2011

Salient Search

Jonathan Vermette
University of Windsor

SALIENT SEARCH

by

JONATHAN VERMETTE

A Thesis
Submitted to the Faculty of Graduate Studies
through Computer Science
in Partial Fulfilment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2011

SALIENT SEARCH

by

JONATHAN VERMETTE

APPROVED BY:

Dr. Phil Graniero
Department of Earth and Environmental Sciences

Dr. Dan Wu
School of Computer Science

Dr. Scott Goodwin
School of Computer Science

Dr. Subir Bandyopadhyay
Chair of Defense
School of Computer Science

May 19, 2011

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

All real-time pathfinding algorithms suffer from some degree of suboptimal behaviour on the part of the agent. A consequence of the need to perform a move before it is guaranteed to be optimal, this is inversely proportional to the amount of effort given to planning between each move.

Many real-time algorithms employ a constant-bounded local search to plan a single move at a time. However they need multiple trials to converge on an optimal solution. More recent hierarchical approaches produce good results after a single trial, but rely on extensive pre-processing, limiting their use in dynamic environments. A newer algorithm, Time Bounded A*, conducts a global A* to find an optimal path on the first trial, while creating partial paths for an agent to follow. However, harder search problems can induce the appearance of indecisiveness on the part of the agent as all of its time is spent moving back and forth between subgoals.

To remedy this, we introduce Salient Search. This algorithm adds new features to TBA* to track and dedicate search effort to a given subsection of the open list. Experiments on maps taken from popular computer games show that for small planning slices, Salient Search reduces the indecisiveness shown by an agent. Further, the effect is stronger on more difficult problems.

Dedication

To Jennifer...

Acknowledgements

I would first like to thank my supervisor, Dr. Scott Goodwin for inviting me to work in Game AI, and providing plenty of guidance and direction during my time as a graduate student. He has helped me win an argument that has gone on between parents and kids all over: All those years playing video games, instead of being a waste of my time have put me on the path to a satisfying and challenging career.

I would like to the thank the members of the thesis committee for taking time out of their schedules to review this thesis, and for providing valuable feedback during the proposal. Their suggestions guided this work.

Special thanks goes to Dr. Nathan Sturtevant, previously of the University of Alberta, for replying to my early enquiries into his work and for providing access to the code repository for his Hierarchical Open Graph project. The dataset of maps used in this thesis were derived from the set provided through this framework.

Finally, I would like to thank the Montana's restaurant of Windsor, Ontario for providing table coverings that patrons are encouraged to draw on. It was here while having lunch one afternoon that Dr. Goodwin and I sketched out the initial idea that became Salient Search. And sketch is the key word. From this meeting I suspect that this thesis has the curious distinction of being among the first to be conceived in crayon and butcher's paper.

I would thank my peers as well as the reader for not drawing undue conclusions of the inherent quality of a crayon-based thesis.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Glossary

A*

An informed search algorithm. A best-first algorithm, notable for being *optimally efficient*; A* expands the fewest number of nodes while guaranteeing the path found is optimal.

backtracking

A move made by an agent in real-time pathfinding to return to a previously visited node, in pursuit of a new subgoal.

closed list

A data structure used in pathfinding algorithms to identify search nodes that have already been expanded.

heuristic

A function to estimate the optimal distance between two nodes or states in a problem space. Heuristic functions that never overestimate the optimal distance are said to be *admissible*, and important property.

heuristic error

The difference between a heuristic estimate and the true optimal cost between two nodes or states in a problem space.

learning real-time search

Search algorithms that repeatedly revisit values to update them and converge

to their 'true' value. LRTA* is a notable example.

most promising node

The next node to be expanded in best-first search. In A*, the most promising node is the node on the open list with the lowest f-score. In TBA*, the most promising node at different points in time form the endpoints of the various paths created.

octile grid

A two-dimensional grid space where diagonal moves are permitted.

optimal path

Between two points, the optimal path is that which has the lowest cost to traverse.

open list

A data structure used in pathfinding algorithms to identify search nodes that are ready to be expanded.

pathfinding

The problem of identifying a series of moves or transformations in a problem space that produces a desired state or solution.

real-time pathfinding

Any pathfinding algorithm that incorporates both planning and execution by an agent, while guaranteeing a constant bound on the amount of planning for every execution, independent of the size of the problem space.

resource limit

A limit defining the bound on planning in a real-time algorithm.

salient

The subtree of the explored search space that are successors of the *salient root*.

salient expansion

Node expansion conducted on nodes defined by the salient list.

salient list

The data structure that describes which nodes on the open list are part of the salient.

salient search

A real-time pathfinding algorithm derived from TBA*.

salient root

The node that initially describes a salient. When expanded, all successor nodes are referenced by the salient list.

strategy

A decision function to choose between the best nodes of the open and salient lists.

subgoal

Some intermediate destination node that the agent is moving towards, and not necessarily the stated goal of the problem

suboptimality

The ratio to which a path between two points is greater in cost than the optimal.

time bounded A*

A real-time pathfinding algorithm. Notable for employing a global search technique while ensuring a constant bound on planning.

travel ratio

The ratio between the cost of the solution path, and the total cost incurred by the agent reaching the goal.

Chapter 1

Introduction

1.1 Problem Domain

This is a thesis on *pathfinding*. Broadly speaking, pathfinding is the problem of determining a series of moves or transformations (the *path*) that results in a solution state. Examples include moving through an environment in a computer game to some desired destination, a series of commands to manipulate a robotic arm to a given configuration, or the fewest moves to a checkmate of an opponent in chess. Pathfinding algorithms can be applied to any problem that can be expressed in terms of a weighted, directed graph where each possible state of a problem is expressed in terms of a *node*. In general, a pathfinding problem can be expressed by the tuple $\{\mathcal{P}, \mathcal{S}, \mathcal{G}, \mathcal{E}, h\}$, where:

- $\mathcal{P}$ is the *problem space* or *domain*. The complete set of possible states, or nodes, of the problem.
- $\mathcal{S}$ is the initial or *start* state or node.
- $\mathcal{G}$ is the *goal* node(s).
- $\mathcal{E}$ is the *succession operator*, which gives a set of valid transitions and associated costs in $\mathcal{P}$.

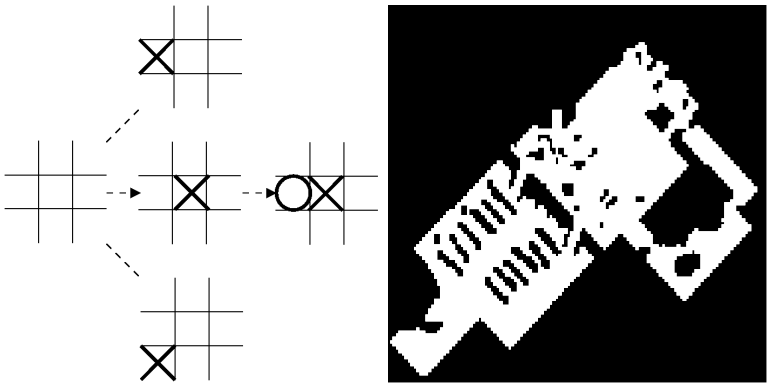


Figure 1.1: Game trees and Virtual Spaces, two types of problem domains where pathfinding applies.

- h is a *heuristic function.*

Two examples of problem domains are illustrated in Figure 1.1. On the left is a portion of a game tree for Tic-Tac-Toe. From the initial empty board at the beginning of the game, a series of moves are possible alternating the placement of X's and O's on the board, creating successive configurations. Each possible configuration of X's and O's comprises a single state in this game tree. A goal state would be any state with three X's or 3 O's in a row, depending on which is desired. If the symmetrical configurations[†] are collapsed into a single state, then the game tree for Tic-Tac-Toe is actually quite small [Dew89]. Other examples of games that are readily described by game trees include Chess, Checkers, Go, or puzzles like the 8-puzzle or a Rubik's Cube.

On the right is an example of a virtual space: a grid-based map taken from a computer game [CSE00] that is used later in this thesis. In a virtual space such as this, states are each (x, y) coordinate cell, and edges defined for each move between adjacent cells.

This thesis works in a subset of pathfinding, known as real-time pathfinding. Real-time pathfinding algorithms incorporate an agent performing transformations

[†]For example, the topmost state represents four different legal moves in the game, if one rotates the board to see the other moves.

alongside the search for a solution. Also, by the definition in [BSLY07], a *real-time* pathfinding algorithm is one that can perform a constant amount of work between a given time interval, regardless of the size of the problem space. More detail on the definition is in [Koe01].

[BSLY07] notes that the real-time pathfinding algorithms remain important in robotics (citing [KS98, Koe98, KTN+99, KTS03]) and video games. Robotics systems are known to use variants of the D* algorithm [Ste94], including D*-Lite and AD* [KL02, LFG+05, FHL08], especially autonomous vehicle navigation.

1.2 Contribution of this Thesis

In this thesis we introduce a variant of real-time heuristic search dubbed Salient Search. This algorithm builds on the Time Bounded A* (TBA*) algorithm [BBS09]. Salient Search allows for targeted expansions of a chosen subset of the open list through the use of a secondary data structure, while preserving the constant time guarantees that define real-time pathfinding. Also used is a secondary function called a *strategy* to influence intermediate goal selection and weighting of the open list. In addition to describing Salient Search, this thesis provides the results of an empirical analysis of Salient Search versus TBA*.

All real-time pathfinding algorithms suffer from some degree of suboptimal behaviour on the part of the agent. A consequence of the need to perform a move before it is guaranteed to be optimal, this is inversely proportional to the amount of effort given to planning between each move. In certain applications like large scale crowd simulation or real-time strategy computer games, computing time devoted to pathfinding must be further divided up amongst many agents, each with their own separate search problems. As a result, the limited computational budget magnifies the scale of this suboptimal behaviour.

For TBA*, multiple paths are created over the lifetime of a search. With more

difficult search and/or small planning windows, the paths can be wildly divergent, sharing no common portion beyond their starting location. In this circumstance, an agent would make little progress before requiring a return to the starting location to continue forward. This indecisiveness or 'squirreliness' on the part of the agent is undesirable in the context of computer games. A user witnessing such behaviour may issue an additional search command thinking something is wrong, effectively throwing away any progress made as search begins anew.

The algorithm proposed in this thesis, Salient Search, performs better than TBA* in avoiding this behaviour for small planning windows. The experiments presented in Chapter 4 show that the features introduced in Salient Search cause an agent to change direction fewer times than they would using TBA*. Further, the effect is stronger with more difficult problems.

1.3 Organization of this Thesis

The rest of this thesis is organised as follows: in Chapter 2 a brief overview of real-time pathfinding is provided. Several milestone techniques in the field are described, showing the evolution of real-time pathfinding. An in-depth look at TBA* finishes the review of past techniques. TBA* is a recent algorithm from which Salient Search is derived. Chapter 3 introduces the Salient Search algorithm itself, which forms the main contribution of this thesis. The two distinguishing features of the algorithm are the *Salient List* and *Salient Strategy*. Chapter 4 presents an empirical study of Salient Search. This study compares the behaviour of Salient Search against its parent algorithm, TBA*. Chapter 5 provides some concluding remarks, as well as observations on possible future work with Salient Search.

Ending this thesis are appendices with additional information. Appendix A contains tables of additional data not directly discussed in the body of the thesis. Appendix B is an index of all the maps that were used to conduct the empirical study.

Chapter 2

Real-time Pathfinding

2.1 A*

The most widely known best-first search algorithm is the A* algorithm, introduced by Hart in [HNR68].

Algorithm 1 A*

```
1: add Start to OPEN
2: goalFound ← false
3: while OPEN ≠ empty AND Goal not found do
4:     next ← OPEN with lowest f
5:     neighbours ← expand next
6:     add next to CLOSED
7:     for all neighbour in neighbours do
8:         if CLOSED contains neighbour then
9:             continue
10:        else if OPEN contains neighbour then
11:            update neighbour
12:        else
13:            add neighbour to OPEN
14:        end if
15:    end for
16: end while
```

Algorithm 1 is the pseudocode for A*. At the beginning of A* search, the closed list is empty, and the only node on the open list is the starting node. This first node is

expanded - its neighbors have their f-scores calculated and are placed onto the open list (lines 7-15).

The next node from the open list is the one with the best (e.g. lowest) f-score, and similarly processed (lines 4-6). This process is repeated until either the goal is reached or the open list is first exhausted (line 3).

When the goal is found to be reachable from the start, a path is traced backwards from the goal. This is accomplished by having each node maintaining a parent reference to the node that produced it. In this way the open and closed lists form a tree data structure, with the open list forming the leaf nodes, and the starting location as the root.

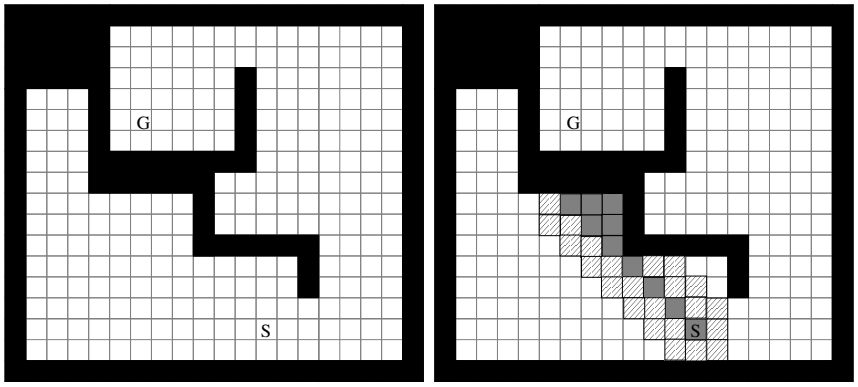


Figure 2.1: A snapshot of A* in action finding a solution from S to G. A* uses the open list to process or 'expand' nodes in a best-first fashion, adding to or updating neighbouring nodes on the open list where necessary. Processed nodes are put onto the closed list. Here, closed list nodes are shaded gray, surrounded by open list nodes (the hatched nodes) to be processed.

Calculating $f(n)$

As mentioned, determining the ordering of expansion is determined by calculating a score for each node n. In A* this equation takes the form of $f(n)$:

$$f(n) = g(n) + h(n)$$

$g(n)$ is the *traversal cost*, exactly how much it costs to transition to n from the starting state by adding up all costs in between. This is determined by $g(n)$ for the parent state plus the transition cost to this state from the parent state. For the starting node $g(n) = 0$.

$h(n)$ is the *heuristic function*. This function is an informed 'guess' as to how much it will cost to transition to the goal state from n. In other words, an estimate of $g(goal) - g(n)$. Heuristics are usually described in terms of the transition costs. For example, for a Rubik's cube where each state describes a configuration of the faces, the cost and transitions can be the number of and type of planar rotations of the toy. A (not efficient) example of a heuristic for a Rubik's cube would be a count of the number of faces solved; positioned and oriented as they would be in the finished cube. In a computer game, it may be the Euclidean[†] distance between two coordinates. In chess, it could be the number of legal moves between the current and desired layout of pieces on the board.

2.2 Learning Real-Time Algorithms

2.2.1 LRTA*

Richard Korf noted the exponential running time to find a solution as a serious drawback of A*. Also problematic was the need to wait for the complete solution to be found by the algorithm before an agent could make any move. Noting that solutions that can be found quickly but are not necessarily optimal (such an algorithm is often both satisfactory and sufficient, or *satisficing* according to Herbert Simon [Sim96]), Korf addressed these limitations in [Kor90] by introducing one of the first real-time search algorithms, Learning Real-Time A* (LRTA*).

In LRTA*, every iteration from the beginning involves a search from the agent's

[†]Straight line.

Algorithm 2 LRTA*

1: **while** $loc \neq goal$ **do**
2: breadth-first search to depth d from loc
3: identify S' with lowest f-score from loc
4: $h(loc, goal) \leftarrow \max(h(loc, goal), g(loc, S') + h(S', goal))$
5: *move* towards S'
6: **end while**

current location out to all nodes up to d states away.

It is in setting a maximal depth bound that LRTA* claims real-time performance. Although breadth-first search is known for a branching factor b to be of complexity $O(b^d)$, fixing the depth places a constant upper bound on the number of expansions for a given branching factor. The search tree produced in iteration is not carried over to the next iteration. By not retaining the tree, the amount of time to perform a certain number of expansions does not change across iterations. Adjusting the value of d however is still subject to the expected combinatorial explosion.

During expansion, every node is evaluated with the same scoring function used in A*, $f(n) = g(n) + h(n)$. However, the values for g and h take on a slightly different meaning. In A*, g(n) is the actual optimum cost to arrive at n from the *starting* location. In LRTA* however, g is the cost to n from the agent's *current* location. Thus, this score is only valid until a move is made by the agent[‡].

For $h(n)$, initially this value is the heuristic cost from n to the goal, the same as A*. However, this changes after the agent makes a move. Once all the nodes with the agent's d-neighbourhood are evaluated, LRTA* chooses the node with the lowest f-value as the subgoal for that iteration, and the agent makes a move towards this goal. It is here where LRTA* differs from A* for heuristic scores.

If the agent was at node a and LRTA* dictates a move towards node b, the value of $h(a)$ is updated to match $h(b)$[§]. Doing so serves the following purpose: if the

[‡]This is also technically the case for A* since the agent does not move until the algorithm completes.

[§]Current h scores for every node get stored in a hash table for later lookup.

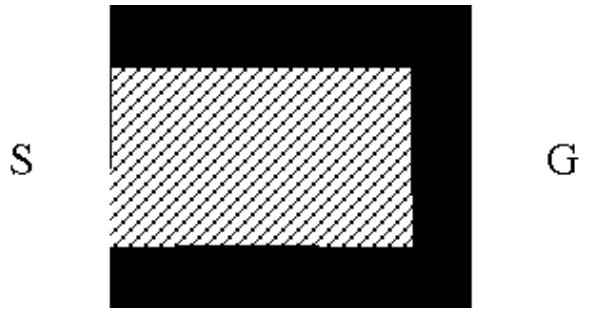


Figure 2.2: An example of a *heuristic depression* [Ish92] encountered with LRTA*. Moving from S to G, an agent will be 'trapped' in the shaded area until heuristic scores within it become sufficiently high.

progression of the algorithm causes the agent to return to a, the extra work needed to return is remembered in the heuristic score. This key insight allows an agent using LRTA* to search out of a local minimum if no path to the goal is found within it.

Since LRTA* assumes the use of an admissible heuristic, the initial heuristic value is assured to be no more than the true cost h^* for a node. Because of this, over time the subsequent updates to the heuristic for n approach $h^*(n)$. Over multiple runs, heuristic scores are updated less and less as they approach the true cost to the goal, and the agent follows a more optimal path to the goal. The optimal path from start to goal is found when no updates are necessary in a given run.

