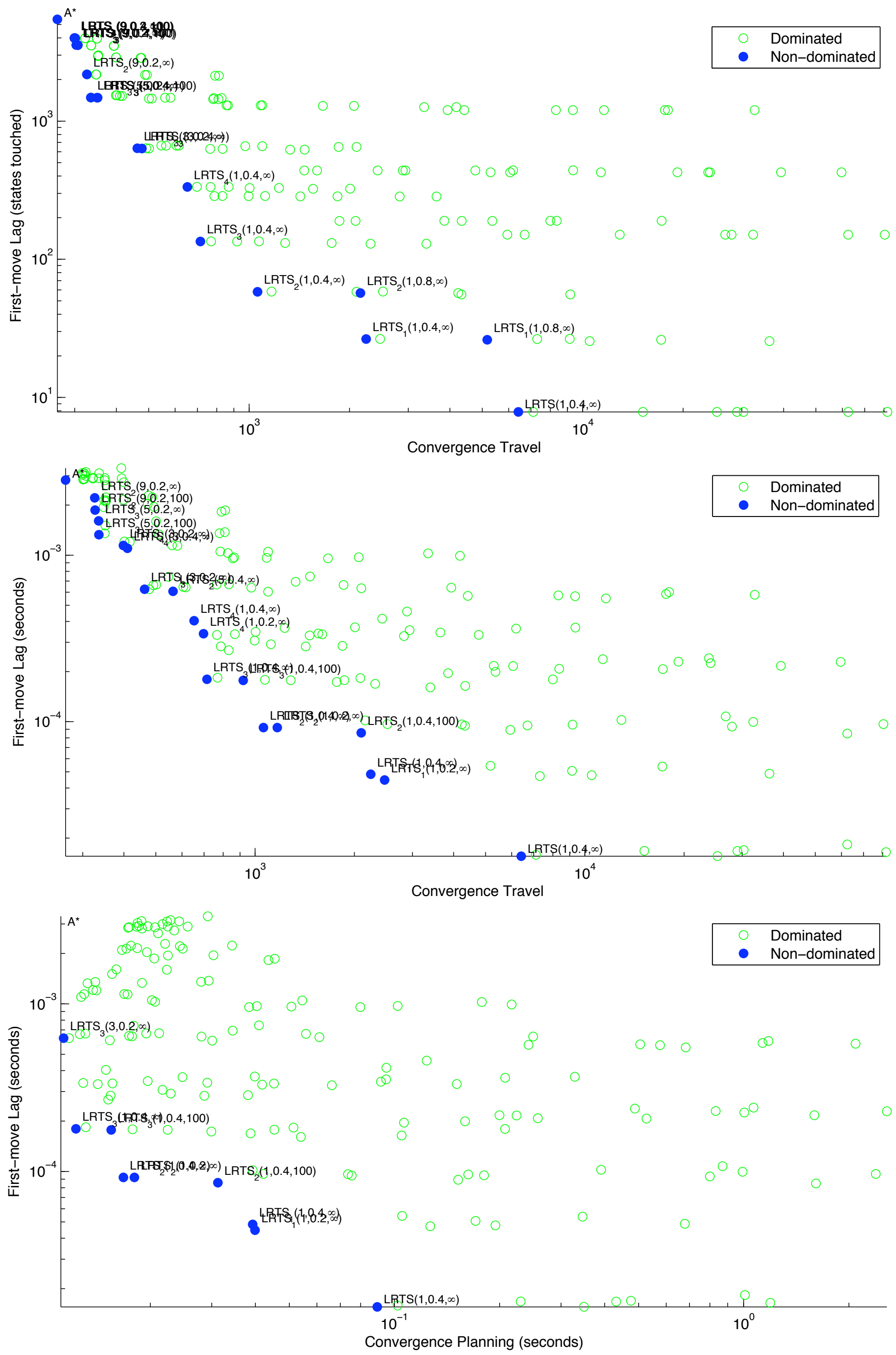

Figure 28: Dominance for several pairs of performance measures. Part 1.

Appendix E. Dominance on Individual Problems

In Section 5.1 we introduced the concept of dominance and demonstrated that PR LRTS with abstraction dominates all but extreme of the search algorithms that do not use abstraction. This analysis was done using cost values *averaged* over the 3000 problems. In this section we consider

