Real-time Cooperative Pathfinding

Alexander Toresson

Examensarbete för 15 hp
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in computer science, 15 ECTS credits
Department of Computer Science, Faculty of Science, Lund University

Abstract

Currently, not much research has been done on using real-time pathfinding algorithms in a cooperative environment, and adapting them to function well for this purpose. This thesis evaluates existing real-time pathfinding algorithms in such an environment, and improves and extends them to perform better. Existing algorithms are for example LRTA*, LRTS and PR LRTS, and this thesis extends LRTS and PR LRTS, comparing different approaches to this. A new algorithm, CPR LRTS, based on PR LRTS and WHCA* with map abstraction, is introduced. The new algorithms are compared to some existing non-real-time cooperative pathfinding algorithms and to each other, and CPR LRTS is shown to be superior to other real-time algorithms and competitive with other non-real-time cooperative pathfinding algorithms.

Sammanfattning

För tillfället har inte mycket forskning gjorts på hur man använder vägsökningsalgoritmer med realtidskrav i en kooperativ miljö och hur man anpassar dem till fungera bra för detta ändamål. Detta examensarbete går igenom och evaluerar existerande sådana algoritmer i en kooperativ miljö och utökar och förbättrar dem till att fungera bättre. Existerande algoritmer är till exempel LRTA*, LRTS och PR LRTS. Algoritmerna LRTS och PR LRTS utökas och olika sätt att göra detta på jämförs. En ny algoritm, CPR LRTS, tas fram. Den baseras på PR LRTS och WHCA* med abstrahering av kartor. De nya algoritmerna jämförs med varandra och med några existerande vägsökningsalgoritmer som inte har realtidskrav. Det visas att CPR LRTS är överlägsen de övriga algoritmerna som har realtidskrav och ofta är jämförbar med algoritmerna som inte har realtidskrav.

Contents

Acknowledgements

I would like to thank my examiner Ferenc Belik for getting me started with my thesis. I would also like to thank Jacek Malec for being a very good advisor, grasping the problem area and providing valuable feedback.

I would like to thank Nathan Sturtevant for providing the Hierarchical Open Graph framework and his WHCA* and CPRA* implementations for it, and Vadim Bulitko, for answering questions regarding LRTS.

This thesis would not have been possible without these 4 persons.

Finally, I'd also like to thank Marcus Klang and Linnéa Persson for proofreading this report and providing good criticism.

1 Introduction

1.1 Pathfinding

Pathfinding is the act of calculating paths between two locations. That is, it is the latter part of path planning, and is executed after it has been decided where an agent is going. Path planning also encompasses deciding where an agent should go, and when it should go there.

A pathfinding problem is usually discretized by turning the search space into a graph, where nodes represent locations and edges ways of travelling between them. Edges may have uniform cost, but usually they have costs assigned to them. The latter representation where edges have costs will be the representation used in this thesis.

It is common to use cell-based node layouts for maps in games and for other applications. Those cells can for example be tiles (squares lying in a grid where agents can move upwards, downwards, left and right), octiles (tiles where diagonal movement is also allowed) and hexagons. In this thesis, it is assumed that the map consists of octiles, as that is the most commonly used cell-based layout.

Common graph pathfinding algorithms can be divided into a few categories by their properties:

- **Complete/incomplete**
  Whether the algorithm always finds a solution if one exists.

- **Optimal/non-optimal**
  An optimal pathfinding algorithm always finds the best solution to a problem, measured by some criteria. Often, distance or time are the criteria chosen to be minimized. Edge weights are chosen to match this criteria.

- **Real-time/non-real-time**
  A pathfinding algorithm is real-time if the time it requires to calculate what direction an agent should take at every moment is provably within a certain bound, regardless of the problem space. Usually, the space used is also assumed to be within a certain bound.

- **Continuous/non-continuous**
  Does the algorithm calculate the full path to the goal at once, or does it perform it continually while the agent traverses the search space? Note that in a cooperative environment, even non-continuous algorithms may need to reevaluate their calculated path, if it turns out that it has become blocked at some point.