It has been noted however that the first-run performance of LRTA* can be quite poor. Korf noted that if an agent is inside a local minimum surrounded by states with higher scores, then LRTA* will 'bounce back and forth ... until it "fills in the hole", ... at which point it will escape to the rest of the graph.' Ishida has shown that the number of runs to determine an optimal path (the *convergence process*) can take quite some time due to what he termed *heuristic depressions* [Ish92]. These are areas in the problem space that have to be 'filled' with the true heuristic scores before an agent can learn its way out. Figure 2.2 shows an example of such a depression.

Because first-run performance in turn affects the rate of convergence to an optimal path, subsequent work in real-time pathfinding has been largely devoted to developing

variants to LRTA* that return better results after the initial run.

2.2.2 K LRTA*

Koenig introduced a variant LRTA* to improve first-run performance in [Koe04] that has since been referred to as K LRTA*. In this new algorithm, the breadth-first local search approach of LRTA* is replaced with an A*-shaped search that terminates after n expansions. This local A* is conducted from the agent's current location towards the goal state. Once search is concluded, the minimum path between the current location and the node with the minimum score along the local search fringe is followed, much like LRTA*. However, instead of a single step the agent moves along this path until it reaches the end at the fringe of the A* search, or it is found that the path is invalid due to an obstacle because of the freespace assumption. Updates of heuristic scores are done within the local space defined by the closed list after the A*'s n iterations. The h scores are updated for *all* nodes in the closed list. The scores of these nodes are updated using Dijkstra's algorithm [Dij59].

2.2.3 LRTA*(k)

LRTA*(k) was proposed by Hernández and Meseguer in [HM05] to speed up changes to heuristic scoring. They proposed allowing up to k updates to heuristic scores per step, beyond the single heuristic update in LRTA*. In this variant, when the heuristic value of some node v is updated, then the successor states of v are also considered, up to a maximum of k successors. This bound of k contrasts K LRTA* where, in the worst case, the number of potential updates are the complete closed list of the A* space, which is bounded by the branching factor of the search space after the n expansions. It is not necessarily the case that the k nodes are distinct; a single node can be reconsidered multiple times, with each consideration counting towards the k updates. An additional condition is that updates can only occur on nodes that have

been previously visited by the agent. The order of consideration is handled with a queue. The authors note that with $k = 1$, their algorithm is simply LRTA*, as the only heuristic considered for some iteration is the one being left by the agent.

Later in [HM07], Hernández and Meseguer relaxed the constraint on updates only applied to previously visited nodes. Instead, the k updates can apply anywhere in the local space. This updated variant was termed LRTA*$_{LS}$(k). Relaxing this constraint increased the convergence speed to an optimal path over their original implementation.

2.2.4 RTAA*

Koenig proposed another variant called Real-time Adaptive A* (RTAA*) in [KL06]. This variant builds on the previous K LRTA* in [Koe04]. Instead of conducting a second Dijkstra search inside the space explored by each A* shaped search to update heuristics, Koenig proposed that RTAA* utilise scores from the A* search directly. This was a further improvement over K LRTA*.

2.2.5 P-LRTA*

Rayner et al. proposed Prioritised-LRTA* (P-LRTA*) in [RDB+07]. The pseudocode of P-LRTA* is shown in Algorithm 3. At the start of each search iteration, P-LRTA* examines only the immediate neighbours of the current state (i.e., breadth-first search of depth 1) and looks at the neighbour with the best f-score. This node is placed into a priority queue, with priority equal to the magnitude of the heuristic change. Local search is conducted by processing this priority queue, with a node's neighbors being placed onto the queue when it is processed. The agent chooses a move to the most promising neighboring node of its current location after the queue has been processed up to n times. P-LRTA* has several differences to other LRTA*-based algorithms. The queue is not emptied after every move (lowest magnitude nodes are dropped

if the queue is full), and unlike LRTA*(k) duplicate entries are not allowed in the queue. Rayner also notes that because the queue persists across moves, the 'shape' of the update space is not well defined or necessarily contiguous, unlike K-LRTA*'s A* shaped space.

Algorithm 3 P-LRTA*

```
1: while loc ≠ goal do
2:     update(loc)
3:     for 1..N, queue not empty do
4:         next ← queue.pop()
5:         update(next)
6:         queue.push(successors(next))
7:     end for
8:     loc ← min neighbor(loc)
9: end while
```

2.3 Hierarchical Real-Time Algorithms

There has also been a body of work into real-time heuristic search that leverages the use of abstraction techniques. This has led to multiple additions to the family of algorithms, some of which are also based on the work of LRTA*.

Hierarchical pathfinding draws its motivation from the way people naturally plan out trips at different levels of detail. [BMS04] gives the example of planning a trip from a particular address in Los Angeles to one in Toronto. Given a complete, high detail map of the entire North America road network, A* would be used to determine the shortest route down to the metre, but given the scale of the problem this would be highly impractical. Instead, a human planner plans a more abstract route between cities, with the low level detail of navigating individual streets left undetermined until they enter a particular city.

It is this style of planning that is used in Hierarchical Path-Finding A* (HPA*) applied to two-dimensional grids. The problem space is separated into rectangular

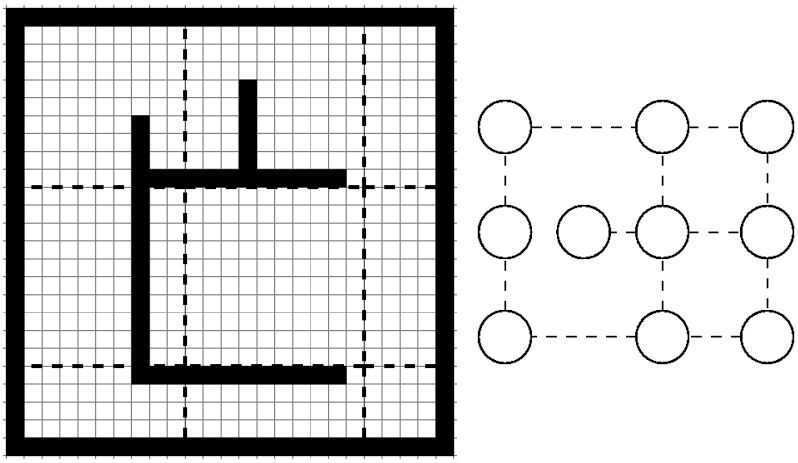


Figure 2.3: An example of graph abstraction. The grid cells are partitioned into rectangular clusters, denoted by the dashed lines.

spaces in a regular fashion, with all nodes inside a rectangle comprising a *cluster*, as seen in Figure 2.3. An abstraction of these clusters is produced by identifying nodes along the border of these clusters that are adjacent to other such nodes in neighboring clusters. Nodes of the abstracted graph are correlated to these cells. Edges are between all nodes common to a cluster by identifying an optimal path between these points. Valid paths are limited to those that are enclosed entirely within the cluster. The cost of the edge is defined as the cost of the optimum path between the nodes. Intra-cluster connections are defined by joining the two adjacent nodes of a cluster pair with an edge.

The resultant collection of nodes and edges produces a highly relaxed version of the original grid. This abstract grid itself can be abstracted again producing an even more abstract grid; the process could be repeated until the entire problem space is abstracted to a single node, producing a hierarchy of abstraction levels. The underlying grid is level 0 (L_0). For a full detail map of the North American road network, L_0 would correspond to the individual streets. The initial abstraction is L_1, and subsequent abstractions L_2, L_3, and so on. This could correspond to abstractions to the level of individual cities, then counties, then abstracted to states and provinces,

then countries. The topmost abstraction would encompass the entire network.

Pathfinding on this hierarchy of abstractions occurs in a top-down approach. Given start and goal locations S and G, the cluster(s) they belong to are identified and a path is planned between them using the edges of the abstract level. This is followed by a series of refinements as each path edge is defined by a path traversing the cluster(s) at the next lowest level of abstraction. This refinement continues down to L_0.

If S_u^k is understood to be a node u in level k, with the cost of an edge E_{uv}^k between u and another node v, the path for this edge on level $k-1$ can be cached.

HPA* however does not qualify as a real-time pathfinding algorithm. While the grid abstractions bring a divide and conquer approach to heuristic search, there is no constant bound on planning for any given level. At the lowest level, paths between grid clusters are still traced out with classic A*. This style of pathfinding remains subject to problem size.

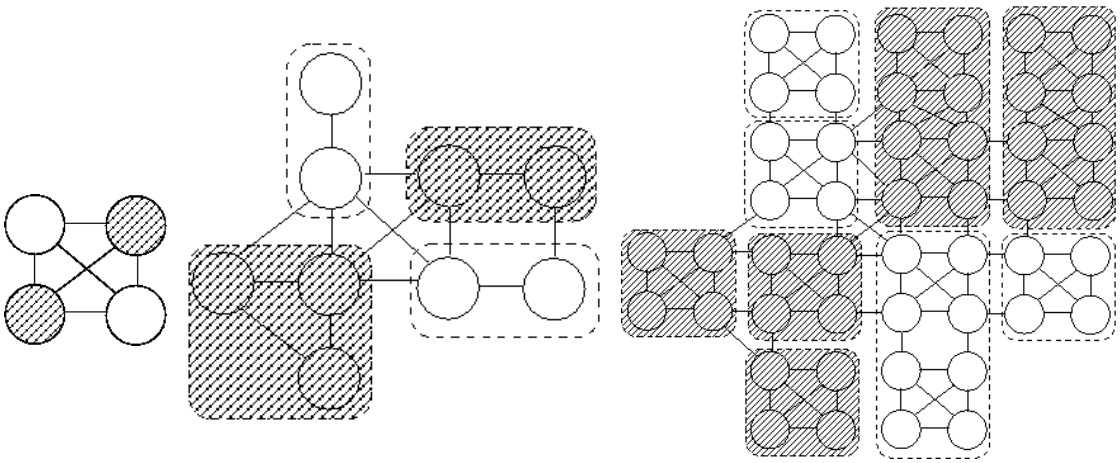
2.3.1 PR LRTA*

State abstraction for real-time search was applied in [BSLY07] which introduced a new LRTA* variant called Path Refinement Learning Real-Time Search (PR-LRTS, later referred to as PR-LRTA* [BBS09]).

Instead of producing an abstraction by cutting up the L_0 grid into rectangular clusters, PR LRTA* abstracts groups of nodes by cliques of completely connected nodes, a technique described in [SB05].

PR LRTA* builds of abstraction hierarchy using the clique technique from HPA*, up to l levels of abstraction. LRTA* search is then conducted in this abstract space.

Because the states that LRTA* are working on are at an abstract level, this single move is potentially a path of multiple moves at the ground level grid. In this corridor at the ground state, A* search is conducted to determine how the agent should move to complete the move at the higher abstraction. Figure 2.4 illustrates this. L0 is



(a) L2 (b) L1 (c) L0

Figure 2.4: Corridor produced by applying LRTA* to an abstract hierarchy in PR LRTA*. A move determined by LRTA* at the top-level abstraction (a) defines a corridor for A* search at the ground-level abstraction (c).

the original search graph, with successive abstraction layers L1 and L2. If LRTA* determines a move between the two unshaded nodes in L2, this defines the unshaded corridor in L0 to define the boundary for A* search. This process repeats until the top level LRTA* search defines a corridor reaching the goal.

Real-Time performance is guaranteed the same way it is assured for LRTA*. With the LRTA* search limited to a maximum depth, an upper bound is effectively placed on the maximum size of the corridor at the ground level, regardless of the actual layout. This places a constant bound on the number of A* expansions performed at the ground level.

2.3.2 D LRTA*

D LRTA* [BLS+08] is a rather significant departure from classic LRTA*, incorporating several ideas into a single algorithm. Whereas LRTA* searches out all nodes up to d steps away, D LRTA* searches out to d' steps, where d' can be dynamically adjusted every iteration. The value to use for d' on any particular iteration is de-

termined though the use of a *classifier*. This classifier takes in information about the agent's recent performance and makes a constant time decision as to what the depth should be. The classifier described uses several statistics easily computed in real-time, including the heuristic estimate of the agent's current distance to the goal. Real time behaviour is assured by placing an upper bound on d', which thus places a constant upper bound on the number of expansions for a branching factor. This is the same idea behind the real-time claims of LRTA* or PR LRTA*.

The second characteristic to D LRTA* is the use of a database to store previously computed partial solutions. Here, the problem space is abstracted to some level Ll using a clique abstraction. Then, for each pair of l nodes, the first action to take between each is calculated and stored. Because the l nodes are an abstraction of many nodes at $L0$, a representative node from $L0$ is chosen here for its respective l node.

2.4 Time-Bounded A*

While the introduction of abstraction hierarchies produced results with an order of magnitude improvement over LRTA*, this does come at some cost. Abstraction hierarchies are resistant to use on dynamic maps. Any change at the base grid L0 can affect edge costs for the next level, forcing re-computation of the abstract graph. In the worst case the change of a cut-edge can affect the abstraction hierarchy in its entirety. With algorithms like D LRTA*, which relies quite heavily on the off-line computation of a pattern database, extensive re-computation happens for even a single map [BBS09].

This led to the introduction of TBA*, a real-time algorithm that does not use the abstraction hierarchies or learning approaches described in the previous sections. Rather, TBA* is best described as a version of time-sliced A* (A* that can be halted and resumed) that meets the guarantees of a real-time search algorithm. This sec-

tion will provide a somewhat detailed explanation of how TBA* works. As it is a recently introduced algorithm, much of the following information in this section is taken directly from [BBS09].

Algorithm 4 TBA* $(start, goal, P)$ from [BBS09]

```
1:  solutionFound ← false
2:  solutionFoundAndTraced ← false
3:  doneTrace ← true
4:  loc ← start
5:  while loc ≠ goal do
6:      if not solutionFound then
7:          solutionFound ← A*(lists, start, goal, N_E)
8:      end if
9:      if not solutionFoundAndTraced then
10:         if doneTrace then
11:             pathNew ← lists.mostPromisingState()
12:         end if
13:         doneTrace ← traceBack(pathNew, loc, N_T)
14:         if doneTrace then
15:             pathFollow ← pathNew
16:             if pathFollow.back() = goal then
17:                 solutionFoundAndTraced ← true
18:             end if
19:         end if
20:     end if
21:     if pathFollow.contains(loc) then
22:         loc ← pathFollow.popFront()
23:     else
24:         if loc ≠ start then
25:             loc ← lists.stepBack(loc)
26:         else
27:             loc ← loc.last
28:         end if
29:     end if
30:     loc.last ← loc
31:     moveagenttoloc
32: end while
```

The pseudocode of Time Bounded A* is shown in Algorithm 4, as it appears in their paper. Lines 5 through 32 constitute the main portion of the algorithm. The

algorithm is broken down into distinct planning and execution sections. With every iteration, lines 6 through 20 encompass the planning portion, where a round of search is carried out. Lines 21 through 31 cover the execution phase, which determines the move an agent will perform. The algorithm continues to run until the agent has reached the goal (line 5).

2.4.1 Planning

The planning portion of TBA* is broken down into two distinct steps, expansion and tracing. On line 7, expansions are performed by running A* search if the solution has not been found in some previous iteration. The parameter *lists* encompasses both the open and closed lists, which are reused across iterations to facilitate global search. If A* finds the goal, this is recorded and search is suspended for subsequent iterations, if any.

Lines 9 through 19 handle the generation of paths for the agent to follow. On the first iteration, the most promising node on the open list - the node that will be expanded next when A* resumes - is used to begin tracing a path on line 13. Paths are constructed by following the parent references of each node until the starting node is reached, or the node where the agent is currently located.

Tracing a path may take multiple iterations. When a path is traced out, it becomes the current path an agent is following (line 15). The process repeats until a path ending at the goal is traced out. Figure 2.5 shows an example of the different paths TBA* can produce for a search.

To claim real-time performance, there must be a limit on the amount of planning in a given iteration. In TBA* one of the parameters defined is the *resource limit*, R[¶]. R is broken down into limits on the amount of work allocated per iteration to node expansion and path tracing respectively. The limit on the number of expansions (N_E)

[¶] R is never given a unit definition in the paper, but it generally understood to represent the effort in CPU time to expand a single node, subject to the particulars of platform and implementation.

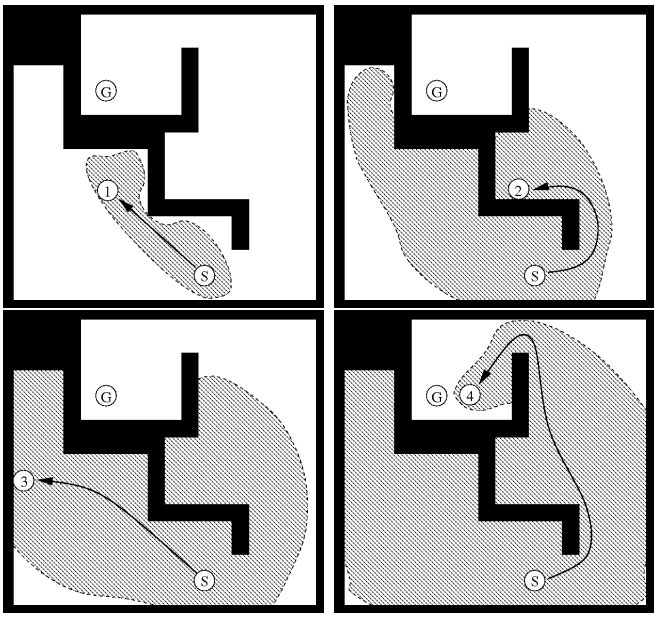


Figure 2.5: A depiction of successive subgoals for Time Bounded A* at different intervals. Each number represents the node returned from mostPromisingState, along the search fringe at that point with the arrow being the path traced to it from the start S. The algorithm terminates when the agent (not shown) arrives at the goal G. The shaded area is the area covered by the open/closed lists for that iteration.

for an iteration is:

$$N_E = \lfloor R \times r \rfloor \quad r \in [0, 1]$$

For example, with $R = 200$, and $r = 0.75$, $N_E = 150$, or 150 node expansions are performed in the A* portion of the algorithm every iteration. The remainder is allocated to performing to tracing out paths (N_T), determined as:

$$N_T = (R - N_E) \times c$$

Here, c is a coefficient. The authors of TBA* pointed out that tracing out a path usually involves following pointers, an operation that is faster than node expansion. Hence, $c = 5$ would mean tracing a single step on a path is 5 times faster than expanding a node. Thus, using the earlier values of R, r and c, N_T would be 250.