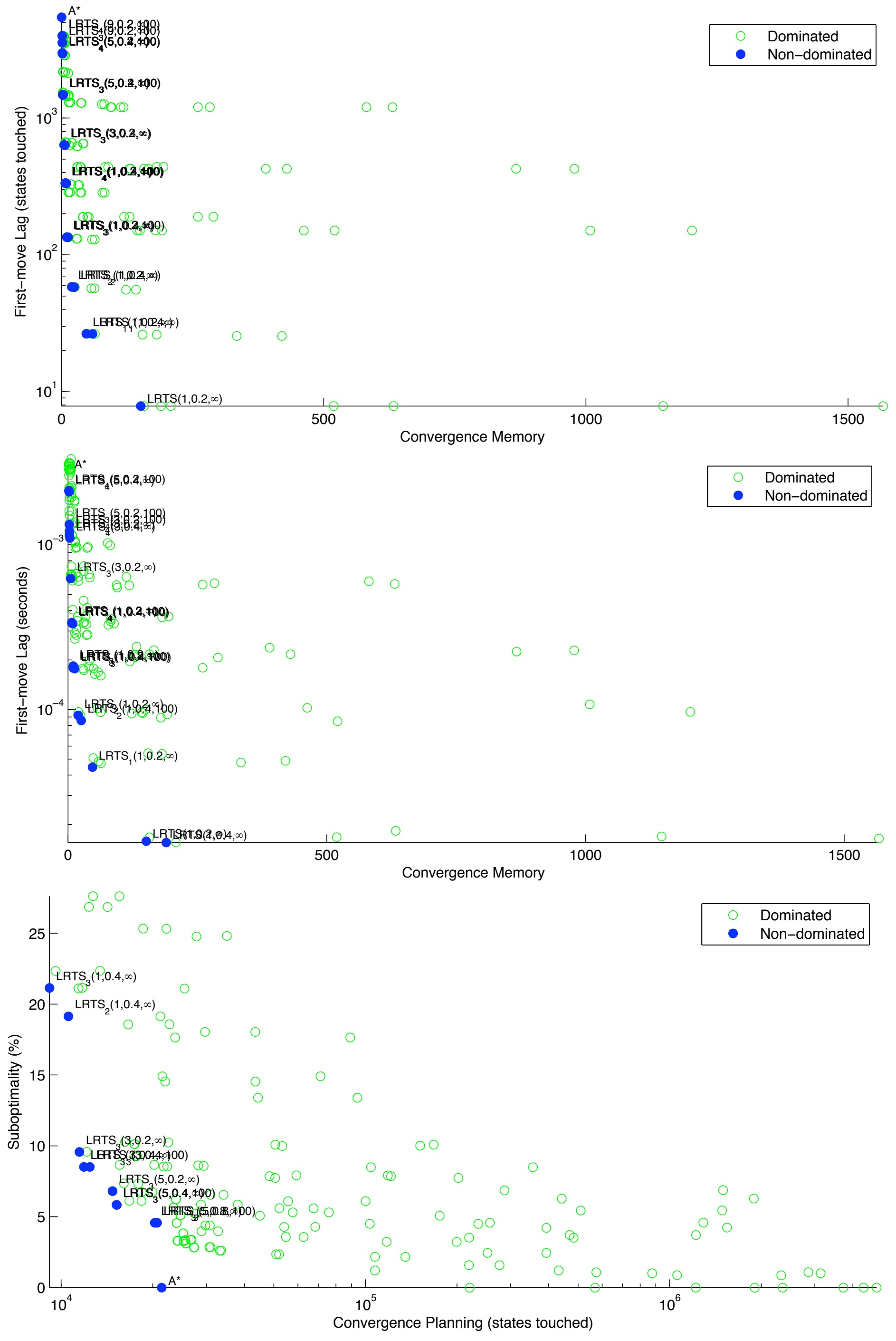

Figure 29: Dominance for several pairs of performance measures. Part 2.

dominance on individual problems. Due to high variance in the problems and their difficulty, we report percentages of problems on which dominance is achieved. For every pair of algorithms, we measure the percentage of problems on which the first algorithm dominates the second. We then measure the percentage of problems on which the second algorithm dominates the first. The ratio between these two percentages we call the *dominance ratio*.

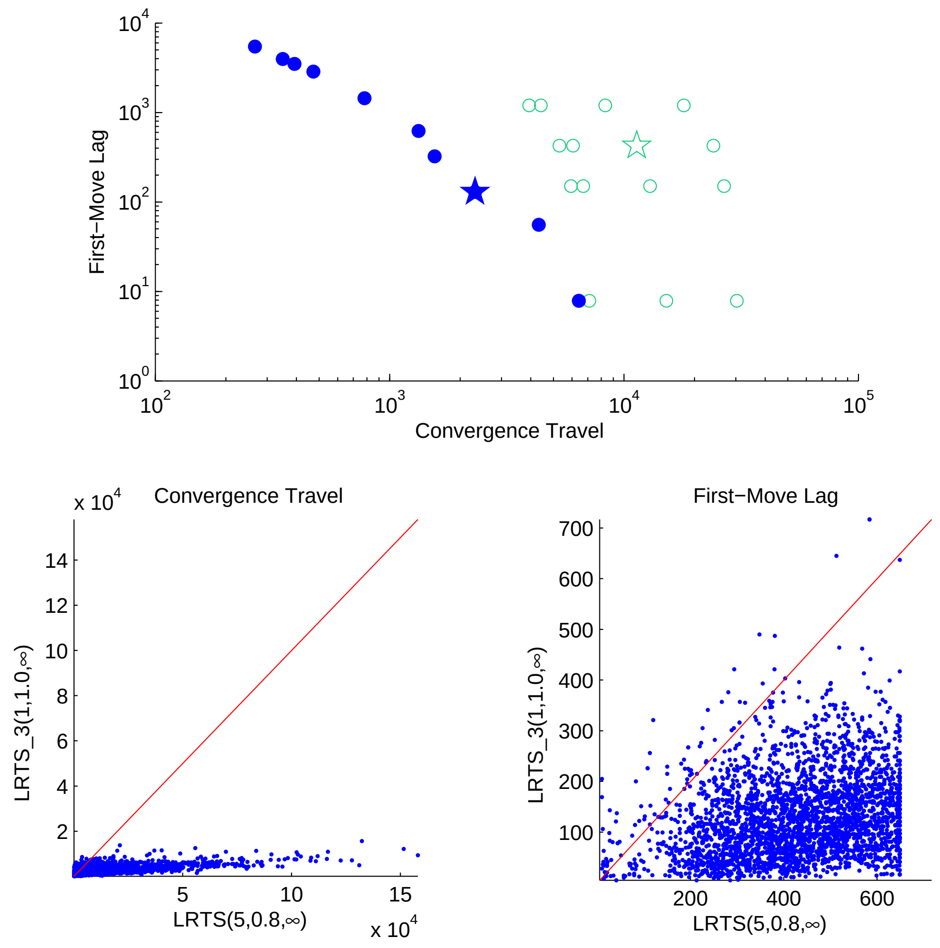

Figure 30: **Top:** $\text{LRTS}_3(1, 1.0, \infty)$ is shown as a filled star; $\text{LRTS}(5, 0.8, \infty)$ is shown as a hollow star. **Bottom left:** convergence travel. **Bottom right:** first-move lag.

Table 4: Statistics for the two algorithms from Figure 30.