- **Cooperative/single-agent**
  Does the algorithm take other agents into special consideration when performing pathfinding? A common approach to handling a cooperative environment with a single-agent algorithm is to treat cells that contain another agent as currently blocked. All single-agent algorithms in this thesis use this approach.
  Cooperative pathfinding algorithms try to introduce cooperation into the pathfinding

of agents, so that they cooperate in reaching a goal. When cooperating, performance of the single agent may have to be sacrificed for the greater good of the group. Examples of cooperativity in pathfinding is temporal-spatial search and methods that reduce collision rate.

1.2 The problem of cooperative pathfinding

When several agents move around in an environment, they will at times block each other's paths. If this is ignored, an algorithm will not be able to solve some situations, and will solve other situations suboptimally.

Consider for example the situation where two agents meet in a narrow corridor, where neither can pass each other. One of them needs to move out of the corridor again for the other to pass by. A situation that resembles this one is when an agent has its goal position in the narrow corridor, and is standing at it, blocking the path of other agents.

Another situation is when two agents collide while traversing an open area, and one or both have to recalculate their paths, while their paths could have been calculated so that they never would have collided in the first place. This problem gets worse when the current level of congestion gets worse.

However, the problem to optimally solve a cooperative pathfinding problem is PSPACE-hard[9], that is, space needed (and thus time) depends upon graph size and grows polynomially. Also, the degree of the polynomial increases dramatically when the number of agents increase, as branching increases polynomially.

At every discrete time step of the simulation, the number of possible actions depends on the graph and the number of agents. If the branching factor in a graph is b, there are n agents in the environment, the problem can be solved within m time steps, and all agents move from an arbitrary node to a neighbouring node in 1 time step, there are b^{nm} states that can be considered.

Thus usually the planning for every agent is done separately, as that significantly reduces time complexity. Such an algorithm will not be able to solve every problem, but it will nonetheless be able to solve many of them. All algorithms described in this thesis use separate planning per agent.

1.3 Why cooperative and real-time?

One possible use case would be an environment where several agents, controlled by one central computer, coexist and move around in the environment. To guarantee that the problem is schedulable by the computer, one wants to have an upper bound on the computation time needed for every agent.

Real world examples of this is for example controlling robots that are moving around in an environment, and controlling units in a computer game.

A variation on the scenario above is when every agent has its own computer, and has to be able to decide on a path to take in real-time, while cooperating with other agents to minimize overall plan execution time.

The overall plan execution time is the time it takes for all the agents to together solve a problem, and thus it is the time taken until the last agent solves its subproblem. Of course the situation gets more complicated if several plans are executed at once or are overlapping, as they would often be in reality.

Aspects which influence the plan execution time are:

- Cost of planned routes, including potential collisions on the way.
- Time before the agent starts moving, after receiving an order.

As this problem is PSPACE-hard[9], the full problem space cannot be searched, and a reasonable approximation has to be used.