2.4.2 Execution

The second half of the algorithm is concerned with execution, where moves are actually performed by the agent in accordance with a simple rule. The agent checks if it is on the path it is currently following, if so, it takes a step along the path (line 22), otherwise it takes a step backwards (line 25). This assures that the agent will eventually return to the path that TBA* has selected for the agent to follow. Because the closed list an acyclic tree and all paths are built from this tree, if the agent keeps stepping back it is assured to eventually end up on the path it is currently set to follow.

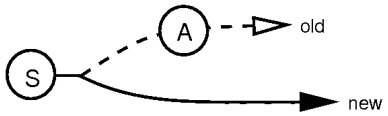


Figure 2.6: When TBA* switches paths to a newer one, the agent may be required to retrace steps by moving backwards until it reaches the latest path. In the worst case, the agent returns to the starting node.

For clarification, it is not necessarily true that the new path TBA* that the agent backtracks towards is the same path the agent will reach and begin following. It is conceivable that TBA* will identify and trace out any number of paths while the agent is backtracking. The decision to take a step along a path or backtrack is always made relative to the current path referenced by $pathFollow$.

Once the algorithm identifies a path from the starting node to the goal, no more path switching occurs. If the agent is not somewhere along this path, it will backtrack until it is, and follow this final path to the goal. The agent is guaranteed to end up on this path; in the worst case it will return to the starting node.

2.4.3 Real-Time Claim

To claim that TBA* is a real-time algorithm, the amount of search the algorithm is able to perform in the expansion phase should remain constant despite problem

size. In other words, the algorithm should be able to perform the same number of expansions with each successive iteration and not taking longer to do so. To do this, the authors have constant-bounded state expansion.

For A* based search, the open list is typically implemented as a priority queue using a binary heap. Nodes on the open list are prioritised based on f-score. While retrieval of the next state to expand is an O(1) operation, insertion of successors is $O(\log n)$, with n being the number of nodes on the heap. Because n grows with the problem size, any algorithm that maintains an open list across iterations using a heap cannot be real-time.

To claim real-time, TBA* uses a variation of a data structure used in the *Fringe Search* algorithm [BEHS05], which is an evolution of Iterative-Deepening A* (IDA*) [Kor85]. Fringe search was focused on improving the run-time performance of IDA* type algorithms by facilitating the reuse of the open list (the search fringe) across successive iterations, negating the rework normally necessary.

Key to this algorithm was the introduction of two lists, the *now* and *later* lists, to collectively store the nodes of the Open List. Each iteration, the algorithm runs over all nodes in the *now* list, either deferring their expansion by moving them to the *later* list, or performing an expansion, with the successors going into the *later* list for the next iteration. On the next iteration, the *later* list is now the *now*. The authors mention that IDA* iterates in a left-to-right fashion, whereas A* is best-first, which requires sorting. Fringe Search can sort or partial sort by using multiple buckets for the *later* list based on f-score. This notion was carried over to TBA*. Instead of *now* and *later* buckets, TBA* uses a bucket for each f-score. These buckets are stored in a hash table, keyed by f. Add and Remove operations on these buckets are O(1).

With the open list being a constant-time data structure, the bound on real-time now hinges on how many nodes are expanded each planning stage. TBA* runs A* for a fixed number of iterations. The number of iterations is the parameter N_E.

2.5 Summary

This chapter examined several algorithms that define the current state of real-time pathfinding applied to grid-based problem spaces. Since Korf defined the area with the introduction of LRTA*, much of the work since has focused on improving the rate at which LRTA* based algorithms converge on an optimal solution. This has typically been achieved by developing variants that feature improved first-run performance. This is important in the context of video games since subsequent trials are not likely to occur, and *any* occurrence of poor behaviour is to be avoided if possible.

Time-Bounded A* brings first-run optimal solutions to real-time pathfinding by integrating classic time-sliced A* with a simple set of rules to govern agent behaviour. But, while the first-run performance of TBA* is an improvement over previous algorithms, the binary advance-or-retreat nature of execution means improvements can still be made.

Chapter 3

Salient Search

3.1 Motivation

With TBA*, it was noticed that the agent could often spend time moving back and forth as the (current) path changes from one subgoal to another. The agent would have to backtrack in order to put itself back on a path to follow, like in Figure 3.1. If arriving on the current path requires k steps backwards, those k steps add to the total travel cost of the agent without necessarily moving the agent any closer to the goal. This increases the *Travel Ratio* of the search, work the agent is performing that adds up above the cost from the start to the goal.

Salient Search began with the following idea: at time t, the agent may be closer to the goal than the cost of that return trip to follow the optimal path found for S, G. Since the cost is already expended by the agent to reach its current location, it may be cheaper to move forward if such a path exists and can be found in time. This is like the expression 'the point of no return'.

Figure 3.2 illustrates this idea. The agent A is moving towards a subgoal g by following a given path. Suppose a solution is found from the start to the goal G, given by the solid arrow. If the path isn't defined through the agent's current location, the algorithm dictates the agents backs up along the path it has travelled until it is. But,

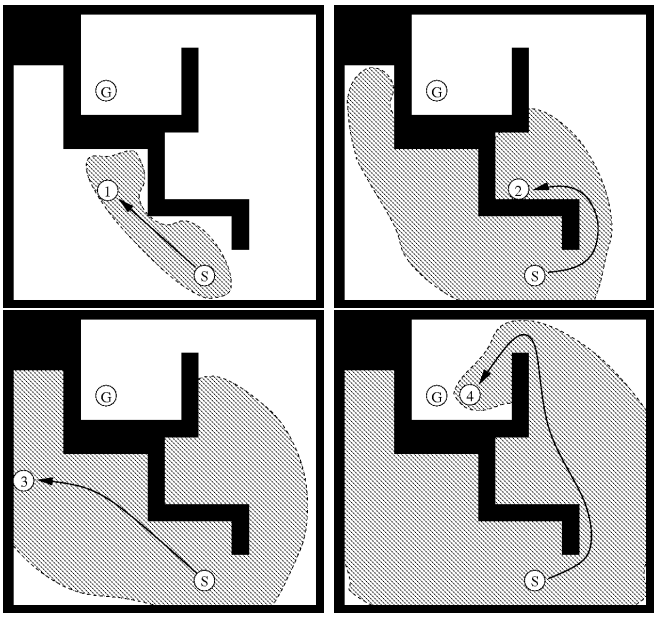


Figure 3.1: Each successive path shares no common section with the previous one. The agent will likely have to backtrack to the start.

it is possible that there is a path from g to the goal that is short enough that, if known about, is more desirable than backing up. In other words, if $A \rightarrow g \rightarrow G \leq A \rightarrow S \rightarrow G$, backing up is undesirable.

Finding such a path requires search. However, search is halted when the goal is found. Also, since TBA* is running A* search, it is known that A* finds the optimum path before any other. A* on its own will not find a path through g first.

This calls into question the notion of what now constitutes an optimum path. There is no question that from S, A* and thus TBA* will find the best path. But not taken into consideration is reality that the agent is no longer at S, and such an optimum path may no longer be so. Starting A* over again from the agent's current location will not do; the search horizon imposed by real-time requirements invites isolating the agent in a heuristic trap of oscillating moves. This is the very thing real-time algorithms like TBA* avoid, and LRTA* eventually does when the heuristic cost updates converge. Updating all scores in the search space to reflect the current position cannot be done in real time. Thus, without a change to the 'best from S'

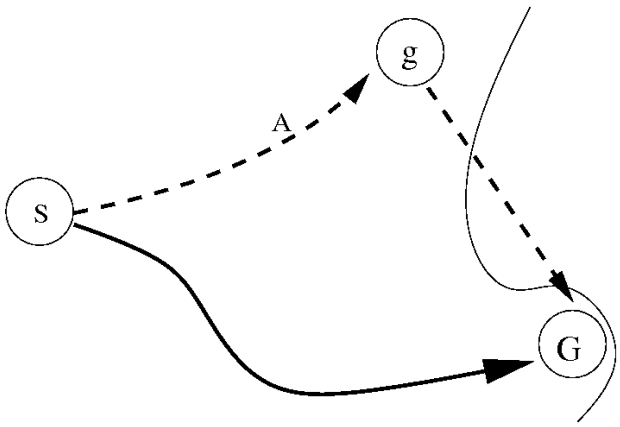


Figure 3.2: The agent is moving towards g when a solution is found. In TBA*, the agent backs up along the path $A \rightarrow S \rightarrow G$. However, it may be desirable to move $A \rightarrow g \rightarrow G$.

search order, a solution passing through g will not be found.

As well, on harder problems or problems involving a poor heuristic, A* performance tends to degrade. The search frontier expands outward in a more uniform fashion resembling breadth-first search as the best-first strategy of A* is weighed down by many nodes sharing the same score. When this happens, the lengths of successive paths from subgoals increases more slowly. These paths are also more divergent, possibly sharing only the start as a common node. When this happens, agent behavior becomes erratic. The effect on the agent resembles more of a random walk than a real attempt to move towards the goal.

Salient Search is proposed to attempt to address this situation, by allocating search from the last subgoal, while retaining the information provided from the global search. The rationale behind this concept takes two forms.

Salient Search is Time Bounded A* with the application of two new concepts:

- An additional data structure to allow us to define the salient and force search to occur along its fringe.
- The introduction of strategies to facilitate choice of subgoals, not simply relying on the open list's next node.

3.1.1 Why 'Salient' Search

The use of the word salient is derived from the military notion of the phrase, where it describes a projection into enemy territory along a front line, a spot in the topology of the battlefield where a force has broken through the front to advance towards some objective. Comparatively, in pathfinding the progression of search using a consistent heuristic can be visualised with contour lines [RN03], and a contour defined by the Open List of a search, which is sometimes called a *fringe*. The analogy here is as follows: A* search produces a series of contours as search progresses. Salient Search breaks through the contour by performing search at one point of the contour, producing a salient.

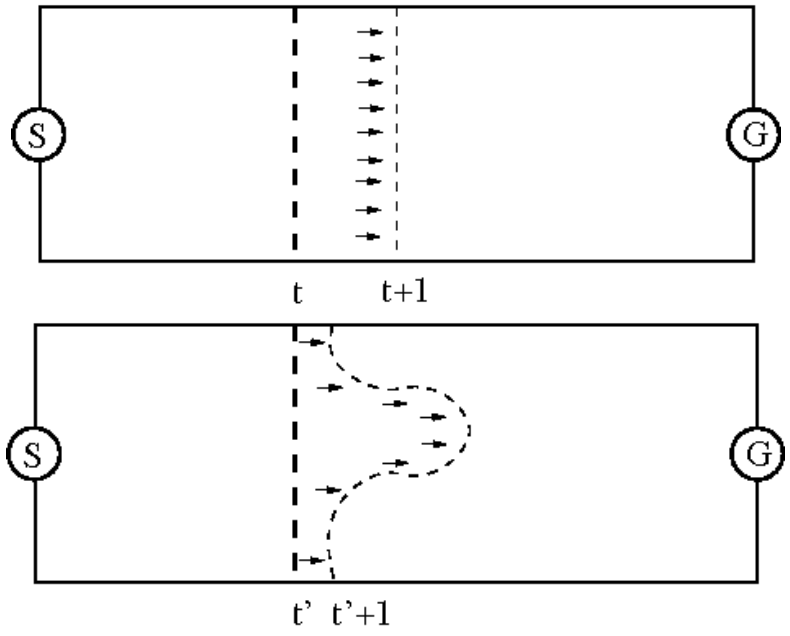


Figure 3.3: A visual representation of a salient applied to heuristic search. The salient is the bulge of explored space beyond the border that would be defined once time $t+1$ is reached.

This concept is illustrated in Figure 3.3. Here the search space between the start and goal has been transformed into a 2-dimensional corridor. Normal A* search is

seen progressing as a straight line sweeping across this corridor at the top of the figure. As the algorithm iterates from time t to time $t+1$, this sweep line has moved forward, much like an army moving a front line forward with a collective advance. Eventually, this line reaches the end as the goal is reached and a path can be traced back.

With Salient Search however, the normal progression is interfered with. In the bottom portion of Figure 3.3, at time t a point along the fringe is chosen, and in the next iteration work is applied to expand this node and its successors. As a result, space that may not have been searched out by time $t+1$ instead has been, distorting the normal progression of search. This is illustrated with the search line reaching past the original A* line in one area. This is the salient: attention was focused at one point to move the frontier beyond where it would be expected to be. This comes at the expense of other areas, where the line is not as far to the right as it would otherwise be.

3.2 Salient Expansion

Applied to our real-time pathfinding algorithm, the salient itself begins as a single node in the search space. Recall that TBA* builds a path by choosing a suitable candidate from the open list, with the $mostPromisingState$ function, which becomes the endpoint of said path. The salient begins from this same node. Every time a salient is initially defined, it begins with this single node, which we term the *salient root*. But where the path is built by backtracking through the search tree, the *salient* is defined by moving deeper through the growing search tree.

Also recall that on the next iteration of the A* search, this candidate node will be the next to be expanded, possibly generating successor nodes that will then be placed into the open list. Because their parent (the root) is part of the salient, successors will be part of the salient as well. This process continues - new salient nodes coming

from the successors of previous nodes - until a new root is defined, at which point membership in the salient is no longer defined. Subsequent additions to the salient begin from the new root.

The *salient list* then is to be understood as all nodes currently on the open list that are descendents of the current *salient root*. Salient expansion is conducted with every node expansion when the node is itself on the salient.

Algorithm 5 shows pseudocode of the Salient Search algorithm. This is the same as TBA* in Algorithm 4. However, two lines are changed. On line 7, the A* search has been replaced with a version called SalientA*, which takes in an additional parameter. The other change is on line 11. In TBA*, *pathNew* was set to the head node of the Open List. Here, a function is called that takes in a *strategy* parameter.

A node on the open list is chosen as the salient root on line 11 of Algorithm 5. At this point, the salient is the root itself. When this node is expanded, it is removed from the open list, and its successors (if any) are now members of the salient when they are inserted onto the open list. This is true for their successor nodes, and so on. The definition of the salient in this way is illustrated in Figure 3.4.

These expansions happen up to N_S times. N_S itself is described in 3.2.1. The remaining work is done following the order dictated by the open list. If a node to be expanded normally is referenced in the salient, it is treated like a salient node as a means of preserving the structure of the salient - its successors are placed onto the salient list.

3.2.1 The Salient List

The data structure of the salient list is a modified version of the bucket structure that TBA* adapted from [BEHS05]. This altered structure incorporates a second version of the bucket list to reference nodes as belonging to the salient. In this salient list, pointers to nodes are stored, as opposed to search nodes themselves. Each value in a bucket is a pointer to a corresponding node in the open list. These pointer buckets

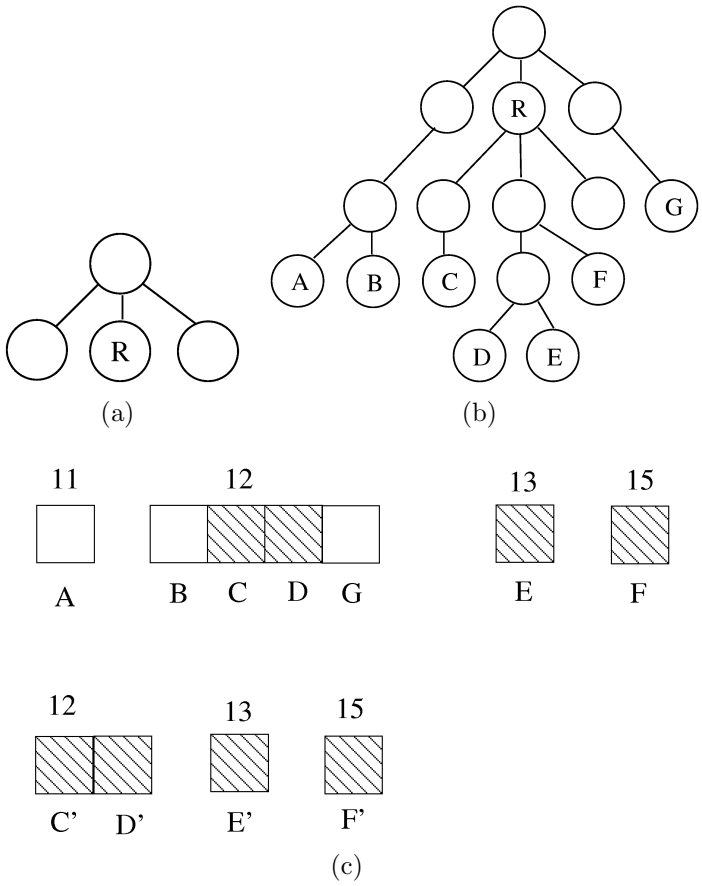


Figure 3.4: The salient on the search tree. At some point R is chosen as the salient root when R is in the open list for the tree on the left. From then on, all successors of R on the open list define the border of the salient. If nodes A through G are the open list, then the salient is defined by {C,D,E,F} in the tree on the right. The bottom portion illustrates what the open and salient lists look like at this point, with each set of blocks corresponding to a bucket, and the number above each being the f scores (and hash key of the bucket) of the nodes therein.

are notated with the same f scores as the parent bucket they contain references for.

For the open list, node order within a particular bucket is dictated by insertion order. This is because the bucket itself is implemented as a queue[†].

[†]Specifically, a Last-In, First-Out (LIFO) queue, or stack.

Algorithm 5 Salient Search $(start, goal, P)$

```
1: solutionFound ← false
2: solutionFoundAndTraced ← false
3: doneTrace ← true
4: loc ← start
5: while loc ≠ goal do
6:     if not solutionFound then
7:         solutionFound ← SalientA*(lists, start, goal, N_E, N_S)
8:     end if
9:     if not solutionFoundAndTraced then
10:        if doneTrace then
11:            pathNew ← lists.nextSubGoal(loc, strategy)
12:        end if
13:        doneTrace ← traceBack(pathNew, loc, N_T)
14:        if doneTrace then
15:            pathFollow ← pathNew
16:            if pathFollow.back() = goal then
17:                solutionFoundAndTraced ← true
18:            end if
19:        end if
20:    end if
21:    if pathFollow.contains(loc) then
22:        loc ← pathFollow.popFront()
23:    else
24:        if loc ≠ start then
25:            loc ← lists.stepBack(loc)
26:        else
27:            loc ← loc.last
28:        end if
29:    end if
30:    loc.last ← loc
31:    moveagenttoloc
32: end while
```

Ordering

To ensure that the salient list can be used in a consistent manner, expansions on the salient list follow the same best-first approach applied to the open list. In fact, expansion on the salient is done using the same A* search, with expanded nodes being

Algorithm 6 SalientA*$(lists, start, goal, N_E, N_S)$

```
1: i ← N_S
2: goalFound ← false
3: while i > 0 and salientList is not empty and not goalFound do
4:     i ← i − 1
5:     if salientList.next() = goal then
6:         goalFound ← true
7:     end if
8:     expandSalient()
9: end while
10: for remaining N_E − N_S + i and not goalFound do
11:     goalFound ← A*
12: end for
```

Algorithm 7 expandNext()

```
1: next ← openList.next()
2: if salientList.contains(next) then
3:     expandSalient()
4: else
5:     move next to closedList
6:     for successor in next.successors() do
7:         if not closedList.contains(successor) then
8:             updateOrInsert(successor)
9:         end if
10:    end for
11: end if
```

placed onto the same global closed list.