Algorithm	Convergence travel	First-move lag	Both	Dominance ratio
LRTS$_3$(1,1.0,∞)	72.83%	97.27%	70.97%	81.89
LRTS(5,0.8,∞)	27.17%	2.67%	0.87%	

At the top of Figure 30 we see a reproduction of the corresponding plot from Figure 28 where two particular algorithms are marked with stars. The filled star is $\text{LRTS}_3(1, 1.0, \infty)$ that uses three levels of abstraction. The hollow star is $\text{LRTS}(5, 0.8, \infty)$ that operates entirely at the ground level. Statistics are reported in Table 4. The bottom left of the figure shows advantages of the PR LRTS with respect to convergence travel. On approximately 73% of the 3000 problems, the PR LRTS travels less than the LRTS before convergence (points below the 45-degree line). With respect to the first-move lag, the PR LRTS is superior to the LRTS on 97% of the problems (bottom right in the figure). Finally, on 71% of the problems the PR LRTS dominates the LRTS (i.e., outperforms it with

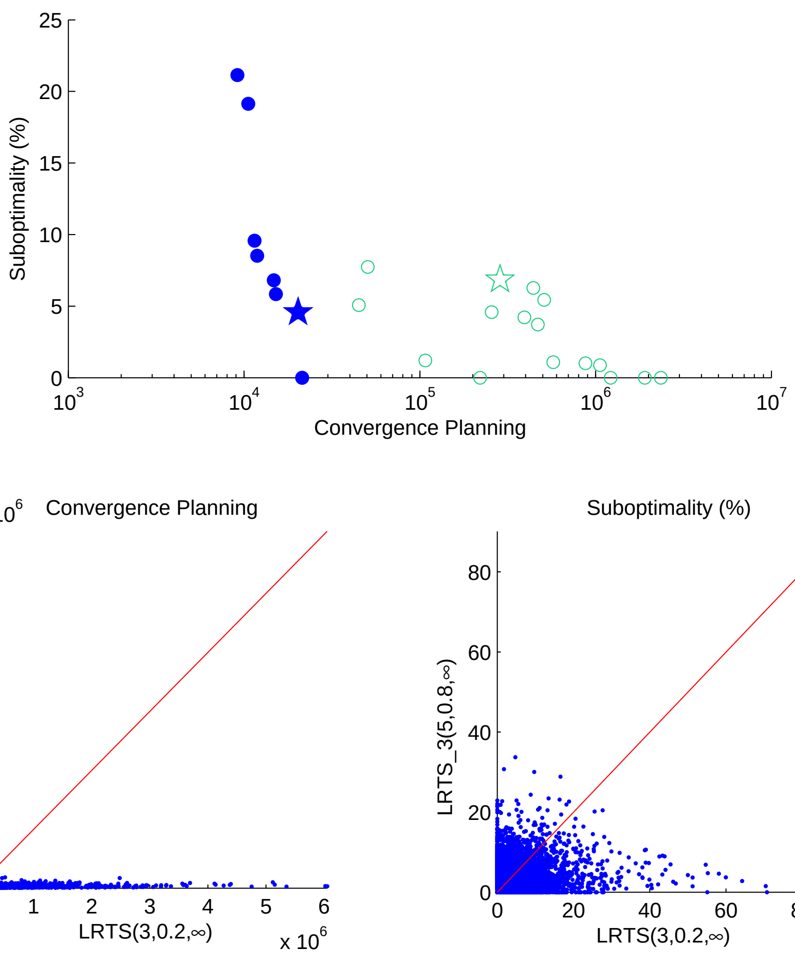

Figure 31: **Top:** $\text{LRTS}_3(5, 0.8, \infty)$ is shown as a filled star; $\text{LRTS}(3, 0.2, \infty)$ is shown as a hollow star. **Bottom left:** convergence planning. **Bottom right:** suboptimality.

respect to *both* measures). On the other hand, the LRTS dominates the PR LRTS on approximately 1% of the problems. This leads to the dominance ratio of 81.89.

Table 5: Statistics for the two algorithms from Figure 31.

Algorithm	Convergence planning	Suboptimality	Both	Dominance ratio
$\text{LRTS}_3(5,0.8,\infty)$	80.03%	55.80%	48.97%	4.01
$\text{LRTS}(3,0.2,\infty)$	19.93%	40.97%	12.2%	

Similarly, Figure 31 compares two algorithms with respect to convergence planning and suboptimality of the final solution. At the top of the figure, we have the corresponding plot from Figure 29 with $\text{LRTS}_3(5, 0.8, \infty)$ shown as a filled star and $\text{LRTS}(3, 0.2, \infty)$ shown as a hollow star. Percent points for domination on individual problems are found in Table 5. The plot at the bottom left of the figure shows that PR LRTS has lower convergence planning cost than the LRTS on 80% of the problems. The plot at the bottom right shows suboptimality of the solutions these algorithms produced. The PR LRTS is more optimal than the LRTS 56% of the time. Finally, the PR LRTS dominates the LRTS on 49% of the problems. Domination the other way (i.e., the LRTS dominates the PR LRTS) happens only on 12.2% of the problems. This leads to the dominance ratio of 4.01.

There are several factors that influence the results. First, there is a high variance in the difficulty of individual problems due to their distribution over five buckets by optimal path distance. Consequently, there is high variance in how the algorithms trade off antagonistic performance measures on these problems. In the case when there is a large difference between mean values, such as in Figure 30, dominance on average is supported by dominance on the majority of individual problems. Conversely, a small difference in mean values (e.g., 2.3% in suboptimality for the algorithms in Figure 31) does not lead to overwhelming dominance at the level of individual problems.

We extended this analysis to all pairs of algorithms displayed in Figures 28, 29. For convergence travel and first-move lag, the dominance ratio varies between 5.48 and ∞ with the values below infinity averaging 534.32 with the standard error of 123.47. For convergence planning and suboptimality, the dominance ratio varies between 0.79 and ∞ with the values below infinity averaging 5.16 with the standard error of 1.51. Finally, only a set of 181 algorithms was tested in our study. Therefore, the results should be viewed as an approximation to the actual dominance relationship among the algorithms.