1.4 Problem formulation

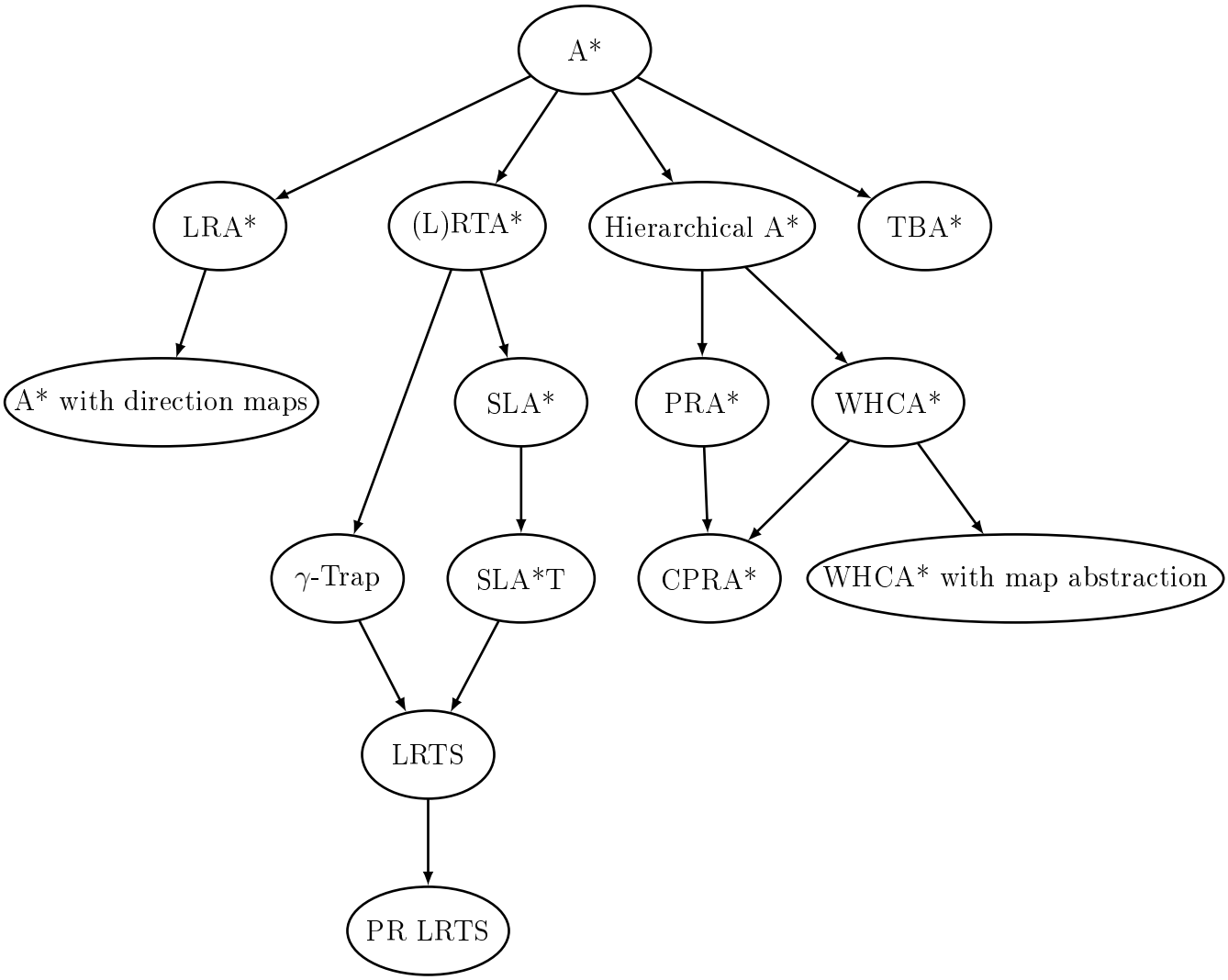
Real-time algorithms have not been evaluated in cooperative environments, as far as I know. The question this thesis tries to address is thus: how would they perform, what parameters enable them to work best, and can they be improved?

The algorithms will be evaluated in fully-known and fully-controlled environments. That is, both which nodes are blocked and not blocked is known, and all agents are fully controlled and known.

2 Background

2.1 Relations between the covered algorithms

The following graph illustrates the relationships between the algorithms covered in this section. An arrow from algorithm A to algorithm B means that B uses ideas from A.



2.2 A*

A*[7] is the classic pathfinding algorithm for graphs. It is used in, among other areas, robotics, AI gameplay and game theory.

A* is related to Dijkstra's algorithm. As Dijkstra's algorithm, it uses a priority queue of nodes with costs (the so called *open list*), and expands nodes in the order of ascending costs in the queue. The differences from Dijkstra's algorithm is that A* has a goal node for the search and calculates the cost of nodes in a different way. Usually a *closed list* is also

kept, through which one can check whether a node has already been expanded or not. This is useful in the cases where nodes do not need to be expanded more than once (see below).

Where Dijkstra's algorithm uses the exact cost from the start node as the cost of a node, A* uses a function $f = g + h$, where g is the exact cost from the start node to the current node, and h is an estimate of the distance from the node to the goal node. h is called the heuristic function, and the quality of it determines how well A* performs.