A node is added to the open list at its creation in the search, when its parent node is expanded. In Salient Search, there are two separate cases where expansion occurs: when a node is expanded on the open list, and when it is expanded on the salient list. If salient expansion is occurring then we know the successors will also be part of the salient and the successors are also added to the salient. If open list expansion is occurring, the node is checked for membership on the salient before it is expanded, and switching to salient expansion if it is. This ensures the successors are added to the salient list.

Algorithm 8 expandSalient()

```
1: if salientList not empty then
2:     next ← salientList.next()
3:     move next to closedList
4:     for successor in next.successors() do
5:         if not closedList.contains(successor) then
6:             updateOrInsertSalient(successor)
7:         end if
8:     end for
9: end if
```

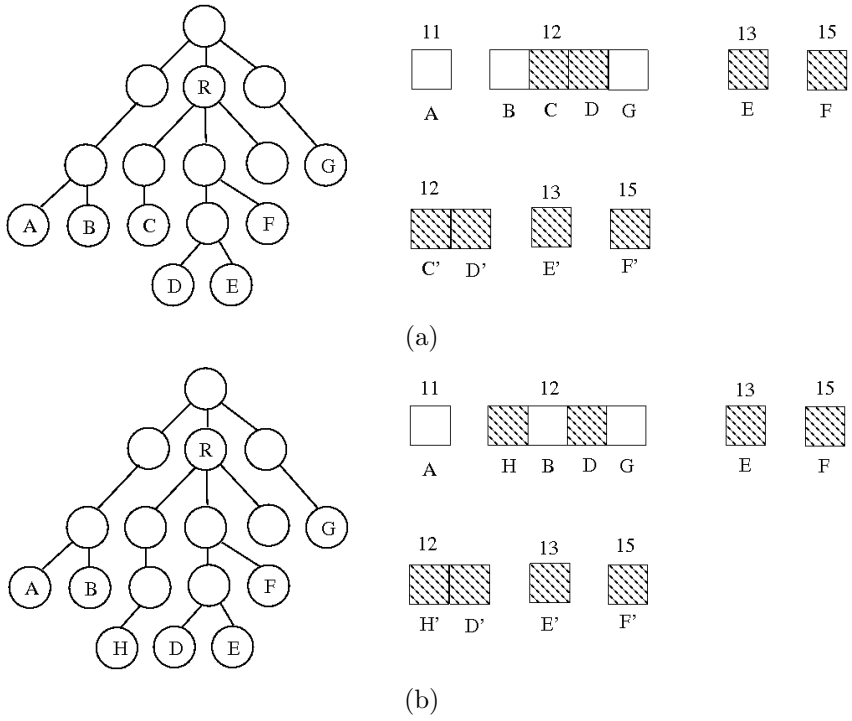


Figure 3.5: The next node to be expanded by salient expansion is always identified by the head of the salient list. Here it is node C. It is removed from the open list, and its successors are put into both lists in the preferred order.

In TBA*, nodes are only removed from the open list in best-first order. In Salient Search however, the best of the salient may be located in a different location.

When successors are generated, the node generated may match another node already on the open list, but with a better f-score. This is possible in Salient Search

even if a *consistent* heuristic is being used since Salient Search does not follow A*'s expansion order globally[‡].

With a heap based open list, this would involve updating the node's key, a $\Theta(\log n)$ operation. However, because the open list in this algorithm uses constant time ($O(1)$) operations for adding and removing nodes, those operations can be used to achieve an update.

Thus, with all actions appropriately mirrored on the salient list, it is assured to be a subset of the open list with the following properties:

- All nodes on the salient list are descendents of the salient root
- All descendents of the salient root that are on the open list are also in the salient list
- The salient list maintains the same ordering as the open list

3.2.2 Allocating Work - N_S

To control the amount of expansion limited to the salient, a new parameter is introduced. N_S controls the number of expansions per iteration that are allocated to ensuring work is done in the salient, being a value $0 \leq S \leq E$. Any remaining work is used to perform expansions as normal by the ordering of the open list.

A particular value for N_S however is not a guarantee that N_S salient expansions will be performed. First, if the salient list is exhausted, then salient expansion obviously cannot be performed. This is possible if for example the salient root expands into a closed off space search and is exhausted within it. Secondly, through regular expansion, a salient node could be selected for expansion through regular A* search when the node is at the top of the lowest scored bucket. If this occurs, it is treated as a salient expansion, with successors being placed on the salient list (Algorithm 7,

[‡]For this reason, it was decided to exclude 'A*' from the name of the algorithm, unlike TBA* or LRTA*.

lines 2-3). Detecting this is a trivial operation because the node will be at the head of both the open and salient lists. With these two situations, the number of salient expansions in any given iteration can exceed or be less than N_S.

When N_S is 0, no action will be taken to perform node expansions out of order, effectively turning off salient expansion. Search will proceed exactly as it would in TBA*. That is to say, the planning portion of the algorithm will remain unchanged, proceeding in the same way A* does. This is not to say that the salient will not exist. The definition of the salient list is maintained from R like in Figure 3.4, there is simply no explicit allocation of work to produce successors from the members of this list. The list remains maintained for use with the *strategy*, which is described in Section 3.3. Strategies affect the choice of paths, so even with search proceeding exactly as it would under TBA*, Salient Search can result in different behaviour for an agent given the same problem.

3.3 Subgoal Selection by Strategy

Recall that TBA* produces paths by choosing a subgoal. These subgoals are always the most promising node currently residing on the open list. This subgoal becomes the endpoint of a new path. As long as this node is not the goal, search will continue, because the subgoal is always the next node to be expanded. This remains true in Salient Search. But, the salient list now provides additional information regarding this decision. We know which nodes are descendents of the path that has just finished being traced back, allowing us to know if a prospective subgoal will produce a path which lengthens the current one. This information can be used to modify the selection of a subgoal; the decision can be made to instead selected the most promising node of the salient list instead.

This choice is made by using a heuristic termed a *salient strategy*. A *strategy* is any function which uses the head nodes of the open and salient lists along with any

other parameters necessary, and returns which one will serve as the next subgoal. The decision will decide not only if the next path is an extension of the current one, but also whether the new salient will be an extension of the current salient.

Trivially, if the salient list is empty or the head of the salient list is the same node as the head of the open list then the decision is moot. The only other restriction on a strategy is that it satisfies the realtime requirements.

What follows are two example strategies. The first is a simple tie-breaking strategy for nodes with the same f score. The second strategy makes use of the agent's location.

3.3.1 Simple Tie-Breaking

In the *Tie-Breaking* (TB) strategy, the preference for a subgoal with the best f score remains the same. However, as mentioned in 3.2.1, the salient list allows constant time determination of whether a potential subgoal extends the current path. Extending the current path means the agent will not require backtracking if it is following the current path. Given the lack of sorting between all nodes with the same f-score, we can use this information to choose a subgoal. If the head nodes of lists are n_{Open} and $n_{Salient}$ respectively, the strategy is:

$$TB(n_{Open}, n_{Salient}) = \begin{cases} n_{Open}, & \text{if } f(n_{Open}) < f(n_{Salient}) \\ n_{Salient}, & \text{Otherwise} \end{cases}$$

Since by definition n_{Open} is the head of the lowest f-bucket on the open list, $n_{Salient}$ will be in the same bucket, or have a higher f-score. Because the salient list has the same ordering as the open List, n_{Open} is not a descendent of the (current) subgoal unless the two are the same node.

3.3.2 Agent Heuristic Distance

In the *Agent Distance* (AD) strategy, the heuristic used in planning is used to answer the question of which subgoal is closer to the agent at that point in time. The heuristic estimates are taken, and the subgoal with the lower distance is chosen.

$$AD(loc, n_{Open}, n_{Salient}) = \begin{cases} n_{Open}, & \text{if } h(loc, n_{Open}) < h(loc, n_{Salient}) \\ n_{Salient}, & \text{Otherwise} \end{cases}$$

The rationale behind this strategy is that a goal that is farther away from the agent will take longer to reach, increasing the likelihood that movement towards the subgoal will be wasted effort when the subgoal again changes.

Heuristic distance is used because discovering the actual travel distance is an application of the Lowest Common Ancestor (LCA) problem. The traversal involves backtracking from the current agent location A to the lowest common ancestor C, and then forwards to the desired subgoal n_g. The total cost would correspond to:

$$TravelCost(A, n_g) = g(A) + g(n_g) - 2g(C)$$

No constant time solution exists to identify C. In the general case, finding a solution takes $O(h)$ where h is tree height[§]. Constant time queries can be devised but this requires an $O(n)$ scan of the tree, which is unfeasible with the lists (and thus the trees) in constant flux as they grow. Work on the LCA problem can be found in [HT84, AHU73, BFc00].

[§]The tree used is the closed list.

3.4 Summary

In this Chapter the Salient Search Algorithm was introduced. Salient Search is a variation of Time Bounded A* that attempts an opportunistic tradeoff between planned solution quality and the realised agent path. Salient Search differs from TBA* by using a *salient list* to maintain an ordered reference to all descendents of the current subgoal. Salient Search uses this list to conduct *Salient expansion* - deliberate search from the current subgoal.

Salient Search also modifies the subgoal selection technique of TBA* with the introduction of the *salient strategy*. Strategies allow for dynamic control of the selection of a subgoal. Doing so can result in an agent executing a different travel path. In conjunction with salient expansion, the order of search is influenced in a purposeful fashion.

Chapter 4

Experimental Setup and Analysis

The following chapter will provide an empirical analysis of the Salient Search algorithm.

4.1 Problem Domain

All testing is done in the domain of two-dimensional grids. For real-time algorithms, grid-based testing has been used since at least early work on LRTA* [IK91], and remains a popular choice for testing pathfinding algorithms in recent literature [BBS09, BSLY07, Stu07, ZSH+09, SYCK09].

In this thesis, we used a collection of grid-based maps taken from a collection of video games. These are taken from a larger collection of maps taken from the Hierarchical Open Graph (HOG) Framework [Stu10]. The appeal of using these maps is they are sourced from several real successful computer games, adding practical appreciation to the results. This set is also utilised in other literature [BSLY07, BB09, BEHS05], including Time Bounded A* [BBS09].

From the set of maps found in HOG, a total of 50 different maps were used in the experiments, one of which can be seen in Figure 4.1. Thirty-five of the maps are taken from the Baldur's Gate collection [CSE98, CSE00], and an additional 15 from the Warcraft 3 collection [Ent02]. A complete listing of these maps along with visual

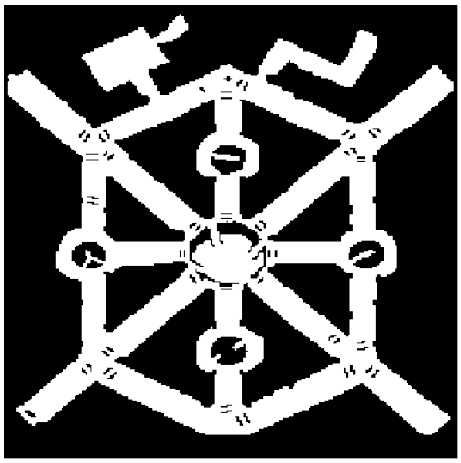


Figure 4.1: AR0701SR, an example map (derived from [Stu10, CSE00])

representations can be found in Appendix B.

All maps are scaled to 320x320 cells in size. Scaling was accomplished by taking a bitmap representation of the problem space[†] and resampling to the target dimensions via nearest neighbour interpolation. For scaling up to a larger size this resampling technique preserves the graph homomorphism. The cells themselves are all in one of two states: either freely traversable by an agent or blocked.

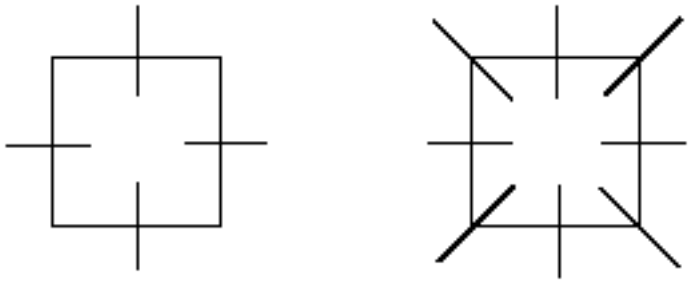


Figure 4.2: Different grid types. Left: The Cardinal grid. Right: The Octile grid

The maps themselves merely describe blocked and unblocked spaces. Also needed is a description of legitimate moves the agent can perform, which are described by a topology. Two different grid topologies were used in this experiment: cardinal and octile.

[†]Each pixel corresponds to a single grid cell.

In a cardinal grid, there are only 4 possible moves from a given grid cell, up, down, left, and right. The octile grid is an extension of this by allowing moves in diagonal directions. These types can be seen in Figure 4.2. On the cardinal grid, all moves incur a cost of 1. On octile grids, this is the same for cardinal moves. However, diagonal moves are assigned a cost of $\sqrt{2}$[‡].

It is worth testing with differing topologies because each produces a different effective branching factor, which directly influences the rate of growth of the search space. This in turn can influence the effectiveness of some pathfinding algorithms, with some algorithms outperforming others for a given topology. Further reading on this can be found in [Yap02, BEH+03].

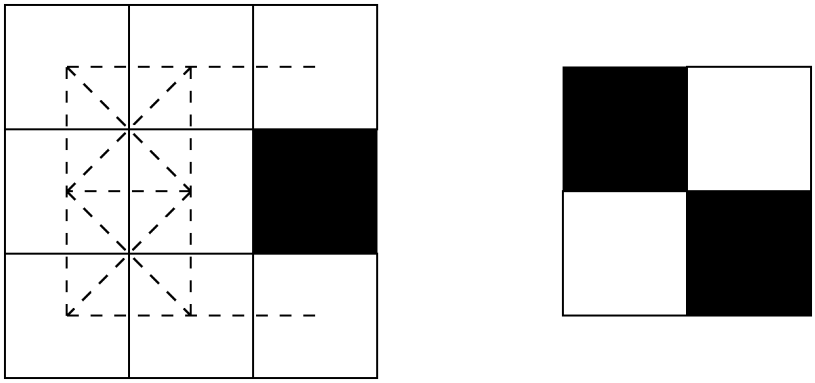


Figure 4.3: An illustration of corner cutting. The dashed lines on the left show valid moves on the grid. An agent cannot move diagonally if doing so would cross a corner of the grid shared by a blocked cell. If corner cutting is allowed, an agent could move 'through' the blocked cells in the example on the right.

Diagonal moves have an additional constraint. They are only permitted if the neighbour(s) common to both cells of the move are unblocked. This situation is commonly known as corner cutting. This also means that a diagonal move is only allowed if moving to the node can be made with two sets of cardinal moves. See Figure 4.3. While there are computer games that permit corner cutting for their agents (e.g. [GG05]), other literature tends to forbid it, so we have done the same.

[‡]In code, this is approximated to 1.42.

4.1.1 Assumptions

There are several assumptions in the testing environment that should be noted:

1. Both Time-Bounded A* and Salient Search are *episodic*. There are no trials or convergence process as with the LRTA*-based approaches; subsequent runs of either algorithm on the same problem do not influence performance, as no information is preserved once the algorithm completes. Thus there is no utilisation of any database of partial or complete solutions for any search instance.

2. The problem environment is static[§], not dynamic. Cells do not transition between a blocked or unblocked state. Similarly, edges between grid cells are not added or removed, and the cost incurred moving between cells does not change.

3. Each search is *single-agent*. Multiple agents are not conducting search simultaneously on a common map.

4. All maps keep the *safety assumption*. From [BBS09]: "The goal state can be reached from any state reachable from the start state". In fact, the maps used have been modified where necessary to prevent any 'islands', so every unblocked state is reachable by every other unblocked state. This was verified by running a breadth-first search with duplicate detection over each map and checking that every unblocked cell was visited by the search.

5. There is no *freespace* assumption as it appears in algorithms like RTAA*. Node expansion involves the true state and cost on the map, not open space beyond a search horizon perceived by an agent. Rather, search proceeds with perfect information of the map.

[§]Technically, Time Bounded A* and Salient Search's agent can be said to inhabit a *semidynamic environment*. Since the agent is potentially incurring a nonzero travel cost with every step before the solution is found, the passage of time with each iteration is antagonistic to a performance score (the travel ratio). The concept is discussed in [RN03].

4.1.2 Search Pair Generation

For each of the 50 maps, start and location pairs were generated. Classic A* was run between these points to determine an optimal path between these points, and the pair was kept if the path found by the search had a cost between 50 and 500. This was repeated until 100 pairs were generated for each map, for a total of 5000 problems. A particular coordinate could be used as a start or goal location in multiple problems, and there was no prohibition against a particular pair used in reverse. This process was repeated separately for each of the grid topologies.

4.1.3 Heuristics Used

A separate heuristic was used for each of the grid topologies. Each heuristic is considered *perfect* in an unblocked grid of its type. With no obstacles, the heuristics used give the true cost h^* between the start and goal locations.

Cardinal Maps

The Manhattan heuristic[¶] is the sum of the distance between two points along the axis of each dimension. In a two dimensional grid world it is simply:

$$Manhattan : \Delta x + \Delta y$$

Octile Maps

For octile grids the heuristic of the same name is used: a cost of 1 for NSEW moves, and $\sqrt{2}$[‖] for diagonal moves. Because the agent can move in both x and y dimensions in a single step, the heuristic is the minimum number of diagonal steps to finish moving in one dimension, plus the remaining number of steps to complete travel in

[¶]May also be referred to as the *Cardinal* or *Taxi-cab* or *City block* distance.
[‖]In code, this is approximated to 1.42.

the remaining dimension. In equation form, this is:

$$Octile : \min(\Delta x, \Delta y) \times \sqrt{2} + |\Delta x - \Delta y|$$

For the given topology, the same heuristic was used regardless of the particular algorithm (A*, TBA*, or SS) being run.