Appendix F. Interaction Between Abstraction and LRTS Parameters

In Section 5.2 we observed general trends in influence of abstraction on the five performance measures. As the abstraction level adds another dimension to the parameter space of LRTS, previously defined by d, γ, T, the natural question is how the four parameters interact. In order to facilitate a comprehensible visualization in this paper, we will reduce the LRTS parameter space from d, γ, T to d, γ by setting $T = \infty$ (i.e., disabling backtracking in LRTS). This is justified for two reasons. First, recent studies (Bulitko & Lee, 2006; Sigmundarson & Björnsson, 2006) have shown that effects of backtracking are highly domain-specific.

Table 6 gives an overview of the influence of abstraction on the parameters of LRTS at a qualitative level. A more detailed analysis for each of the five performance measures follows. It is important to note that these experiments were performed on a set of fixed cost paths and fixed size maps. Consequently, map boundary effects are observed at higher levels of abstraction. We will detail their contribution below.

Table 6: Influence of LRTS parameters on the impact of abstraction. Each cell in the table represents the impact of abstraction either amplified ("A") or diminished ("D") by increase in d or γ. Lower-case "a" and "d" indicate minor effect, "-" indicates no effect.

measure / control parameter	increase in d	increase in γ
convergence travel	d	A
first-move lag	A	-
convergence planning	a	A
convergence memory	D	A
suboptimality	D	a

Convergence travel: increasing the abstraction level generally decreases convergence travel as LRTS learns on smaller abstract maps. Independently, increasing the lookahead depth in LRTS has a

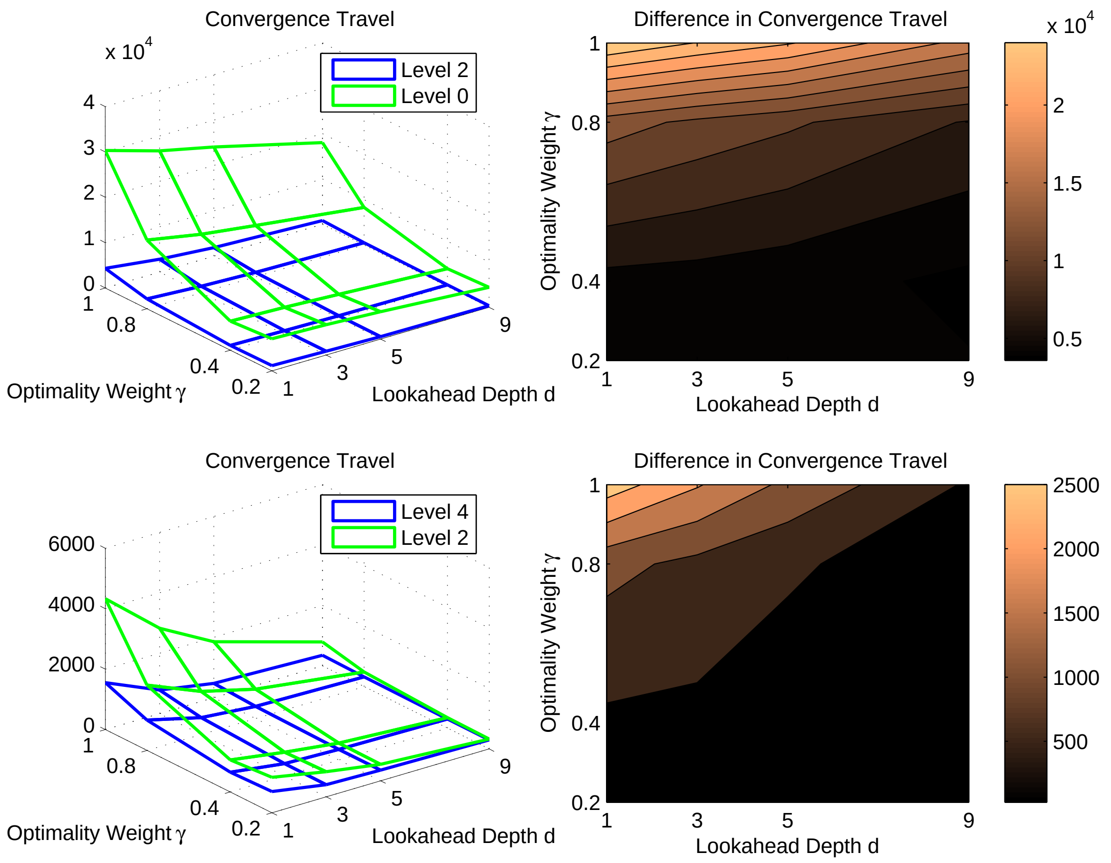

Figure 32: Convergence travel: impact of abstraction as a function of d, γ. Top two graphs: LRTS(d, γ) vs. LRTS$_2(d, \gamma)$. Bottom two graphs: LRTS$_2(d, \gamma)$ vs. LRTS$_4(d, \gamma)$.

similar effect (Bulitko & Lee, 2006). Convergence travel is lower-bounded by the doubled optimal cost from the start to the goal (as the first trial has to reveal parts of the map and, consequently, cannot be final). Therefore, decreasing convergence travel via either of two mechanisms diminishes the gains from the other mechanism. This effect can be seen in Figure 32 where there is a noticeable gap in convergence travel between abstractions levels 0 and 2. But with a lookahead of 9, there is only a small difference between using abstraction levels 2 and 4. Thus, increasing the lookahead slightly diminishes the effect of abstraction (hence the "d" in the table). Increasing γ increases the convergence travel. The higher the value of γ, the more there is to be gained from using abstraction. An increase in γ amplifies the advantage of abstraction.

First-move lag generally increases with both the abstraction level and with lookahead depth. As lookahead depth increases, the size of the corridor used for A* search increases as well. Thus, increasing d amplifies the first-move lag due to abstraction, because PR LRTS must plan once within the lookahead space (within LRTS) and once inside the corridor (within A*) (Figure 33).