A heuristic function can have a few different properties. One of them is *admissibility*, which means that the heuristic never overestimates the cost between two nodes. If a heuristic function satisfies this property, A* will return an optimal path between the start and the goal.

Another property that a heuristic may have is consistency, which means that $f = g + h$ never decreases when getting farther away from the start node. This is a variation on the common triangle inequality. If a heuristic function satisfies this, a node will never have to be expanded more than once.

A* is a *complete* algorithm; that is, it always finds a solution if one exists. It is also *optimally efficient* (assuming an admissible heuristic), that is, it does not expand more nodes than any other complete and optimal search algorithm that uses the same heuristic.

With a consistent heuristic and a binary heap used as priority queue, A* achieves a worst-case time complexity of $O(n \log n)$, where n is the number of nodes expanded. In the worst case, A* will have exponential space usage (but limited to the size of the graph, if the graph is finite).

The performance improvement achieved compared to using Dijkstra's algorithm, if any, depends on how big part of the graph can be skipped with help of the heuristic. A* has the same time complexity as Dijkstra's algorithm, but a slightly higher constant, so to perform equivalently to Dijkstra's algorithm, at least part of the graph has to be skipped.

Algorithm pseudo-code

```
A*(Graph, Heuristic, Start, Goal):
        H[Start] = Heuristic(Start, Goal)
        G[Start] = 0
        Parent[Start] = None
        OpenList <- {Start}
        ClosedList <- {}
        While OpenList not empty:
                CurrentNode = The node in OpenList with lowest F = G + H
                ClosedList.Add(CurrentNode)
                If CurrentNode = Goal:
                        Return reconstructed path from Start to Goal by using Parent[]
                ForEach neighbor of CurrentNode:
                        NewG = G[CurrentNode] + Graph.Distance(CurrentNode, Neighbor)
                        If Neighbor is in ClosedList and newG < G[Neighbor]:
                                G[Neighbor] = NewG
                                Parent[Neighbor] = CurrentNode
                                OpenList.Add(Neighbor)
                        ElseIf Neighbor is in OpenList and newG < G[Neighbor]:
                                G[Neighbor] = NewG
```

```
                    Parent[Neighbor] = CurrentNode
                    Put Neighbor in new position in OpenList
                ElseIf Neighbor is neither in OpenList nor ClosedList:
                    G[Neighbor] = NewG
                    H[Neighbor] = Heuristic(Neighbor, Goal)
                    Parent[Neighbor] = CurrentNode
                    OpenList.add(Neighbor)
    Return empty path
```

When a node's neighbors are being iterated through, it is said that the node is being *expanded*. A node that is considered when another node's neighbors are being iterated through is considered *touched*.

One simple addition to A* is to keep track of which expanded node was closest to the goal. If the goal was never reached, a path to this node is returned instead.

2.3 Map clique abstraction

An abstraction of a map is a version of a map with fewer nodes, where several nodes are merged into one. Such a node is called a parent of the nodes it abstracted, and a node may only have one parent, and must have 1 or more children The obvious exception is on the lowest (non-abstracted) level, where a node of course has no children.

A *clique* in map clique abstraction[16] is a cluster of nodes which all have edges between them, so that all nodes in the clique are neighbors. An *n-clique* is a clique consisting of n nodes.

Map clique abstraction also often allows a node, that has only one edge connected to it (so-called adoptees), to be merged into the clique it is connected to. In this thesis this approach is used for the empirical evaluation.

2.4 LRA*

Local Repair A*[13] is a variant of A* which can be used in a cooperative environment. LRA* simply replans the path when a collision is detected, and considers nodes currently occupied by other agents as blocked.

Usually, randomness is also introduced into the agent's path after collisions to attempt to escape from deadlocks.

2.5 Weighted A*

Weighted A*[6] allows bounded suboptimality of the search, by weighting the heuristic function h by $(1+\varepsilon)$. This focuses the search slightly, making the algorithm expand fewer nodes, thus resulting in lower search time. The reason for this is that an admissible heuristic usually underestimates the cost to the goal. The length of the path returned may be suboptimal, however, but the suboptimality of it will be within a factor of $(1+\varepsilon)$.