4.1.4 Parameters

As described in [BBS09], the parameter N_E is defined as a ratio of the resource limit R by $N_E = \lfloor R \times r \rfloor$, where $r \in [0, 1]$. The authors elected to fix r at 0.9, and we have done the same for our experiment. As well, the parameter c is fixed at a value of 10. The values of the resource limit (R limit) used in the experiments were R = {25, 50, 100, 500, 1000}.

Table 4.1: N_S expansion by percentage of N_E, $r = 0.9$

	R Limit				
%	25	50	100	500	1000
30	7	13	27	135	270
50	11	22	45	225	450
75	16	34	67	338	675

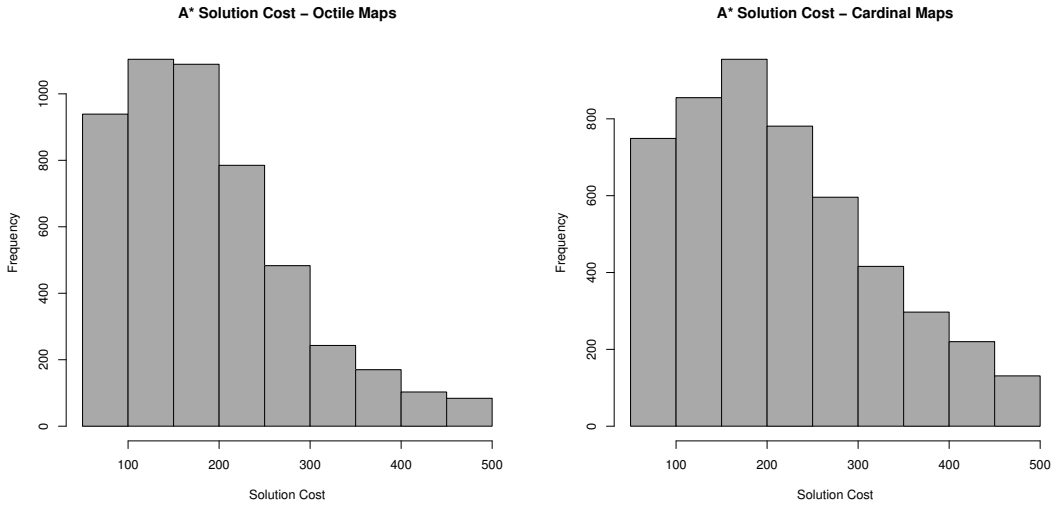
N_S, the parameter used to control the number of expansions inside the salient, took on values corresponding to 30%, 50% and 75% of the value of N_E. The number of expansions this corresponds to for each permutation can be found in Table 4.1. For example, With an R limit of 50, there will be 45 expansions in an iteration, and if 30% should be directed towards the salient, N_S takes on a value of 13.

For Salient Search's *strategy* parameter, both the Agent Distance (AD) and Tie-Breaking (TB) strategies described in 3.3 were tested for each N_S/R limit combination.

Thus, with 5000 different searches across the 50 maps, there are 25000 datapoints created using TBA* (5000 problems × 5 R Limits), and 150000 using Salient Search (5000 problems × 5 R Limits × 3 Salient expansion settings × 2 strategies).

4.2 Results

All experimental results were collected using the same hardware and environment for both TBA* and Salient Search. The hardware was an Intel®Core™i5 750 CPU at 2.67GHz with 4GB of RAM installed. The operating environment was the 64bit edition of Windows 7 Professional. All test code including the algorithms under test were implemented in the C# language, targeting version 3.5 of the .NET Common Language Runtime. Figures and tables regarding the results and statistical analysis of the experiments were created using the R programming language environment [R D10] using the Rcmdr package [Fox05].



(a) (b)

Figure 4.4: Histogram of A* costs for different map styles

As mentioned in Section 4.1.2, classical A* was used to determine the optimum cost for each start/goal pair. Figure 4.4 shows a histogram of the optimum costs

across the entire problem set. It is apparent from this histogram that the cost of the optimum paths are not uniform, but distributed around a mean cost between 150 and 200. The histogram for the cardinal grid type appears flatter, with thousands more paths with a cost > 350. This can be attributed to the lower branching factor of cardinal maps. This is not unexpected; it has been shown previously that a larger branch factor can lower the search depth [BEH+03].

Alongside the distribution of long and short paths, it is useful to note the relative difficulty each path presents. This can be done by recording the *heuristic error*. Using heuristic error as a difficulty measure is described in [MK10] as the difference between the heuristic and real cost for some search at the start. If $h^*(S)$ is the true cost to travel from S to G then the heuristic error is:

$$h^*(S) - h^0(S)$$

The sum of all n^2 search pairs for a map the *total initial heuristic error*, and [MK10] uses this as a difficulty metric for comparing two maps. We have used this concept to measure the difficulty of our problem set based on heuristic error for the beginning of search.

Figure 4.5 is a scatterplot of heuristic error for every search in the experiment, broken down by topology. The x-axis represents the range of least-cost paths found through A* search, whereas the y-axis plots the difference between the cost and the heuristic value for the beginning of the search at S_0. The boxplots along the margins show the inter-quartile ranges.

From this plot, it is seen that the octile set has a lower mean cost, but longer paths are more prone to heuristic error on these maps. Note on the octile plot that all but two paths with a cost above 400 have heuristic error greater than 100. In contrast to this, the cardinal collection has on average higher cost paths, but much less error, with some initial heuristics having no error at all. A thick collection of

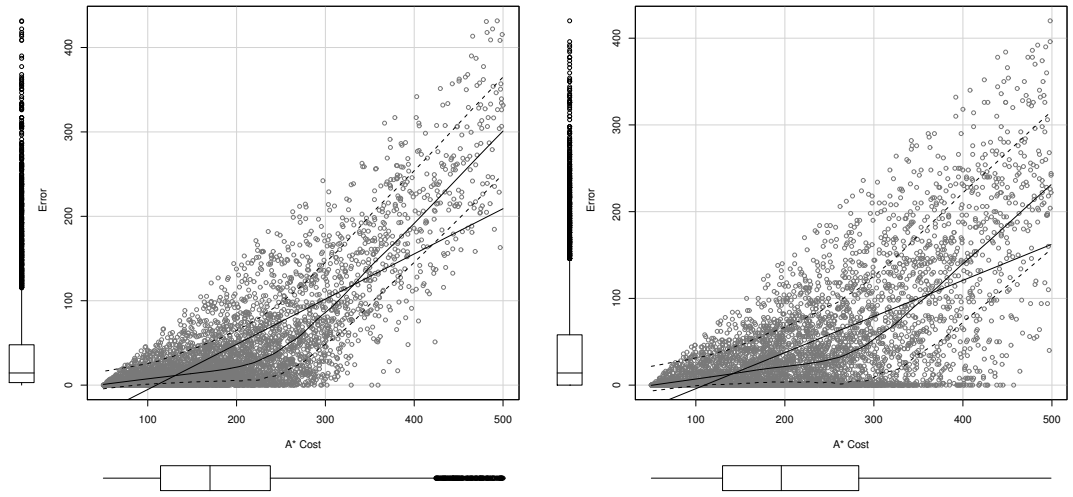


Figure 4.5: Scatterplot of heuristic error for different map styles. Scatterplot (a) plots octile measures. (b) shows cardinal measures.

points with no error can be seen along the bottom of the plot, almost all the way along to the cost ceiling. As well, there is only a single point on the cardinal plot with error > 400, whereas the octile plot has 8 over 400.

This hardness measure can be adapted to isolate the performance of Salient Search against TBA* to a subset of problems considered difficult. Not computing all the values for some map, we instead calculate the *heuristic error ratio* for our subset of 100 searches on each map. For a given start/goal pair P_u, this error ratio is defined as:

$$1 - \frac{h_u}{h_u^*}$$

This produces a mean error score for each of the 50 maps. From this calculation, we selected several maps with the highest mean to be representative of 'hard' problems in our sample set. Analysis specific to this set can be found in Section 4.4.

4.3 Analysis of Results

4.3.1 Path Quality

Since Salient Search admits the possibility of finishing with a suboptimal path, an immediate question is: *Does Salient Search produce more costly paths? If so, how much worse are they?*

There does not appear to be a strong indication that Salient Search returns poorer quality paths. Looking at the raw data on the octile set of searches, 46.72% of the paths were optimal; there was no reduction in path quality, with Salient Search match A* for cost using the same heuristic. The 5-number summary (*minimum*, $Q1$, *median*, $Q2$, *maximum*) of the suboptimality of path quality for Salient Search is (1.000, 1.000, 1.003, 1.015, 1.528). This shows that at least 75% of the problems had a solution within 2% of the optimum value. With single steps on these maps costing at most 1.42, for a path of optimum cost 100 this is no worse than one or two steps divergence from the optimum. In a video game this could be easily unnoticed by a player.

Further, only 1.08% of the paths returned were more than 10% longer than the optimum[††]. For this small percentage of the total, most of the paths were found when a higher R limit was used (646 and 804 paths for R = {500, 1000} respectively for 89.8% of the total). Larger values for the N_S parameter also played a factor, with 1028 or 63.7% of the longest paths appearing when N_S was set to apply 75% of expansions inside the salient.

These results seem to indicate that only in the extreme circumstances - high resources for search in conjunction with a great emphasis on forcing expansion inside the salient - does the Salient Search algorithm produce poorer paths. With smaller time slices preferred anyways, we can conclude that in general Salient Search produces quality paths.

[††]1614 of 150000 paths

4.3.2 Suboptimality of Travel

More important than the solution path either algorithm ends up with is the actual path followed by the agent over the course of the search. This section provides an analysis of the travel path compared to the optimal.

Table 4.2: Average Expansions and Travel Ratio of TBA*

R	Exp./Move	Suboptimality
25	10.98042	1.497127
50	13.66260	1.232140
100	15.86997	1.094670
500	17.81902	1.013893
1000	18.06391	1.006509

Table 4.3: Average Expansions and Travel Ratio of Salient Search by strategy

AD				TB			
R	N_S%	Exp./Move	Suboptimality	R	N_S%	Exp./Move	Suboptimality
25	30	10.91907	1.541986	25	30	11.10009	1.519612
	50	10.97588	1.572389		50	11.15048	1.540335
	75	11.11479	1.597097		75	11.21977	1.571833
50	30	13.73711	1.250812	50	30	13.80344	1.245030
	50	13.82555	1.262895		50	13.84636	1.256882
	75	14.02849	1.276260		75	13.93842	1.267801
100	30	15.94294	1.103526	100	30	15.96143	1.104827
	50	15.96384	1.110234		50	15.91587	1.108191
	75	16.03795	1.114515		75	15.91056	1.112261
500	30	17.60272	1.014343	500	30	17.59812	1.014265
	50	17.48617	1.014317		50	17.44961	1.014366
	75	17.43319	1.014873		75	17.33247	1.014253
1000	30	17.73254	1.006207	1000	30	17.71119	1.006334
	50	17.58462	1.006269		50	17.51687	1.006247
	75	17.70115	1.006541		75	17.54741	1.006289

Table 4.2 is a table of run time efficiency for TBA*. Each row is the mean value for the set of searches performed for the given R value in the left column. The second

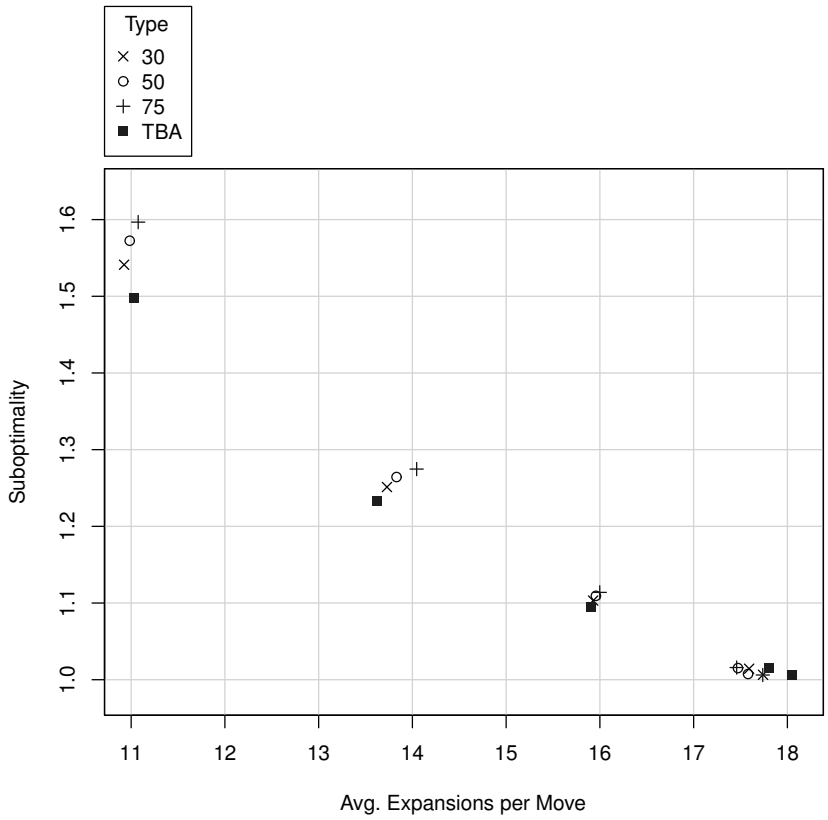


Figure 4.6: Relative quality of agent's travel paths for Salient Search, using the AD strategy.

column is the mean number of nodes expanded for every step the agent takes until it arrives at the goal and the algorithm terminates. The final column gives the relative quality of the path the agent travelled against the path found from start to goal. Put another way, it is the *suboptimality* ratio between *travel cost* and *solution cost*. For instance, if a path is found between start and goal that has a cost of 100, and the sum total of the costs for every move the agent makes to reach the goal is 145, the suboptimality is 1.45.

Table 4.3 shows the same data, but for Salient Search. This table is broken down further to show the mean values across the extra parameters used for Salient Search. The left side of the table is the data using the Agent Distance strategy, with the

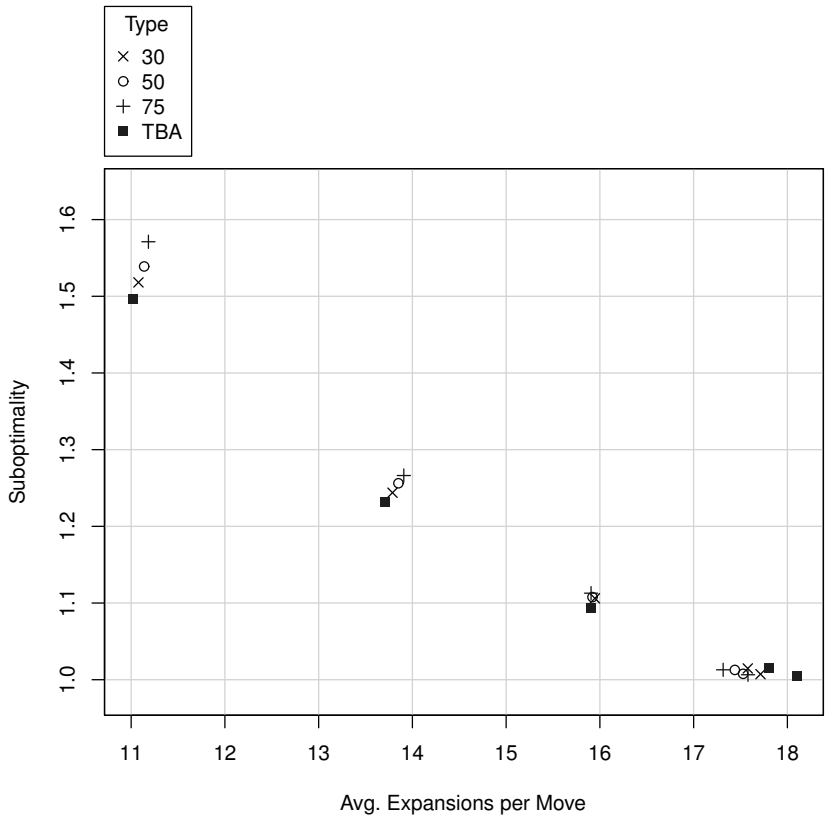


Figure 4.7: Relative quality of agent's travel path for Salient Search, using the TB strategy.

right side showing results for the Tie-Break strategy. Each R limit is broken down by the three different settings employed for N_S, expressed here as a percentage of N_E (Refer back to Table 4.1 for the precise values corresponding to each R/N_S pair). In Figures 4.6 and 4.7 the values are plotted against TBA* for comparison. The values for TBA* are the same in each graph. Each point on the graph is the average of a run of 5000 searches: 100 searches on each of the 50 maps. Points closer to the origin signify better performance.

The strongest trend obvious in this data is the effect of how much work is allowed per iteration. Distinct clusters can be seen in the data, as each corresponds to a different R value. The leftmost values are R = 25, and the values in the lower right

correspond to R = 500 and 1000. Each group of points produces a downward sloping curve.

This is not unexpected; the effect of increasing the resource limit is intuitive. More expansions are performed for every iteration. This leads to a solution being discovered in fewer iterations. Consequently, the agent spends less time wandering between different paths, and travel quality improves.

For each strategy, increasing the amount work focused on salient expansion appears to have a negative effect on suboptimality. For instance, for $R = 25$ and using the AD strategy, travel quality goes from 1.531 to 1.572 to 1.597 as N_S is increased, compared to the suboptimality of TBA* of 1.497. This trend in quality is reflected across all parameter values, though the effect rapidly diminishes with larger values of R, with all variations approaching optimal path quality.

For expansions per move the trend is a little more complex. With small R values the work average has some variance, again with larger N_S producing worse numbers. But from $R = 500$ onward, Salient Search produces lower values relative to TBA*. For example, with $R = 500$, TBA* produces a mean of $\tilde{1}7.819$ moves per agent action, where Salient Search varies from 17.332 to 17.603. Suboptimality scores are marginally improved at high R limits, but this could be noise.

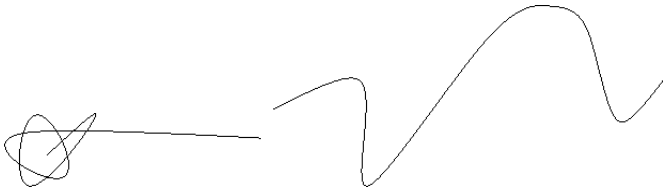
Based on these results, the conclusion is that for the strategies tested Salient Search incurs a penalty to total travel cost within a few percent of TBA*, but given the cost constraints involved, the added travel cost is minimal. For small resource limits agents perform extra moves, but the difference rapidly diminishes as R is increased.