Deeper lookahead amplifies the impact of abstraction. In the simplified analysis below, we assume that the map is obstacle free which leads to all levels of abstraction being regular grids (ignoring boundary effects). The length of a path between two points (expressed in the number of actions) is, thus, decreased by a factor of two with each abstraction level. Under these assumptions, the total number of states PR LRTS touches on the first move is $\Omega(d^2)$ at the abstract graph and

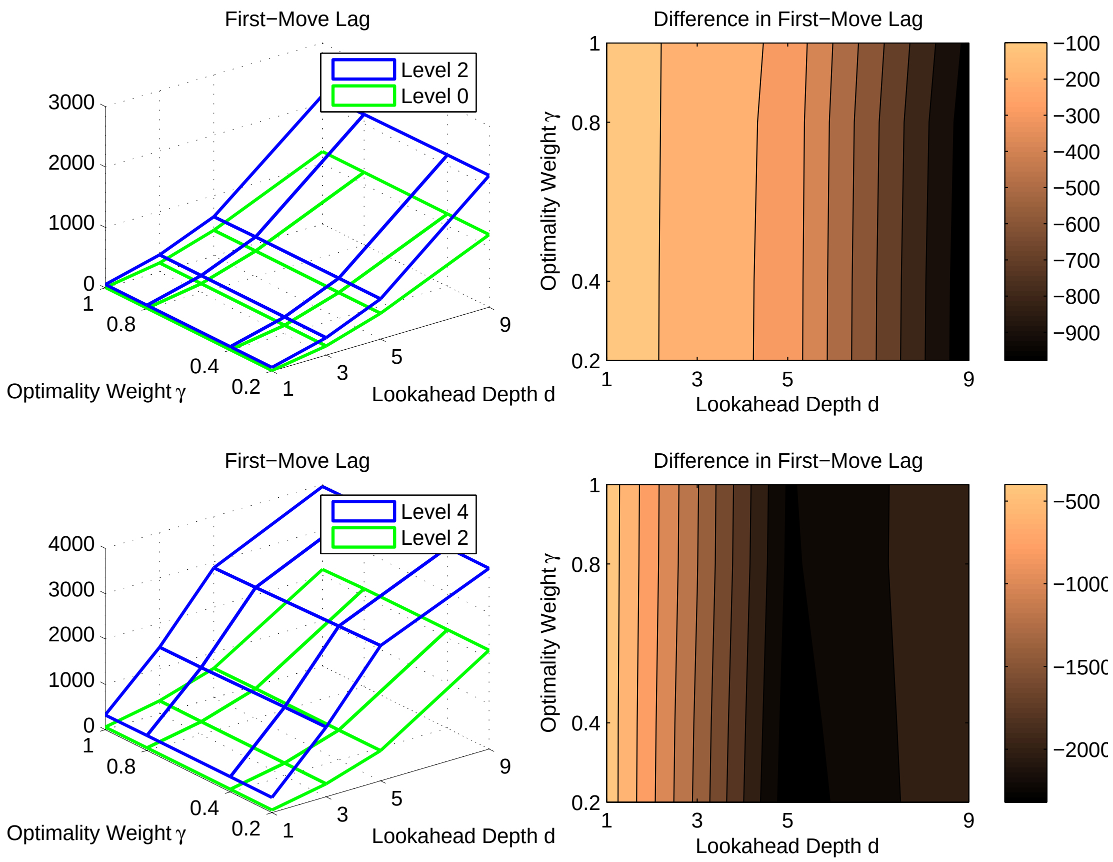

Figure 33: First-move lag: impact of abstraction as a function of d, γ. Top two graphs: LRTS(d, γ) vs. LRTS$_2(d, \gamma)$. Bottom two graphs: LRTS$_2(d, \gamma)$ vs. LRTS$_4(d, \gamma)$.

$\Omega(d \cdot 2^{\ell})$ at the ground graph. The latter quantity is simply the number of edges in the ground path computed as the number of edges in the abstract path (d) multiplied by the reduction factor of 2^{ℓ}. Adding $\jmath$ more abstraction levels increases the first-move lag to $\Omega(d \cdot 2^{\ell+\jmath})$. The increase is a linear function of lookahead depth d. Thus, larger values of d amplify the effect of adding extra abstraction levels.

There are several points glossed over by our simplified analysis. First, the reduction in the path length is not always two-fold as we assumed above. In the presence of walls, higher levels of abstraction are less likely to locate and merge fully-fledged 4-element cliques. Second, boundaries of the abstract graph can be reached by LRTS in less than d moves at the higher abstraction level. This effectively decreases the quantity d in the formula above and the size of the corridor can be reduced from our generous estimate $d \cdot 2^{\ell}$. Finally, “feeding” A* longer abstract path often improves its performance as we have analyzed in a previous section (cf. Figure 22). This explains why at abstraction level 4 deepening lookahead has diminishing returns as seen in Figure 33.

Optimality weight does not affect the number of states touched by LRTS at the abstract level. On the other hand, it can change the cost of the resulting A* search as a different abstract path may be computed by LRTS. Overall, however, the effect of γ on the first-move lag and the impact of abstraction is inconsequential (Figure 33).