2.6 RTA* and LRTA*

RTA* and LRTA*[11] use a search with limited lookahead (lookahead to a specific depth), which it bases on minimax search, and calls minimin search, as costs should be minimized on all plies. The search is a modified Dijkstra search, or a BFS (Breadth First Search) with re-expansion.

As with A*, a heuristic function is used with the search. From the limited lookahead search, a better heuristic value for the node is obtained and stored in a hash table containing improved heuristic values. This limited lookahead is also the reason to why (L)RTA* maintains real-time properties for computational cost and memory usage. When looking up a heuristic value, this table is primarily consulted, and if that does not yield an answer, the regular heuristic function is consulted. The hash table serves to build an improved heuristic function for the domain.

This type of search is non-exhaustive and thus does not guarantee that the agent will move on a path directly towards the goal. That is, it may fall into local minima of the heuristic function. However, thanks to the hash table of stored heuristic values, it will eventually work itself out of the minima and continue on a better path towards the goal.

The main difference between RTA* and LRTA* is that RTA* stores the second lowest $f = g + h$ value generated by the minimin search. This works well when the hash table is not reused for subsequent searches toward the same goal. However, if it were reused, the heuristic values stored in the hash table may become non-admissible. LRTA* thus stores the lowest f value generated by the minimin search, to enable the hash table to be reused for solving additional problems.

2.7 SLA*

SLA*[12] is based on LRTA* with single lookahead, and adds backtracking. That is, moving backwards towards the last position. Backtracking is invoked when the stored heuristic is updated.

2.8 SLA*T

SLA*T[5] is based on SLA*, but uses a different strategy to determine when backtracking is invoked. Backtracking is only invoked when a certain learning limit is reached.

2.9 LRTS

LRTS (Learning RealTime Search) [3] is a unification of a few variations on LRTA*. It incorporates some improvement from SLA*T[5] and γ-Trap[2] in the search over LRTA* and adds additional parameters.

The extensions are:

- **Improved learning through max of min at every level**
  Instead of taking the minimum of the minimums at each level, it takes the maximum of the minimums at each ply.

- **Backtracking**
  When a limit L of learning has been reached, the algorithm doesn't allow this value to be exceeded, and backtracks to the last position if going forward would cause it to be exceeded.

- **Weighting**
  Weighting is not done as in Weighted A*, but it has approximately the same effect. Instead of weighting the heuristic (h) by ($1+\varepsilon$), the distance from the start node (g) is weighted by a factor $\gamma \leq 1.0$. The main advantage is that this will prevent the distance from being overestimated, and thus the stored heuristic will be kept admissible.
  Weighting by a factor γ is, apart from being admissible, equivalent to weighting by $\varepsilon = \frac{1}{\gamma} - 1$.

The algorithm preserves real-time properties with these extensions.

2.10 PR LRTS

PR LRTS [4] is LRTS with path refinement. It computes a path using LRTS at an abstracted level in a map clique abstraction, and then it refines it using A*. This is achieved by performing an A* search in the corridor formed by the path calculated at an abstracted level. That is, all children of the nodes in the abstracted path are part of the corridor.

As the number of nodes in the generated corridor is limited, the computational cost and memory usage of the A* search is bounded by this amount. As the LRTS search is also real-time, the algorithm in its entirety is real-time.

2.11 Hierarchical A*

Hierarchical A*[8] uses abstractions of the graph to compute and use A* searches at iterative levels as a heuristic for searches at lower levels. With a good heuristic, this doesn't bring any performance advantage, but it has advantages when using a bad heuristic.

2.12 TBA*

TBA*[1] (Time-Bounded A*) runs a regular A* search, but stops after a fixed number of nodes have been expanded, and makes the agent go towards the most promising node. It implements priority queues and open/closed lists through hash tables. It does thus achieve real-time properties for the computational cost involved. However, the memory usage that is needed does not have an upper bound, except for the one imposed by the size of the environment, if any.