A low level of salient expansion seems to offer some benefit over none at all as in TBA*, but there is a point where the effect becomes deleterious to the search effort.

4.3.3 Back-stepping Behaviour

It is not just the path followed that is of interest when studying these algorithms, but also how precisely the path is being followed by the agent. The travel ratio was one

such consideration. But that metric only compares the length of the resulting path. There are many ways an agent can move about to produce a travel path of a certain length.



(a) (b)

Figure 4.8: Two travel paths with similar travel ratios but very different behaviour.

For example, imagine the two lines in Figure 4.8 are the paths followed by an agent in two separate searches, with the travel ratio of both being the same. One path involved the agent oscillating around the start point, then moving towards the goal in a relatively straight line. The other is a more leisurely path that generally speaking is always making progress towards a goal, if slowly. Both paths have the same value in length, but can be said to produce vastly different behavioural characteristics for the agent. It is interesting then to determine the behavioural characteristics of both TBA* and Salient Search by examining the related statistics.

Backtracking

In TBA* and Salient Search, any move not made by the agent following some path is a step backwards towards the starting node (line 25 in the respective pseudocodes, Algorithms 4 and 5).

Table 4.4 shows the mean and deviation for the number of backwards steps for Salient Search and TBA*. For each cell, the top value is the mean, and the deviation is the value enclosed in brackets.

These numbers indicate that an agent using TBA* will on average take fewer steps backwards than an agent using Salient Search. As would be expected, small R limits result in a much greater incidence of backtracking, with more subgoals chosen

Table 4.4: Mean backwards steps for agents in SS, TBA*

Algo		R Limit				
	N_S%	25	50	100	500	1000
SS_{AD}	30	52.4532 (72.71234)	25.1072 (42.82833)	10.6588 (22.21770)	1.5562 (4.307221)	0.6730 (1.916352)
	50	55.3902 (73.89290)	26.3782 (44.01201)	11.3808 (23.39013)	1.5562 (4.111912)	0.6852 (1.970197)
	75	58.3542 (74.60884)	27.8424 (44.90504)	11.9184 (24.09462)	1.5988 (4.209155)	0.7136 (2.046561)
SS_{TB}	30	50.2070 (70.17360)	24.5968 (42.47888)	10.9150 (23.67314)	1.5582 (4.413265)	0.6894 (2.041314)
	50	52.4502 (71.04922)	25.7856 (42.99851)	11.3106 (24.19639)	1.5762 (4.386575)	0.6870 (2.017287)
	75	55.8340 (72.13635)	27.2578 (45.07297)	11.7648 (24.34368)	1.5628 (4.379348)	0.6874 (1.963480)
TBA*		48.1688 (69.9099)	23.3884 (41.7811)	9.8780 (22.5055)	1.4724 (4.1774)	0.6892 (1.9863)

Table 4.5: ANOVA for backwards steps

	Sum Sq	Df	F value	Pr(>F)
N_S%	91588	2	29.5717	1.444e-13 ***
R Limit	59464143	4	9599.8536	< 2.2e-16 ***
Strategy	14675	1	9.4767	0.002081 **
N_S% : R Limit	117166	8	9.4576	3.667e-13 ***
N_S% : Strategy	330	2	0.1065	0.898985
R Limit : Strategy	37190	4	6.0039	7.937e-05 ***
N_S% : R Limit : Strategy	532	8	0.0429	0.999968
Residuals	232238895	149970		

before the goal is found, and an agent therefore moving backwards towards the path to follow at that time. As the R limit is increased, there are fewer iterations and therefore fewer subgoals to move between. At the highest R limit a solution is found quickly, and most movement is forward towards the goal. The level of N_S appears to correlate with the mean as well. With each strategy, the mean climbs with the level of salient expansion.

To determine if Salient Search's new parameters are in fact influencing the number of back steps, a multi-way analysis of variance was performed. Table 4.5 shows the results of an analysis of variance (ANOVA) for Salient Search with the number of backwards steps as the response. The table can be read as follows: the first column denotes the factors examined - the level of N_S, the resource limit used, and the particular Salient Strategy used. Interaction effects between factors are denoted with the associated factors separated by a colon (e.g. The row with "N_S% : R Limit" is the interaction between those two factors). The second column is the sum of squares. The third column is the degree(s) of freedom for the given factor(s). F value is the value of the F-test used for the factor and the final column is the calculated p-value.

ANOVA appears to confirm the significance of Salient Search's parameters in influencing the rate of backtracking. The amount of salient expansion is significant to the $p < 0.001$ level. The R limit is as well, but this was expected. The two strategies are significant to the $p < 0.01$ level. Also highly significant is the interaction between the resource limit and the other treatments, each to the $p < 0.001$ level.

Based on the higher observed means and confirmed significance of parameters, we can conclude that Salient Search incurs a penalty on the number of backwards steps taken. As expected, the rate of backtracking drops rapidly with the increasing R limit. This result can explain the slightly higher travel ratio, since backtracking would generally increase the distance between the agent and the goal instead of reducing it.

Changes in Direction

A related statistic is how many times an agent reverses direction; moving forward along a path when the path changes and backtracking is now required. The converse is also relevant, when an agent stop backtracking and instead moves forward.

To clarify, change in direction is not described in terms of agent moves within the problem space, but the paths themselves, whether forward or backward moves in the context of paths. Figure 4.9 represents two paths A and B originating at S

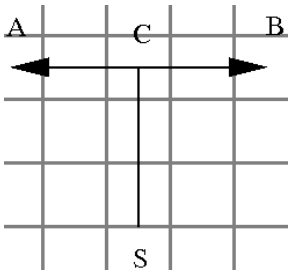


Figure 4.9: A forking path.

and forking at C. Suppose an agent was following A and past C when the algorithm switches the path to follow to B. The agent will change direction and backtrack. If B is still the path to follow when the agent reaches C, then it will begin to take steps forward along B. This is the second change in direction, even though the headings of all moves the agent makes is to the right.

Table 4.6 lists the figures for this statistic. Table 4.7 is the ANOVA with the number of direction changes as the response. A mean value of 5 indicated the agent on average reversed direction from forward to back or vice versa 5 times before reaching the goal.

The table of means illustrates a marked performance change between the two strategies. TB has a slightly lower value at most R limits as opposed to the AD strategy, as well as a smaller deviation. As expected, this statistic also drops quickly with increases to the R limit.

The analysis confirms this observation. All three treatments are significant to the $p < 0.001$ level, so there is an effect on the number of direction changes. There is also an interaction between the resource limit and each of the other treatments, both to the $p < 0.001$ level.

When considering the results of these two statistics, it appears the following behaviour is occurring: When choosing interim subgoals, Salient Search exerts a slight pressure on choosing subgoals that for the most part will produce a path that is an extension of the previous path. This was the intended effect of the TB strategy. If

Table 4.6: Mean changes in direction of travel for agents in SS, TBA*

Algo		R Limit				
	N_S%	25	50	100	500	1000
SS_{AD}	30	5.0252 (4.8648)	2.9752 (3.3072)	1.9404 (2.3759)	0.9000 (1.5865)	0.6024 (1.2331)
	50	5.1728 (4.8953)	3.0252 (3.1761)	2.0356 (2.4962)	0.8952 (1.5917)	0.6028 (1.2449)
	75	5.3244 (4.7666)	3.1392 (3.1384)	2.0604 (2.4406)	0.8888 (1.5968)	0.5964 (1.2569)
SS_{TB}	30	4.5644 (4.5428)	2.7372 (3.0852)	1.8512 (2.3627)	0.8872 (1.5836)	0.5940 (1.2199)
	50	4.6444 (4.5286)	2.8112 (3.0670)	1.9444 (2.4078)	0.8784 (1.5753)	0.5964 (1.2428)
	75	4.8244 (4.4458)	2.8720 (3.0289)	1.9668 (2.4385)	0.8484 (1.5636)	0.5812 (1.2545)
TBA*		4.7356 (4.920380)	2.7480 (3.264265)	1.8236 (2.496343)	0.8928 (1.574358)	0.6184 (1.227796)

Table 4.7: ANOVA for changes in direction of travel

	Sum Sq	Df	F value	Pr(>F)
N_S%	263	2	15.7788	1.406e-07 ***
R Limit	369132	4	11087.9545	< 2.2e-16 ***
Strategy	1111	1	133.5445	< 2.2e-16 ***
N_S% : R Limit	327	8	4.9133	4.320e-06 ***
N_S% : Strategy	3	2	0.1733	0.8409
R Limit : Strategy	1235	4	37.0990	< 2.2e-16 ***
N_S% : R Limit : Strategy	8	8	0.1152	0.9987
Residuals	1248171	149970		

the agent was on and following the previous path when the new one is produced, no backtracking is necessary to switch over, and the agent will continue moving forward without interruption. No direction change, no backtracking. However, if there is a bigger shift (if the new path is not an extension at all, or the new path branches off the current one 'behind' the agent), the agent will have more ground to recover, reversing direction and having that much further to walk backwards.

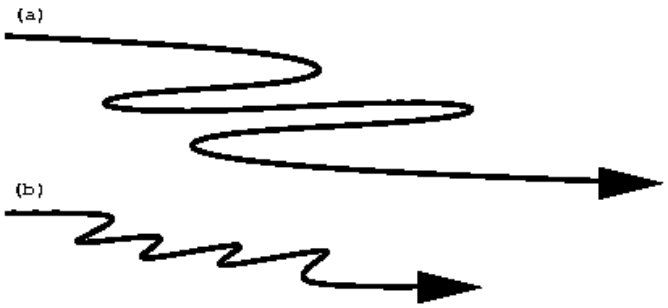


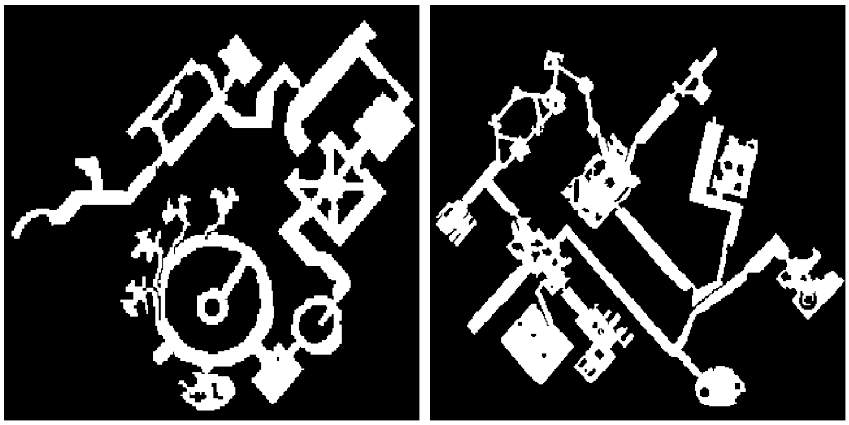
Figure 4.10: An illustration of backtracking behaviour for both algorithms. Salient Search (a) changes direction less, but incurs a larger movement penalty for doing so. TBA* (b) wastes less time on alternative paths at the expense of more changes.

Direction changes must occur in pairs - when the agent changes direction to move backwards it will necessarily change direction again to follow a path - this backtracking/direction change for Salient search is likely caused by an agent following a path for a longer period of time before a newer path branches off. Figure 4.10 illustrates the idea. This can be seen described as a characteristic of 'stubbornness', where an agent is prone to following a path longer than would occur with TBA*. If the path forced by the salient doesn't produce a solution, the agent would then have to spend additional steps backtracking.

4.4 Performance on Harder Problems

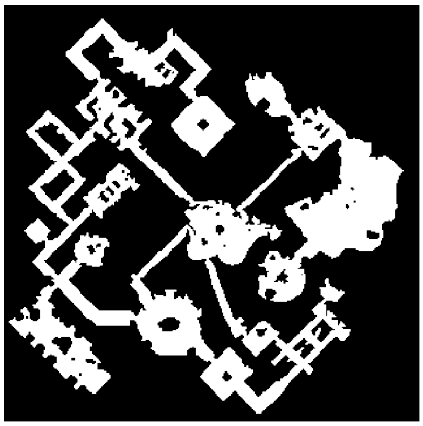
In Section 4.2 we described the *heuristic error ratio* as a means of describing the relative difficulty of searching on a map. Comparing a sample of errors for two maps allows a comparison of difficulty. This information was compiled and from this list several maps were identified as being more difficult. This section is analysis of the algorithms limited to the set of searches done on these maps.

The particular maps are AR0202SR, AR0307SR, AR0602SR and AR0705SR, for 400 searches. Figure 4.11 is an illustration of these maps. A complete table of the relative heuristic error for all 50 of the maps can be found in Table A.1. Characteristic

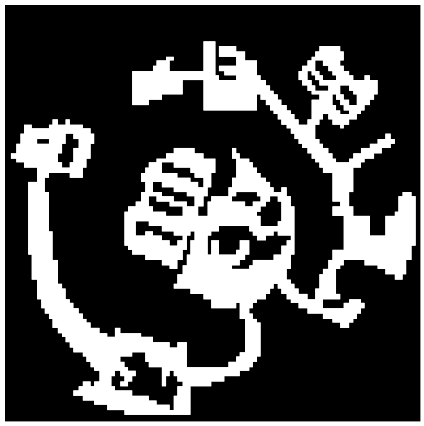


(a) AR0202SR

(b) AR0307SR



(c) AR0602SR



(d) AR0705SR

Figure 4.11: Four maps featuring harder search. The set of problems for each features high heuristic error.

to these maps are plenty of long corridors and little open space.

Note that not every one of these 400 problems in this set featured a large heuristic error. Given the random nature of creating the problems, some in fact have no heuristic error at all, being trivial, straight line paths. Rather, the 100 searches for these maps as a set had some of the worst mean heuristic error.

Unsurprisingly, the paths themselves in this set are of higher cost than the general population. Optimum cost for the harder maps has a median of 272.3, with an IQR of 199.2, compared to 170.1 and 123.6 respectively for the full set of problems.

Path Quality on Harder Maps

In Section 4.3.1 the cost of the paths returned by Salient Search was compared to the cost of the path found by classical A*. Similar to the results found there, the low incidence of poorer quality paths persists in the subset of harder maps. Only 1.54%[‡‡] of the paths have a solution cost that is more than 10% over the optimum cost. Thus, a comparison of the travel ratios for TBA* and Salient Search remains meaningful for the harder problems.

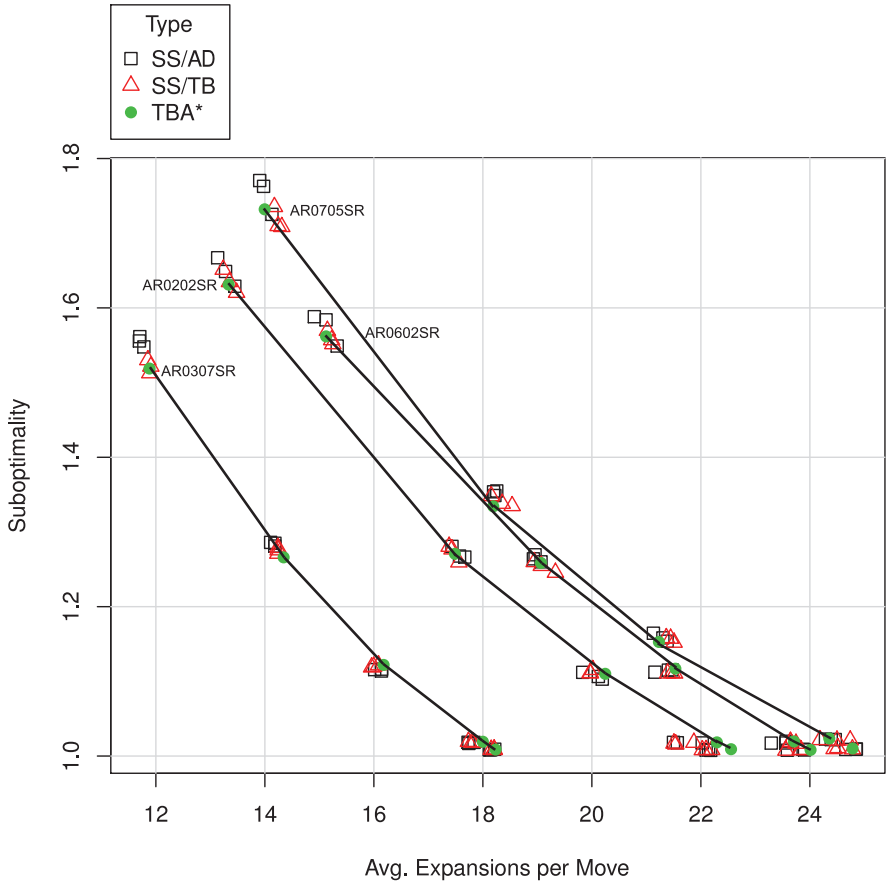


Figure 4.12: Performance of Salient Search against TBA* on the harder maps

Figure 4.12 shows the performance of Salient Search against TBA* these four

[‡‡]185 of 12000 paths

maps. This is the same type of graph found in Figures 4.6 and 4.7. For clarity, the lines are drawn to demonstrate the grouping of point clusters for each map.

Compared to the results in Section 4.3.2, the performance profile appears similar. Salient Search appears to be producing a sharper slope of values, dropping below the trend of TBA* as R increases. One notable difference from the general results are the values for average expansions for Salient Search at the lower values for R. Referring back to Figures 4.6 and 4.7, Salient Search's values were firmly to the right of the equivalent point of TBA*. However on this plot the values for Salient Search are more in line with the TBA* plot, appearing to adhere to the curve plotted by the TBA* points.

Looking between the two groups of Salient Search points, it appears that there is a tradeoff between the two strategies. Especially at the lower values for R, the AD strategy has a slightly lower move average, coming at a slightly higher travel ratio.

Backtracking on Harder Maps

With a high heuristic error suggesting that the expansion of the search space will spread out in a somewhat inefficient fashion, there is an expectation that successive paths produced by TBA* can vary wildly in direction. This would necessitate more backtracking by the agent as it often finds the path it is following to be replaced by one without a common element. This is apparent for the harder maps.

Table 4.8 shows the mean and deviation for the number of backwards steps for Salient Search and TBA* on the harder maps.

One somewhat surprising result here is the lack of a difference in the scores for the different strategies. There is one curious value (86.2 for SS_{AD}, $R = 25$, $\mathrm{N}_S\% = 75$), but otherwise no significant difference. There does appear to be a trend of TBA* having an advantage at lower R limits, with the scores converging to 0 as R grows.