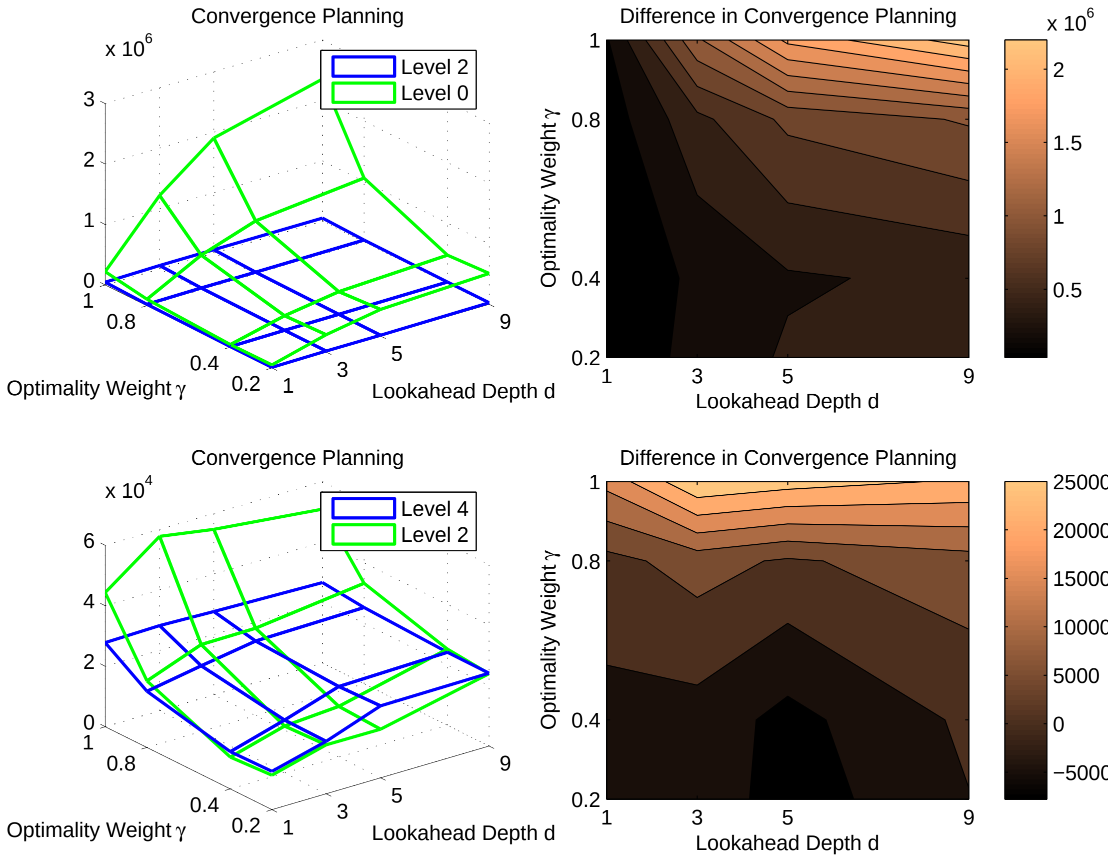

Figure 34: Convergence planning: impact of abstraction as a function of d, γ. Top two graphs: LRTS(d, γ) vs. LRTS$_2(d, \gamma)$. Bottom two graphs: LRTS$_2(d, \gamma)$ vs. LRTS$_4(d, \gamma)$.

Convergence planning: As the abstraction level increases, convergence planning generally decreases. The effect of d is more complex, because deeper lookahead increases the cost of each individual planning step, but overall decreases planning costs as convergence is faster. The interplay of these two trends moderates the overall influence as seen in Figure 34.

The effect of γ on convergence planning is non-trivial. In general, lower values of γ reduce the convergence planning cost. Note that convergence planning cost is a product of average planning time per unit of distance and the convergence travel. As we discussed above, optimality weight amplifies the effects of abstraction convergence travel. At the same time, it does not substantially affect increase in planning per move as the abstraction goes up. Combining these two influences, we conclude that optimality weight will amplify effects of abstraction on convergence planning. This is confirmed empirically in Figure 34.

Convergence memory: Abstraction decreases the amount of memory used at convergence because there are fewer states over which to learn. The effects of d and γ are the same as for convergence travel described above. This is because there is a strong correlation between convergence travel and convergence memory that we have previously discussed. Visually Figures 32 and 35 display very similar trends.

Suboptimality: Increasing the abstraction level increases suboptimality. For plain LRTS, lookahead depth has no effect on suboptimality of the final solution. However, when we combine deeper