If the agent is not on a direct path to the most promising node, the agent either backtracks to the first common node or performs a small search to find a shortcut to a node that is on a direct path to the most promising node.

2.13 PRA*

PRA*[15] (Partial Refinement A*) uses prebuilt map abstractions and performs an A* search in a base abstraction, then successively refines the result until it reaches the unabstracted representation. At every refining step, the A* search is restricted by the corridor around the previously abstracted search. PRA* has as a parameter the length of the path that will be refined at each step; the length of the path will be truncated to this.

2.14 WHCA*

WHCA*[13] (Windowed Hierarchical Cooperative A*) uses a temporal-spatial search, that is, a node in the search tree is not only defined by the position, but is also defined by the time the agent will be there. The search is also windowed; this means that it is performed a fixed number of steps into the future, and the most promising node at the edge of this window is selected.

To plan all the agents' paths at the same time carries an all-to-high computational cost[9]. Instead, a reservation table is used. The path of every agent is planned independently, and every position and time that a node is blocked is recorded in the reservation table. This is a reasonable approach to decrease the computational costs involved, which although it may not solve all problems, will be able to solve many of them.

To focus the search, a spatial reverse A* search is used as heuristic for the temporal-spatial search. That is, it is used to calculate the exact distance from a node to the goal, excluding the influence other agents may have on this distance. This generally focuses the search a lot, as this is often a very high-quality heuristic. It gets worse with a higher level of congestion in the environment.

However, this reverse A* search leads to a high initial cost when performing the first windowed temporal-spatial search, as it needs to perform a full A* search from the goal to the start (as well as to some nodes neighbouring the start), but the cost for subsequent searches will be lower.

2.15 WHCA* with abstraction

WHCA* with abstraction[16] performs the reverse A* search at an abstracted level, instead of at the base map level. This reduces memory usage and computational costs for the reverse A* search, but sacrifices it for less accuracy and lower resolution in the heuristic generated from it, which leads to more nodes expanded in the regular temporal-spatial A* search.

This is nonetheless often a sacrifice that is worth it, as it especially decreases the cost to calculate the reverse A* heuristic on the first time step. However, the abstraction level to run the reverse A* search on should be selected carefully.

2.16 CPRA*

CPRA*[16] is a modified PRA* search that replaces the last A* search with a WHCA* search. As PRA*, it takes a parameter which is the length of the path to refine at each step,

and the paths are truncated to this length.

2.17 A* with Direction Maps

Direction maps[10] adds penalties to the heuristic function by storing a direction for each node. An agent gets a penalty for not moving in the exact direction that is stored in the node. The penalty is proportional to the difference between the direction stored in the node and the actual direction the agent is moving in. The direction for a node changes when an agent moves across it, and the new direction is a weighted value between the old direction and the direction the agent is moving in. With time, the direction map will form lanes by discouraging agents from moving against the general flow of other agents.

Direction maps take two parameters: The weighting between an old direction and the new direction of a tile (α), and the maximum penalty that direction maps can incur (w_{max}).

3 New methods

The problem can be approached in at least two major conceptual ways:

1. Base a new algorithm on WHCA* or CPRA*, reduce number of calculations the way TBA* does. Or, equivalently, base it on TBA*, but extend the algorithm to be cooperative.

2. Base a new algorithm on PR LRTS, and extend it to be cooperative.

In this thesis, approach (2) was chosen, as it seemed more promising.

Algorithm parameters and their effects were investigated. Also, various methods to decrease branching were investigated, for example direction maps and weighted A*.

3.1 Extending TBA* to a cooperative environment

TBA* could be extended to work in a cooperative envorinment by replacing the spatial A* search with a temporal-spatial A* search, and optionally add windowing. To use a reverse A* search as an improved heuristic (like in WHCA*) is not possible, to preserve real-time properties.

The main advantage of this approach compared to extending PR LRTS is that it may have less of a chance to deadlock. However, search will likely be less focused as a reverse A* heuristic cannot be used, and many more nodes will thus be expanded, affecting performance and memory usage.