Compared to the general set of searches however, the difference between TBA* and Salient Search is smaller. In the general set, mean backwards moves varied from

Table 4.8: Mean backwards steps for agents in SS, TBA* on harder maps

Algo		R Limit				
	N_S%	25	50	100	500	1000
SS_{AD}	30	80.5100 (78.64998)	38.5375 (46.97932)	16.3075 (23.00698)	2.7050 (4.626715)	1.215 (2.066264)
	50	83.3000 (80.85694)	39.1200 (48.88163)	16.7950 (24.08533)	2.6525 (4.585773)	1.200 (2.099051)
	75	86.2325 (83.49975)	39.7825 (48.99123)	17.3750 (24.31012)	2.6950 (4.506411)	1.250 (2.230457)
SS_{TB}	30	79.5700 (79.75021)	38.0150 (48.03069)	16.9525 (25.03262)	2.7200 (4.680424)	1.3000 (2.343872)
	50	81.6825 (83.33173)	37.2725 (46.93480)	16.9525 (25.13981)	2.6275 (4.612150)	1.2025 (2.162912)
	75	80.7450 (78.60255)	39.7675 (50.43692)	17.1650 (25.52580)	2.6650 (4.750177)	1.2375 (2.286737)
TBA*		79.5550 (82.215268)	37.3825 (48.136566)	16.5875 (24.983049)	2.7100 (4.671237)	1.2675 (2.235719)

48.16 to 58.35 at R = 25, whereas in the harder set it varies from 79.55 to 86.23, a smaller range in both relative and absolute terms.

Table 4.9: ANOVA for backwards steps on harder maps

	Sum Sq	Df	F value	Pr(>F)
N_S%	2465	2	0.6487	0.5227
R Limit	10828947	4	1424.9371	<2e-16 ***
Strategy	1281	1	0.6743	0.4116
N_S% : R Limit	4010	8	0.2638	0.9774
N_S% : Strategy	508	2	0.1336	0.8749
R Limit : Strategy	3437	4	0.4522	0.7709
N_S% : R Limit : Strategy	2333	8	0.1535	0.9964
Residuals	22741793	11970		

Table 4.9 is the analysis of variance for this data. Here, it is seen that the features of Salient Search failed to influence the variable. The only significant parameter is the R Limit, to the $p < 0.001$ level.

Changes in Direction on Harder Maps

Table 4.10 shows the mean and deviation of the number of times an agent changed direction on the hard maps for TBA* and Salient Search. It would be expected that more difficult searches would increase the incidence of direction changes, and the results indicate this.

Table 4.10: Mean changes in direction of travel for SS, TBA* on harder maps

Algo		R Limit				
	N_S%	25	50	100	500	1000
SS_{AD}	30	7.685 (5.057145)	4.455 (3.278696)	3.115 (2.779585)	1.66 (2.148474)	1.170 (1.548255)
	50	7.830 (5.073011)	4.355 (3.106138)	3.100 (2.861901)	1.66 (2.176291)	1.145 (1.557253)
	75	7.905 (4.741009)	4.425 (3.139110)	3.105 (2.772760)	1.67 (2.200729)	1.135 (1.608803)
SS_{TB}	30	7.100 (4.832015)	4.240 (3.219988)	2.865 (2.588150)	1.565 (2.066750)	1.185 (1.640886)
	50	7.430 (4.815920)	4.035 (3.119793)	2.950 (2.652374)	1.580 (2.003406)	1.140 (1.589565)
	75	7.305 (4.654152)	4.265 (3.185143)	3.050 (2.915261)	1.555 (2.035272)	1.105 (1.573265)
TBA*		7.73 (5.156897)	4.39 (3.417176)	3.01 (2.774734)	1.64 (1.969797)	1.19 (1.609464)

These results seem to indicate that Salient Search enjoys an advantage over TBA* when it comes to the number of changes in direction. The means, while only marginally lower, are nonetheless lower for the majority of test cases.

The Tie-Breaking strategy in particular appears to have a positive influence here: for the smallest R levels, this strategy appears to have 10% fewer direction changes (7.100 against 7.73), along with a smaller deviation. The effect diminishes as the R limit grows, but this can be attributed to the practical bound on map/problem size resulting in a solution in fewer cycles.

This is in line with the performance discussed in Section 4.3.3. In fact, the larger

difference between the two algorithms on these harder maps is a suggestion that Salient Search is more effective at controlling this behaviour when a difficult search problem is applied.

Table 4.11: ANOVA for changes in direction of travel on harder maps

	Sum Sq	Df	F value	Pr(>F)
$N_S\%$	5	2	0.2425	0.7846889
R Limit	63080	4	1631.0757	< 2.2e-16 ***
Strategy	124	1	12.7867	0.0003505 ***
$N_S\%$: R Limit	39	8	0.5073	0.8517583
$N_S\%$: Strategy	1	2	0.0411	0.9597703
R Limit : Strategy	95	4	2.4692	0.0426159 *
$N_S\%$: R Limit : Strategy	11	8	0.1415	0.9972688
Residuals	115731	11970		

Surprisingly, ANOVA in Table 4.11 indicates that different levels of salient expansion are not significant at affecting the rate of direction change. Referring back to Table 4.7, N_S is highly significant with a positive correlation between the two values, whereas here it is not.

Strategy however, plays a larger role, being significant to the $p < 0.001$ level.

4.5 Summary

In this chapter we described and presented the results of an experiment to determine the performance characteristics of Salient Search. This experiment involved 5000 distinct pathfinding problems taken from 50 maps. These maps are derived from several successful computer games. The experiment involved a total run of 150000 searches uses the Salient Search algorithm, and 25000 searches using TBA*.

The results indicate that contrary to initial expectations, Salient Search does not outperform TBA* in terms of travel ratio when small work windows are used. There does appear to be a point where the suboptimality to work ratio of Salient Search

surpasses TBA*, but only with larger planning times where both algorithms are converging to the optimum.

Beyond the cost of the solution itself, Salient Search's use of strategies for selecting subgoals appears significant in affecting some secondary statistics, which in turn produce apparent behavioural differences for an agent. The *Tie-Breaking strategy*, which for equally scored alternatives prefers the choice in the salient, appears to produce an agent that tends to change direction less. The decision to choose based on heuristic distance between the agent and subgoal was expected to have similar results, but in fact had a slight negative impact on performance.

Chapter 5

Conclusion

In this thesis we have introduced Salient Search, a real-time pathfinding algorithm derived from the TBA* algorithm. To review, TBA* is notable for being a real-time algorithm, in that it diverges from the main body of real-time search algorithms. The main body of work builds on the LRTA* approach of repeated local search to build a perfect heuristic. TBA* differs in this regard by conducting a global search effort to determine an optimal solution.

Salient Search keeps the same benefit introduced by TBA*: high-quality paths without the need for convergence trials or extensive precomputation. It differs through the introduction of two concepts:

- The *Salient*. Copying open list node references into a temporary open list, beginning with a single open list node (the salient root) allows a portion of planning effort to be directed exclusively onto a subset of the open list. This can be done without losing the amortised constant time operation necessary to claim real-time performance.

- The use of *Strategy* in choosing interim paths. This thesis introduced two strategies: a same-score tie-breaking strategy to force selection of an equally best-first node in the salient, and a strategy based on estimating the agent's

distance to the candidates. Other strategies are possible, provided the strategy function returns a choice in O(1) running time.

In this thesis we also compared Salient Search against TBA* through an extensive empirical trial, running both algorithms against a range of pathfinding problems taken from popular computer games. From this, we are able to come to two conclusions regarding the algorithm:

First, we are able to conclude that Salient Search's features are effective at influencing the progression of the search. Strategies in particular were highly significant, with one strategy outperforming the other across the statistics we were concerned with. Further, when restricting ourselves to more complex environments, Salient Search coped with the increasing difficulty of the problems better than TBA*.

The effect was strongest with the smallest planning slices, which is an antagonistic but desired setting. Smaller planning slices is a tradeoff when planning effort needs to be spread across multiple agents. Thus, Salient Search could be an effective choice where many agents are involved with planning complex routes. An example may be planning for thousands of agents in a simulated shopping mall.

Secondly, the results of the experiment showed that the choice of strategy is quite significant in the resulting behaviour an agent displays as the algorithm progresses. The tie-breaking strategy in particular was effective in lowering the incidence of direction switching, producing a 'stubbornness'. This change in behaviour on the part of the agent is significant, as it demonstrates that behavioural models can be incorporated directly into real-time algorithms. Strategies can be classified according to the movement behaviours that emerge from their use. This could be useful where 'believable' supercedes the need for an optimal path. An obvious choice is in video games, where constant optimal behaviour on the part of an agent can be seen as 'cheating'.

5.1 Future work

Further investigation can be done in finding an appropriate value for the expansion parameter N_S. The results of our empirical study indicate that lower levels produce result that may be desired, but there is a crossing point where the effect becomes antagonistic. This would suggest that there is some non-zero value for N_S yielding maximum benefit, depending on other factors like the branching factor.

Additional work could be done to look at a strategy-only approach, where expansion is left to occur exactly as A* or TBA* (i.e. $N_S = 0$) and looking at the effect of strategy alone on performance. This may be worthwhile since the experiments performed in this thesis show that compared to N_S, strategy plays a greater role on its own as well as in conjunction with other factors in influencing the behaviour of the algorithm.

There could also be more effective strategies. The strategies presented in this thesis are rudimentary decisions, but demonstrate the strategy function has a measurable impact on performance criteria. Strategies could be devised that attempt to focus on improving a single statistic deemed important to a particular class of agent. For example, a vehicle agent reversing course in a driving simulation could require slowing down. In the context of a game involving a pursuing agent, slowing down at all may be undesirable to following a longer course but at full speed. In this way, a strategy could be tailored to incorporate such domain specific information. Thus in some domains Salient Search may be capable of producing behaviour altogether preferable to other real-time algorithms like TBA*.

References

[AHU73] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 253–265, New York, NY, USA, 1973. ACM.

[BB09] Vadim Bulitko and Yngvi Björnsson. kNN LRTA*: Simple subgoaling for real-time search. In Christian Darken and G. Michael Youngblood, editors, *AIIDE*. The AAAI Press, 2009.

[BBS09] Yngvi Björnsson, Vadim Bulitko, and Nathan R. Sturtevant. TBA*: Time-bounded A*. In Craig Boutilier, editor, *IJCAI*, pages 431–436, 2009.

[BEH+03] Yngvi Björnsson, Markus Enzenberger, Robert Holte, Jonathan Schaeffer, and Peter Yap. Comparison of different grid abstractions for pathfinding on maps. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 1511–1512. Morgan Kaufmann, 2003.

[BEHS05] Yngvi Björnsson, Markus Enzenberger, Robert C. Holte, and Jonathan Schaeffer. Fringe search: Beating A* at pathfinding on game maps. In *CIG*. IEEE, 2005.

[BFc00] Michael A. Bender and Martn Farach-colton. The LCA problem revisited. In *In Latin American Theoretical INformatics*, pages 88–94. Springer, 2000.

[BLS+08] Vadim Bulitko, Mitja Lustrek, Jonathan Schaeffer, Yngvi Björnsson, and Sverrir Sigmundarson. Dynamic control in real-time heuristic search. *J. Artif. Intell. Res. (JAIR)*, 32:419–452, 2008.

[BMS04] Adi Botea, Martin Mller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28, 2004.

[BSLY07] Vadim Bulitko, Nathan R. Sturtevant, Jieshan Lu, and Timothy Yau. Graph abstraction in real-time heuristic search. *J. Artif. Intell. Res. (JAIR)*, 30:51–100, 2007.

[CSE98] Bioware Corp, Black Isle Studios, and Interplay Entertainment. Baldur's gate. [CD-ROM], 1998.

[CSE00] Bioware Corp, Black Isle Studios, and Interplay Entertainment. Baldur's gate II: Shadows of amn. [CD-ROM], 2000.

[Dew89] A. K. Dewdney. A tinkertoy computer that plays tic-tac-toe. *Scientific American*, pages 120–123, October 1989.

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connection With Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[Ent02] Blizzard Entertainment. Warcraft III: Reign of chaos. [CD-ROM], 2002.

[FHL08] Dave Ferguson, Thomas M. Howard, and Maxim Likhachev. Motion planning in urban environments. *J. Field Robotics*, 25(11-12):939–960, 2008.

[Fox05] John Fox. The R commander: A basic-statistics graphical user interface to R. *Journal of Statistical Software*, 14(9):1–42, 8 2005.

[GG05] Firaxis Games and 2K Games. Sid meier's civilization IV. [CD-ROM], 2005.

[HM05] Carlos Hernández and Pedro Meseguer. LRTA*(k). In *IJCAI*, pages 1238–1243, 2005.

[HM07] Carlos Hernández and Pedro Meseguer. Improving LRTA*(k). In *IJCAI*, pages 2312–2317, 2007.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[IK91] Toru Ishida and Richard E. Korf. Moving target search. In *IJCAI*, pages 204–211, 1991.

[Ish92] Toru Ishida. Moving target search with intelligence. In *AAAI*, pages 525–532, 1992.

[KL02] Sven Koenig and Maxim Likhachev. D*-Lite. In *AAAI/IAAI*, pages 476–483, 2002.

[KL06] Sven Koenig and Maxim Likhachev. Real-time adaptive A*. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors, *AAMAS*, pages 281–288. ACM, 2006.

[Koe98] Sven Koenig. Exploring unknown environments with real-time search or reinforcement learning. In *NIPS*, pages 1003–1009, 1998.

[Koe01] Sven Koenig. Agent-centered search. *AI Magazine*, 22(4):109–132, 2001.

[Koe04] Sven Koenig. A comparison of fast search methods for real-time situated agents. In *AAMAS*, pages 864–871, 2004.

[Kor85] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Kor90] Richard E. Korf. Real-time heuristic search. *Artif. Intell.*, 42(2-3):189–211, 1990.

[KS98] Sven Koenig and Reid G. Simmons. Solving robot navigation problems with initial pose uncertainty using real-time heuristic search. In *AIPS*, pages 145–153, 1998.

[KTN+99] Hiroaki Kitano, Satoshi Tadokoro, Itsuki Noda, Hitoshi Matsubara, Tomoichi Takahashi, Atsuhi Shinjou, and Susumu Shimada. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *SMC*, pages 739–746. IEEE Computer Society, 1999.

[KTS03] Sven Koenig, Craig A. Tovey, and Yury V. Smirnov. Performance bounds for planning in unknown terrain. *Artif. Intell.*, 147(1-2):253–279, 2003.

[LFG+05] Maxim Likhachev, David I. Ferguson, Geoffrey J. Gordon, Anthony Stentz, and Sebastian Thrun. Anytime Dynamic A*: An anytime, replanning algorithm. In *ICAPS*, pages 262–271, 2005.

[MK10] Masataka Mizusawa and Masahito Kurihara. Hardness measures for grid-world benchmarks andperformance analysis of real-time heuristic search algorithms. *Journal of Heuristics*, 16:23–36, 2010.

[R D10] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[RDB+07] D. Chris Rayner, Katherine Davison, Vadim Bulitko, Kenneth Anderson, and Jieshan Lu. Real-time heuristic search with a priority queue. In *IJCAI*, pages 2372–2377, 2007.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[SB05] Nathan R. Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *AAAI*, pages 1392–1397, 2005.

[Sim96] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, USA, 3rd edition, 1996.

[Ste94] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *ICRA*, pages 3310–3317, 1994.

[Stu07] Nathan R. Sturtevant. Memory-efficient abstractions for pathfinding. In *AIIDE*, pages 31–36, 2007.

[Stu10] Nathan Sturtevant. HOG - Hierarchical Open Graph, June 2010. `http://webdocs.cs.ualberta.ca/ nathanst/hog.html`.

[SYCK09] Xiaoxun Sun, William Yeoh, Po-An Chen, and Sven Koenig. Simple optimization techniques for A*-based search. In *AAMAS (2)*, pages 931–936, 2009.

[Yap02] Peter Yap. Grid-based path-finding. In Robin Cohen and Bruce Spencer, editors, *Canadian Conference on AI*, volume 2338 of *Lecture Notes in Computer Science*, pages 44–55. Springer, 2002.

[ZSH+09] Zhifu Zhang, Nathan R. Sturtevant, Robert C. Holte, Jonathan Schaeffer, and Ariel Felner. A* search with inconsistent heuristics. In *IJCAI*, pages 634–639, 2009.

Appendix A

Additional Figures and Tables

A.1 is a full breakdown of the heuristic error encountered for each map. Given a heuristic H and a problem $\langle S, G \rangle$, heuristic error is the difference between the score returned by $H(S, G)$ and the A* optimal path. Each value is the mean of 100 searches.

Map	mean	mean %	σ %
adrenaline	38.444	18.51	19.14
Battleground	12.064	7.87	9.92
blastedlands	4.148	2.93	4.47
darkforest	7.582	5.36	5.82
divideandconquer	8.728	4.91	5.25
dragonfire	9.888	6.66	6.79
gardenofwar	18.248	11.19	9.23
gnollwood	6.370	4.69	6.22
harvestmoon	7.792	4.96	6.05
icecrown	4.672	2.72	3.93
mysticisles	18.162	9.37	11.08
petrifiedforest	12.110	6.57	8.70
scorchedbasin	7.522	5.61	6.37
Continued on Next Page...			

Map	mean	mean %	σ %
thecrucible	11.160	6.90	5.01
tranquilpaths	13.516	8.17	7.10
AR0011SR	51.976	19.82	19.21
AR0202SR	111.238	36.58	21.12
AR0204SR	60.324	19.41	17.95
AR0205SR	44.630	20.30	16.23
AR0300SR	57.478	22.17	16.22
AR0307SR	131.048	41.17	20.89
AR0400SR	102.806	30.62	23.70
AR0404SR	68.972	27.70	17.53
AR0405SR	89.270	33.90	22.19
AR0406SR	83.750	30.12	19.77
AR0411SR	107.028	32.05	22.48
AR0414SR	15.046	8.81	13.40
AR0500SR	43.838	16.61	18.40
AR0516SR	25.184	12.80	11.97
AR0602SR	115.160	37.50	19.01
AR0603SR	91.360	27.87	21.74
AR0700SR	29.734	15.01	13.77
AR0701SR	18.462	11.24	10.03
AR0012SR	26.714	14.97	13.64
AR0013SR	14.610	9.14	8.99
AR0014SR	16.884	11.44	11.37
AR0070SR	131.658	36.50	27.87
AR0071SR	83.522	30.58	20.66
AR0308SR	19.778	9.05	14.35

Continued on Next Page...