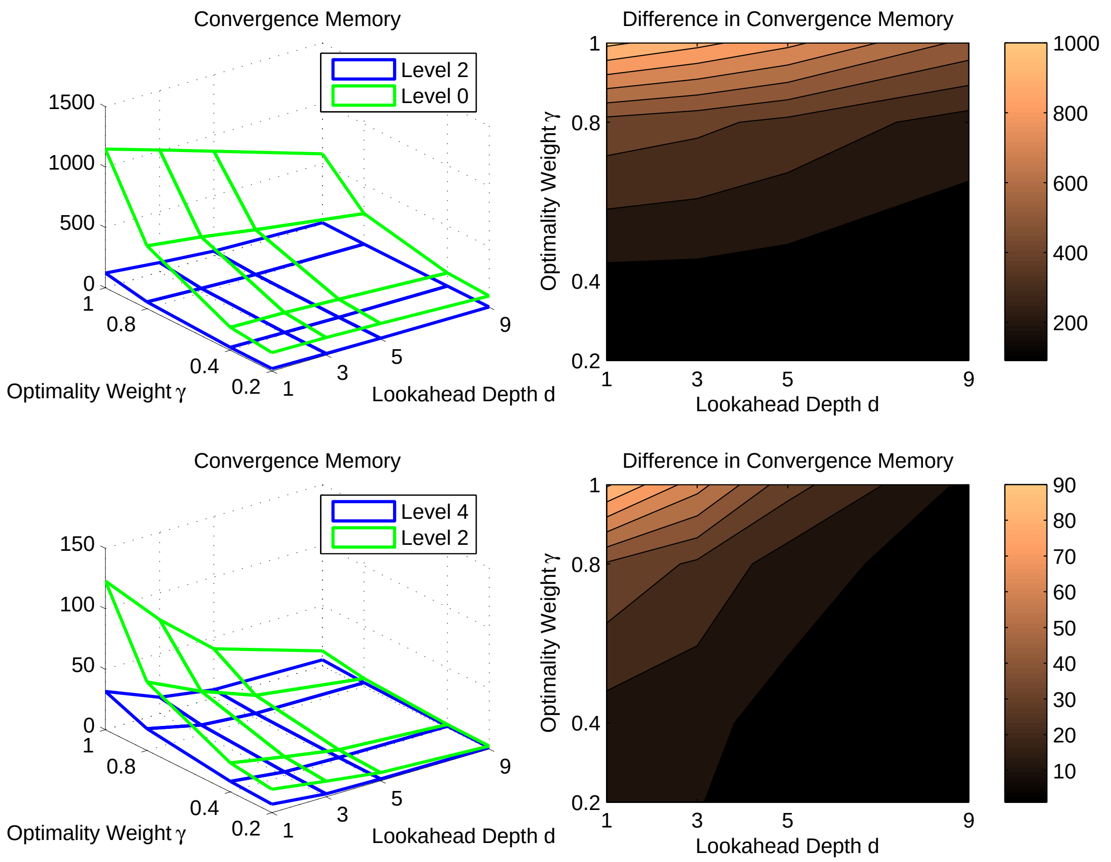

Figure 35: Convergence memory: impact of abstraction as a function of d, γ. Top two graphs: LRTS(d, γ) vs. LRTS$_2(d, \gamma)$. Bottom two graphs: LRTS$_2(d, \gamma)$ vs. LRTS$_4(d, \gamma)$.

lookahead with abstraction the suboptimality arising from abstraction decreases. With deeper lookahead the abstract goal state is seen earlier making PR LRTS a corridor-constrained A*. Additionally, as we discussed in Section 5.2 and Figure 22, refining shorter paths (computed by LRTS with lower d) introduces additional suboptimality. As suboptimality is lower bounded by 0%, increasing lookahead diminishes the effects of abstraction on suboptimality (Figure 36) – hence "D" in Table 6.

Increasing γ decreases the amount of suboptimality when no abstraction is used. When combined with abstraction increasing γ has a minor amplification effect on the difference abstraction makes (Figure 36) for two reasons. First, at abstract levels the graphs are fairly small and γ makes less difference there. Second, the degree suboptimality of an abstract path does not translate directly into the degree of suboptimality of the resulting ground path as the A* may still find a reasonable ground path. Thus, the influence of γ at the abstract level is overshadowed by the suboptimaly introduced by the process of refinement itself (cf. Figure 21).

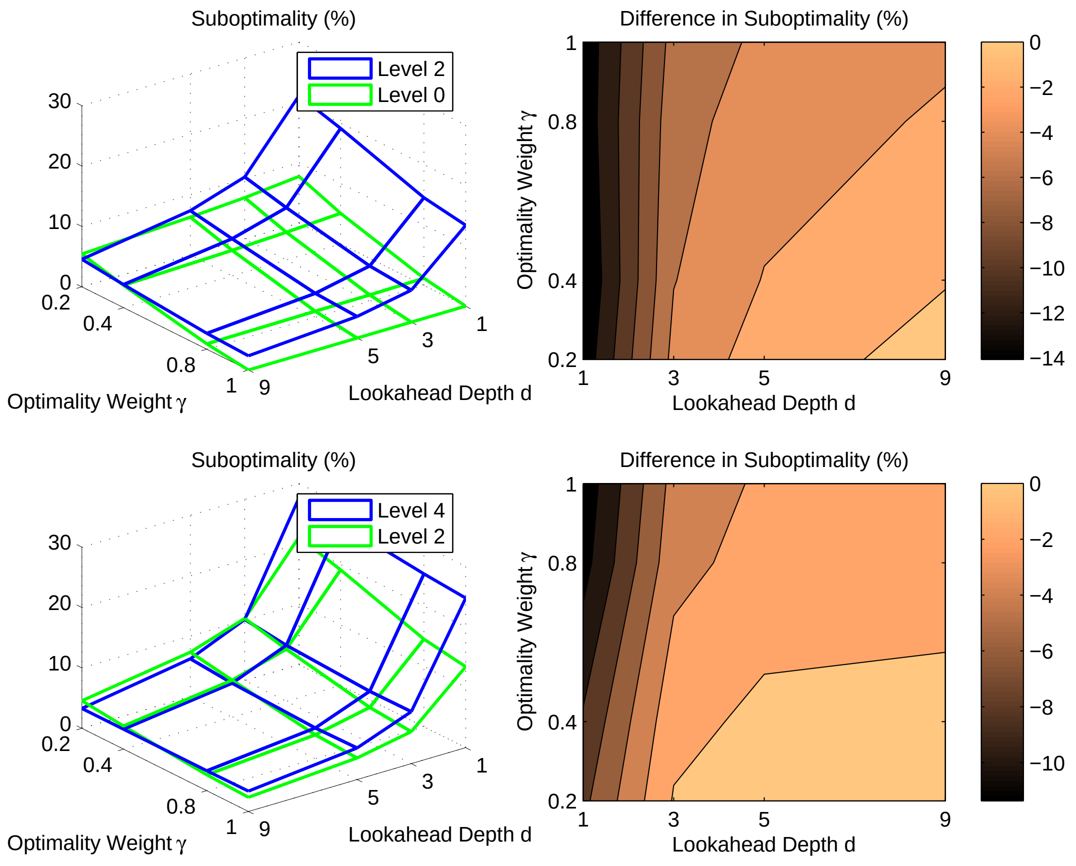

Figure 36: Suboptimality: impact of abstraction as a function of d, γ. Top two graphs: LRTS(d, γ) vs. LRTS$_2(d, \gamma)$. Bottom two graphs: LRTS$_2(d, \gamma)$ vs. LRTS$_4(d, \gamma)$.

References

Aylett, R., & Luck, M. (2000). Applying artificial intelligence to virtual reality: Intelligent virtual environments. *Applied Artificial Intelligence*, *14*(1), 3–32.

Bacchus, F., & Yang, Q. (1994). Downward refinement and the efficiency of hierarchical problem solving.. *Artificial Intelligence*, *71(1)*, 43–101.