To counter this, some kind of windowing may be introduced, but as TBA* always searches from the initial starting position, this has to be dynamically shifted to take effect for different parts of the tree.

Backtracking in TBA* with a temporal-spatial search brings its own problems, as using the closed list to create a path used for backtracking may create a far from optimal path, as a temporal-spatial search was used, and other agents may have moved since the path was generated. Backtracking may also fail, as other agents may now be in the way. An approach to this may be to attempt a backtrack, but if another agent blocks the path, restart the search from scratch with the current position as start position. Another may be to perform a limited lookahead temporal-spatial A* search to find a way back toward the common node. This could also use a spatial A* search as a heuristic.

Another problem is that TBA* with temporal-spatial search may expand many nodes, of which most will become obsolete quite quickly, so nodes may need to be pruned successively. However, one may want to keep some of them, to facilitate backtracking. Pure TBA* does not achieve real-time properties for its memory usage either.

In essence, temporal-spatial searching brings many problems to TBA*. An approach might be to combine spatial and temporal-spatial search.

I deem extending TBA* to be less promising than extending (PR) LRTS, and will thus focus on the latter.

3.2 Extending LRTS and PR LRTS to a cooperative environment

LRTS and PR LRTS have not been evaluated a cooperative environment before, as far as I know. Thus this section describes novel adaptions, modifications and problems of LRTS and PR LRTS in a cooperative environment.

3.2.1 LRTS in a cooperative environment

LRTS can use backtracking. In a cooperative environment, this is a problem, as the path the agent came via may now be blocked. There could possibly be workarounds for this, but I have thus chosen to not use backtracking.

LRTS in a cooperative environment also has the issue of how to treat other agents. That a node is blocked by another agent needs to be taken into consideration, and that can be done in a few ways.

One option is to treat such nodes as if they were permanently blocked. However, because of how LRTS works, the stored heuristic may then become inadmissible when another agent moves to another node. Also, as agents may disturb each other in this way, it may increase path lengths, or even create deadlocks.

Another approach is to separate the calculation of the updated heuristic from the calculation of what path to take. The main problem with this approach is that it will double the number of nodes expanded and touched.

When the heuristic is non-admissible, the direct path to the goal may not be taken, if it is reachable within the lookahead used. Therefore a flag was added that forces an agent to go towards the goal if possible.

Present problems

Agents may enter deadlocks, as some situations may not be solvable. These include situations where agents have reached their goal and do not move to let others through and situations where agents move in opposite directions in a corridor and need to move back out of the corridor. The latter case usually works out in LRTS anyway, as the agent moves to a node a certain number of steps away from the current node, and this will usually eventually solve these situations.

An idea for future work would be to combine LRTS with temporal-spatial planning. Although this option is not explored in this thesis, this may be a viable approach. As in WHCA*, windowing could be used, however, a reverse A* search could not be used as heuristic. No corridor is available, and thus a reverse A* search would not have real-time properties.

Direction maps

To encourage agents to take paths that will avoid collisions, direction maps will be evaluated.

3.2.2 PR LRTS in a cooperative environment

Generating a corridor

As PR LRTS uses an A* search in a corridor generated by LRTS, how it is generated is of utmost importance. With map clique abstraction, there may only be a single edge between two abstracted nodes on the non-abstracted level, when going diagonally. These edges will thus become bottlenecks and sources of deadlocks.

One solution to this is to add the additional abstract nodes, that can be used to go between the two nodes, to the corridor, when this situation is detected. This option is hereafter called "fixing single connectivity".

Another solution is to perform a BFS (Breadth First Search) search from the nodes in the abstract corridor, to add neighbours of them to it, and the depth of this search is hereafter referred to as the "corridor width".

Untraversable corridors

If a corridor is generated that cannot be traversed because it is blocked, there are a few options on how to handle this.

One option is to have the agent stay ground and wait until the corridor is free, but this tends to create many deadlocks.