Map	mean	mean %	σ %
AR0309SR	31.184	17.01	17.53
AR0412SR	15.366	8.02	12.40
AR0413SR	39.960	20.22	13.05
AR0504SR	21.818	12.64	13.93
AR0505SR	21.650	12.58	12.98
AR0510SR	11.852	7.31	7.25
AR0511SR	16.720	10.56	13.73
AR0600SR	20.448	9.93	10.81
AR0601SR	23.954	9.73	14.64
AR0705SR	100.904	32.38	20.05
AR0711SR	37.108	14.94	10.49

Table A.1: Heuristic error breakdown by map

Table A.2: Travel Ratio of Salient Search on cardinal grid-type

AD				TB			
R	N_S%	Exp./Move	Ratio	R	N_S%	Exp./Move	Ratio
25	30	8.053635	1.369197	25	30	8.249332	1.342698
	50	8.129356	1.385923		50	8.305299	1.360589
	75	8.204448	1.405837		75	8.365007	1.383436
50	30	9.792802	1.164522	50	30	9.845291	1.156884
	50	9.962391	1.173848		50	9.941612	1.165815
	75	10.179030	1.183557		75	10.096441	1.177184
100	30	11.050755	1.065947	100	30	11.045822	1.062087
	50	11.154335	1.068111		50	11.039262	1.064304
	75	11.297010	1.070144		75	11.032585	1.065987
500	30	11.761439	1.008413	500	30	11.701709	1.007428
	50	11.735883	1.008454		50	11.534619	1.007529
	75	11.735735	1.009019		75	11.307667	1.007709
1000	30	11.847371	1.003523	1000	30	11.750441	1.003168
	50	11.795003	1.003485		50	11.564756	1.003130
	75	11.879790	1.003693		75	11.453937	1.002990

Table A.3: Travel Ratio of TBA* on Cardinal grid-type

R	Exp./Move	Ratio
25	8.288570	1.312557
50	9.842781	1.140516
100	10.994856	1.054609
500	11.905271	1.006960
1000	11.994006	1.003201

Appendix B

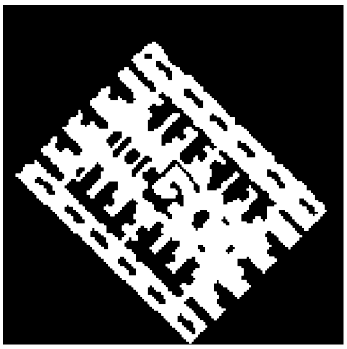
Maps

The following is a complete listing of all 50 maps used in the experiment found in Chapter 4. This set is derived from a collection found in [Stu10], which in turn are originally sourced from [CSE98, CSE00, Ent02].

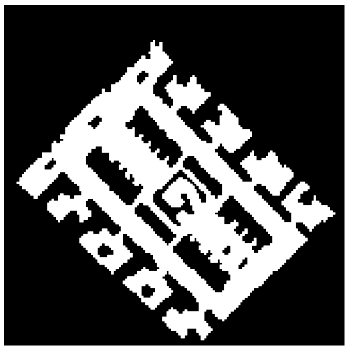
Baldur's Gate Maps

(1) ar0011sr

(2) ar0012sr



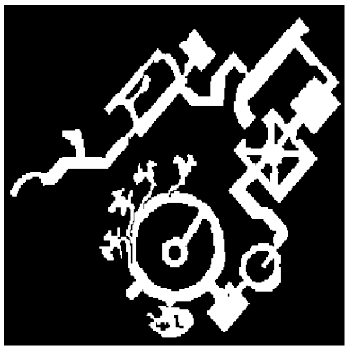
(3) ar0013sr



(4) ar0014sr

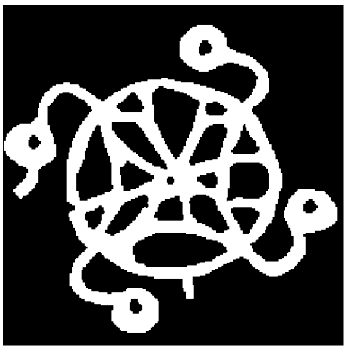
(5) ar0070sr

(6) ar0071sr



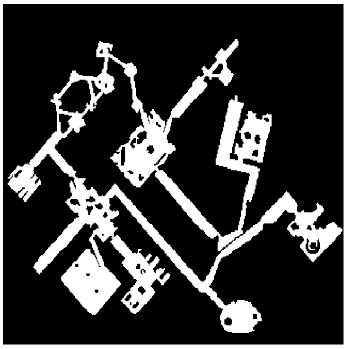
(7) ar0202sr

(8) ar0204sr



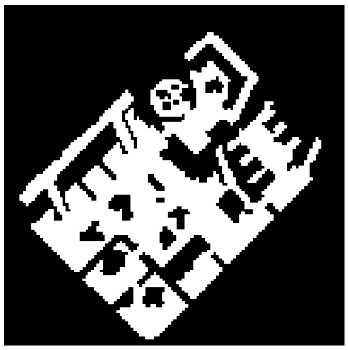
(9) ar0205sr

(10) ar0300sr



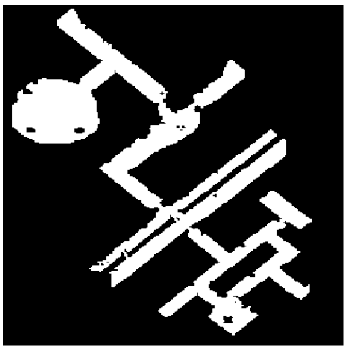
(11) ar0307sr

(12) ar0308sr

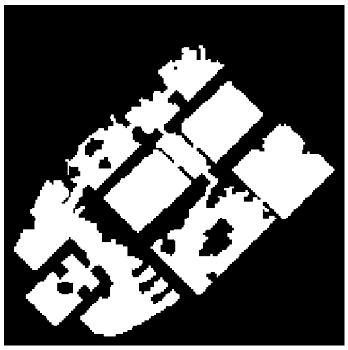


(13) ar0309sr

(14) ar0400sr

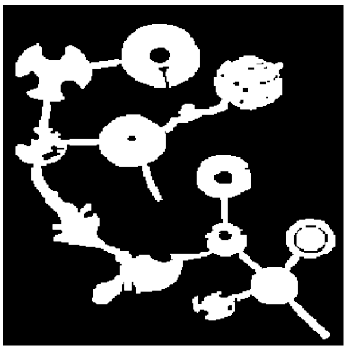


(15) ar0404sr

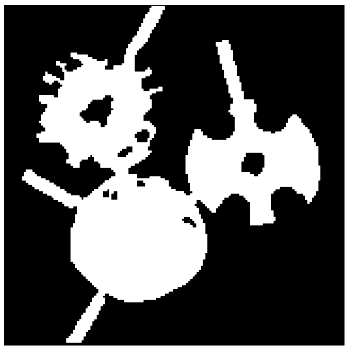


(16) ar0405sr

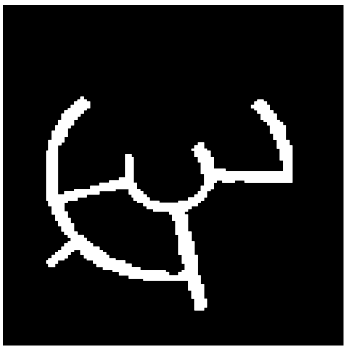
(17) ar0406sr



(18) ar0411sr

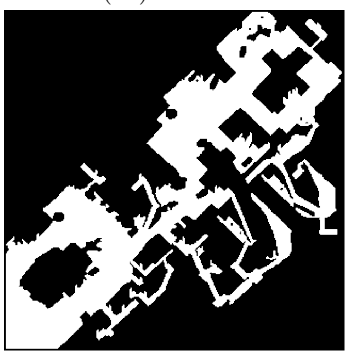


(19) ar0412sr



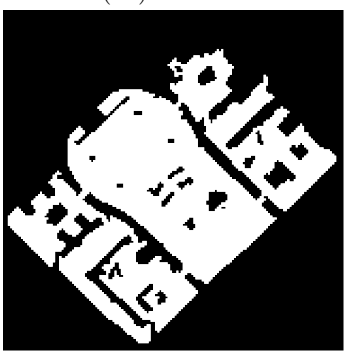
(20) ar0413sr

(21) ar0414sr

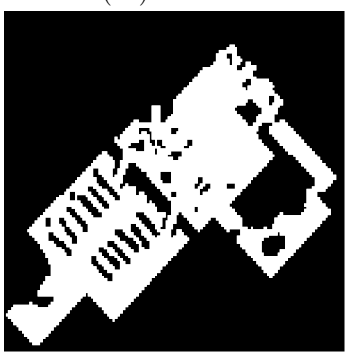


(22) ar0500sr

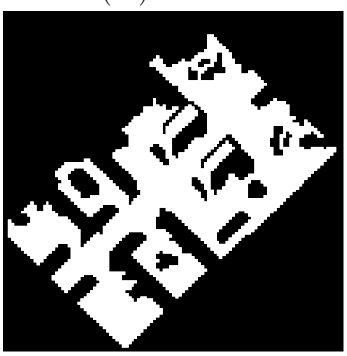
(23) ar0504sr



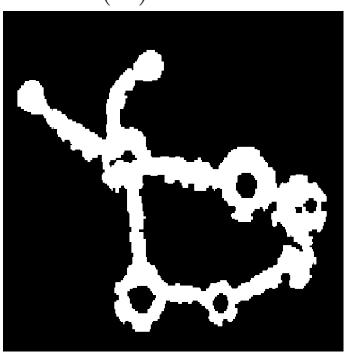
(24) ar0505sr



(25) ar0510sr

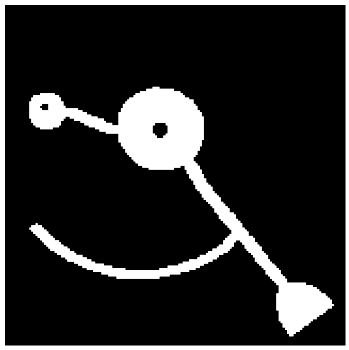


(26) ar0511sr

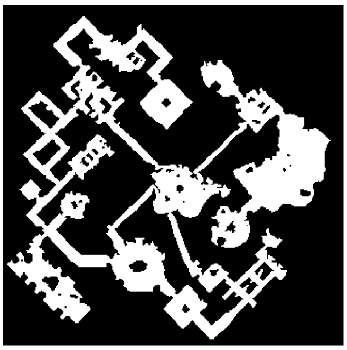


(27) ar0516sr

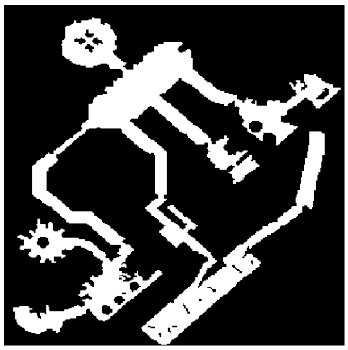
(28) ar0600sr



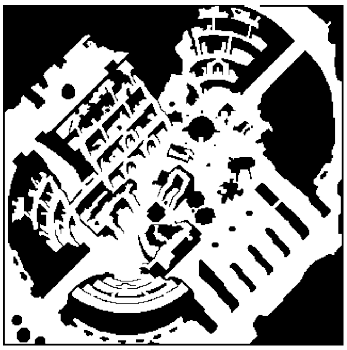
(29) ar0601sr



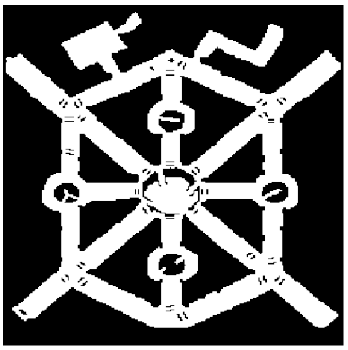
(30) ar0602sr



(31) ar0603sr

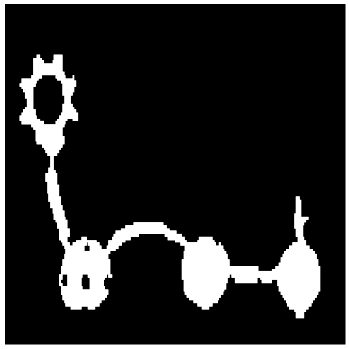


(32) ar0700sr



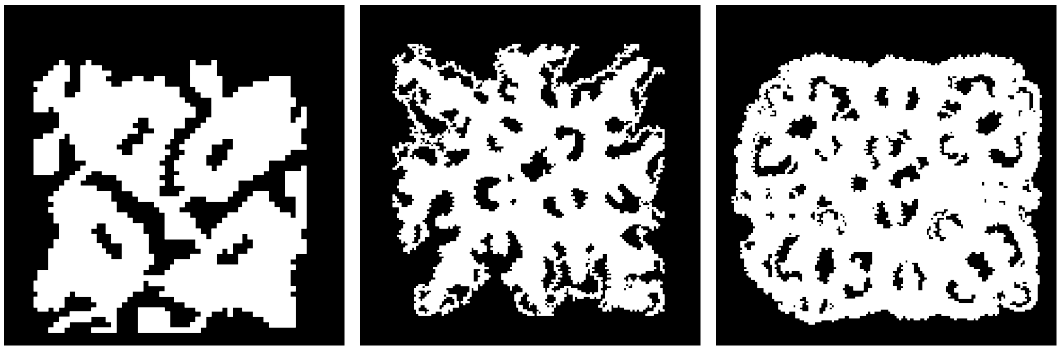
(33) ar0701sr

(34) ar0705sr

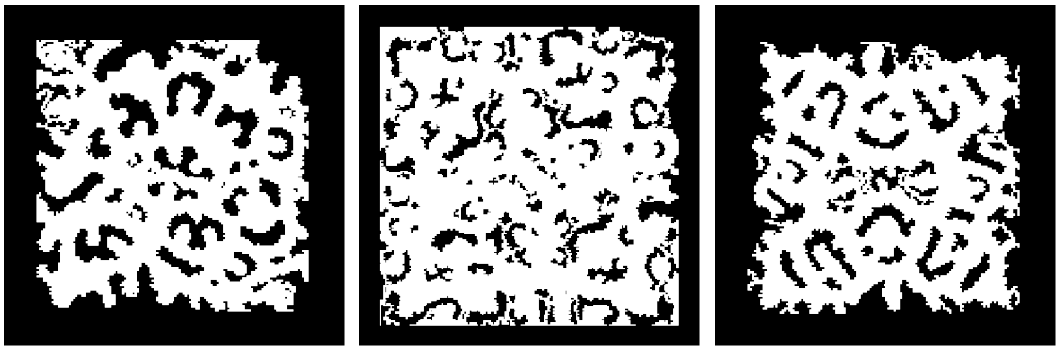


(35) ar0711sr

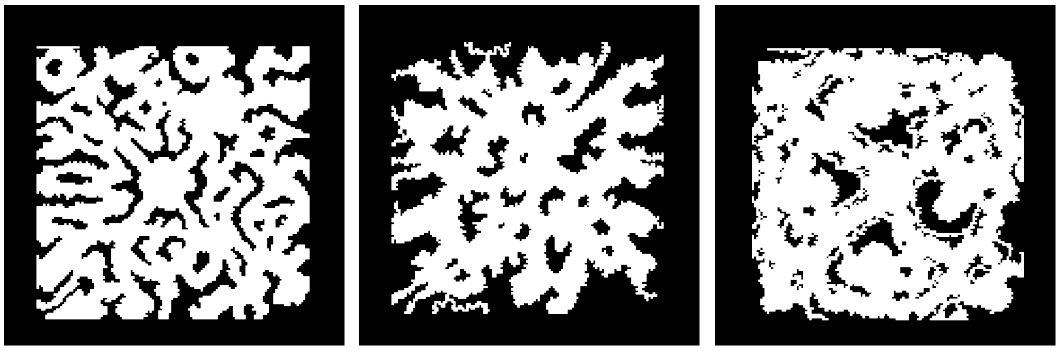
Warcraft 3 Maps



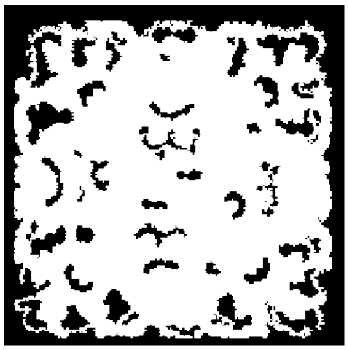
(1) adrenaline (2) battleground (3) blastedlands



(4) darkforest (5) divideandconquer (6) dragonfire

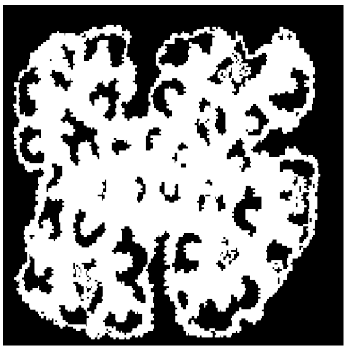


(7) gardenofwar (8) gnollwood (9) harvestmoon

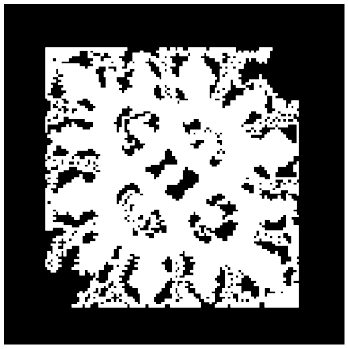


(10) icecrown

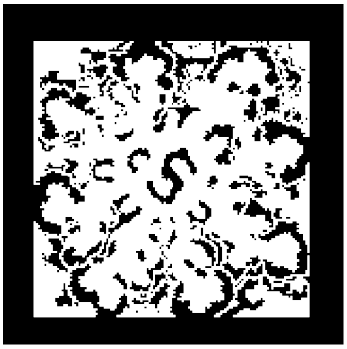
(11) mysticisles



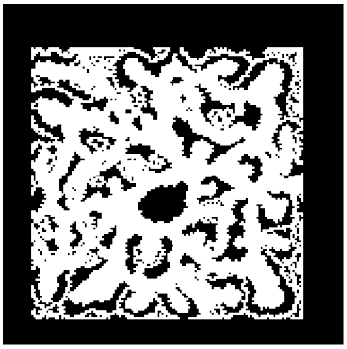
(12) petrifiedforest



(13) scorchedbasin



(14) thecrucible



(15) tranquilpaths

Vita Auctoris

NAME	Jonathan Vermette
PLACE OF BIRTH	Windsor, Ontario
YEAR OF BIRTH	1984
EDUCATION	

2003 - 2008 B. Sc.[H]
School of Computer Science
University of Windsor
Windsor, Ontario, Canada.