BioWare Corp. (1998). Baldur's Gate., Published by Interplay, http://www.bioware.com/bgate/, November 30, 1998.

Blizzard Entertainment (2002). Warcraft III: Reign of Chaos., Published by Blizzard Entertainment, http://www.blizzard.com/war3, July 3, 2002.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near Optimal Hierarchical Path-Finding. *Journal of Game Development*, *1(1)*, 7–28.

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

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

Bulitko, V., Sturtevant, N., & Kazakevich, M. (2005). Speeding up learning in real-time search via automatic state abstraction. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1349 – 1354, Pittsburgh, Pennsylvania.

Culberson, J., & Schaeffer, J. (1996). Searching with pattern databases. In *CSCI (Canadian AI Conference)*, Advances in Artificial Intelligence, pp. 402–416. Springer-Verlag.

Dini, D. M., van Lent, M., Carpenter, P., & Iyer, K. (2006). Building robust planning and execution systems for virtual worlds. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment conference (AIIDE)*, pp. 29–35, Marina del Rey, California.

Ensemble Studios (1999). Age of Empires II: Age of Kings., Published by Microsoft Game Studios, http://www.microsoft.com/games/age2, June 30, 1999.

Felner, A., Zahavi, U., Schaeffer, J., & Holte, R. (2005). Dual lookups in pattern databases. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 103–108, Edinburgh, United Kingdom.

Furcy, D. (2006). ITSA*: Iterative tunneling search with A*. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, Boston, Massachusetts.

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

Furcy, D., & Koenig, S. (2005). Limited discrepancy beam search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 125–131.

Garey, M. R., & Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Hansen, E. A., & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, *28*, 267–297.

Hansen, E. A., Zilberstein, S., & Danilchenko, V. A. (1997). Anytime heuristic search: First results. Tech. rep. CMPSCI 97-50, Computer Science Department, University of Massachusetts.

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

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

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

Holte, R., Mkadmi, T., Zimmer, R. M., & MacDonald, A. J. (1996). Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, *85*(1-2), 321–361.

Holte, R., Perez, M., Zimmer, R., & MacDonald, A. (1995). Hierarchical A*: Searching abstraction hierarchies efficiently. Tech. rep. tr-95-18, University of Ottawa.

id Software (1993). Doom., Published by id Software, http://en.wikipedia.org/wiki/Doom, December 10, 1993.

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

Ishida, T., & Korf, R. (1991). Moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 204–210.

Kitano, H., Tadokoro, S., Noda, I., Matsubara, H., Takahashi, T., Shinjou, A., & Shimada, S. (1999). Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Proceedings of the IEEE Conference on Man, Systems, and Cybernetics*, Vol. 4, pp. 739–743.

Koenig, S. (1999). Exploring unknown environments with real-time search or reinforcement learning. In *Proceedings of the Neural Information Processing Systems*, pp. 1003–1009.

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

Koenig, S., & Likhachev, M. (2002a). D* Lite. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 476–483.

Koenig, S., & Likhachev, M. (2002b). Incremental A*. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1539–1546.

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

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

Koenig, S., & Simmons, R. (1998). Solving robot navigation problems with initial pose uncertainty using real-time heuristic search. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pp. 144 – 153.

Koenig, S., Tovey, C., & Smirnov, Y. (2003). Performance bounds for planning in unknown terrain. *Artificial Intelligence*, *147*, 253–279.

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

Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., & Thrun, S. (2005). Anytime dynamic A*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Likhachev, M., Gordon, G. J., & Thrun, S. (2004). Ara*: Anytime a* with provable bounds on sub-optimality. In Thrun, S., Saul, L., & Schölkopf, B. (Eds.), *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA.

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*, pp. 108–114, Boston, Massachusetts.

Mero, L. (1984). A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, *23*, 13–27.

Orkin, J. (2006). 3 states & a plan: The AI of F.E.A.R. In *Proceedings of the Game Developers Conference (GDC)*. http://www.jorkin.com/gdc2006_orkin_jeff_fear.doc.

Pearl, J. (1984). *Heuristics*. Addison-Wesley.

Pohl, I. (1970). First results on the effect of error in heuristic search. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence*, Vol. 5, pp. 219–236. American Elsevier, New York.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computaional issues in heuristic problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 12–17.

Pottinger, D. C. (2000). Terrain analysis in realtime strategy games. In *Proceedings of Computer Game Developers Conference*. www.gamasutra.com/features/gdcarchive/2000/pottinger.doc.

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

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 25–34, New York, NY, USA. ACM Press.

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

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

Shue, L.-Y., Li, S.-T., & Zamani, R. (2001). An intelligent heuristic algorithm for project scheduling problems. In *Proceedings of the 32nd Annual Meeting of the Decision Sciences Institute*.

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

Sigmundarson, S., & Björnsson, Y. (2006). 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.

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

Stout, B. (1996). Smart moves: Intelligent pathfinding. *Game Developer Magazine*, *October*, 28–35.

Sturtevant, N. (2005). HOG - Hierarchical Open Graph. http://www.cs.ualberta.ca/˜nathanst/hog/.

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

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

Sturtevant, N., & Jansen, R. (2007). An analysis of map-based abstraction and refinement. In *Proceedings of the 7th International Symposium on Abstraction, Reformulation and Approximation*, Whistler, British Columbia. (in press).

Sutton, R. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 216–224. Morgan Kaufmann.

Thalmann, D., Noser, H., & Huang, Z. (1997). Autonomous virtual actors based on virtual sensors. In *Lecture Notes in Computer Science (LNCS), Creating Personalities for Synthetic Actors, Towards Autonomous Personality Agents*, Vol. 1195, pp. 25–42. Springer-Verlag, London.

Yahja, A., Stentz, A. T., Singh, S., & Brummit, B. (1998). Framed-quadtree path planning for mobile robots operating in sparse environments. In *In Proceedings, IEEE Conference on Robotics and Automation, (ICRA)*, Leuven, Belgium.