Instead of finding a path to the goal, it is possible to search for a path to the node that is closest to the goal. This should solve many of the deadlocks, but increase the number of expanded and touched nodes somewhat, as more nodes in the corridor will be expanded and touched when the goal cannot be reached.

Generating a traversable corridor

A traversable corridor is a corridor that is known to be traversable. There are two kinds of traversable corridors that could theoretically be generated:

- Momentarily traversable corridors
- Future traversable corridors

A momentarily traversable corridor is a corridor that is known to currently be traversable. It would enable a refined path to the goal to always be calculated, assuming the algorithm used to calculate this path does not use a reservation table. However, as the corridor is only known to be currently traversable, traversing it may actually fail in the future.

A future traversable corridor may be possible, that is, a corridor which is known to be valid when the agent traverses it. However, it would be hard to achieve, as the exact distance between two neighbouring abstract nodes would need to be known. It is unlikely that it is possible to know this when generating a corridor. Even if this was achieved, a big issue would be to find a good way to store what would be an abstracted reservation table.

This leaves momentarily traversable corridors. To generate one, one approach is to modify the abstraction to reflect that nodes become reachable and unreachable when agents move between them, by adding and removing edges. To enable an agent to actually move, one

also needs to make sure that the edges from the agent's current node to its neighbors are present when the pathfinding algorithm runs.

HOG already provides code that can remove edges from map clique abstractions, but it does not provide code to add edges to them. I set out to implement this, but never succeeded. Implementing this in HOG in $O(\log n)$ time may not be possible, however, the reason being that the map clique code in HOG requires two nodes connected by an edge to either both have a parent or both not, and thus some updates may require relatively many updates to the abstraction. This approach was thus skipped.

Reusing corridors

Agents may collide with other agents when traversing their corridor. In this case, the current corridor should be reused and a new one should not be generated. Generating a new one when this happens will with reasonably long corridors increase the maximum number of calculations needed (over a longer time than one time step) significantly, as an agent which often collides with other agents will perform many more calculations than agents that don't. This could be ignored when the calculations performed per time step is the only thing that matters, however.

In this thesis, corridors are always reused after collisions are detected.

PR LRTS with WHCA*: CPR LRTS

PR LRTS uses A* to refine a corridor to a final path. To introduce explicit cooperativity into the algorithm, WHCA* can be used instead. I've chosen to call this modified algorithm CPR LRTS, as in Cooperative PR LRTS, following the naming pattern of PRA* and CPRA*.

In addition, when an agent reaches its goal, it continues to perform a temporal-spatial reservation search in the corridor defined by the current single abstracted parent node.

Temporal-spatial search with a reservation table helps with solving more situations than a pure spatial search does, for example an agent that blocks a bottleneck may attempt to move out of it to allow another agent to move past. Also, an agent will do this regardless of whether it has reached its goal or not, while with PR LRTS or LRTS the agent would only have a chance of moving out of the bottleneck if it has not reached its goal yet.

Also, if the environment is fully known and one has full control over all agents, no real collisions can occur, and thus paths never have to be cancelled and recalculated. This could be taken advantage of, to interleave the calculations of agents, so that an equal amount of agents need to calculate a new path at every time step, potentially increasing the number of agents that a computer can be guaranteed to be able to calculate paths for significantly.

As with WHCA*, the reverse A* search can be performed at an abstracted level, and thus decrease memory usage and processing time used to build it. Whether this improves overall memory usage and processing time will have to be discovered through experiments, as the WHCA* search through the corridor will expand more nodes in the corridor when the heuristic gets worse.

The reverse A* heuristic and windowed temporal-spatial A* search are also likely to yield better real-time properties. The first time a search is performed, many nodes in the corridor are expanded by the reverse A* heuristic, but by using abstractions for it, this may be cut

down. Subsequent searches are likely to expand many fewer nodes in the corridor. The temporal-spatial A* search will always have a fixed maximum number of calculations per search it performs.

CPR LRTS qualifies as a real-time algorithm, as the computation required is limited by the size of the LRTS lookahead; the corridor abstraction level and widening parameters; and the WHCA* abstraction level and window size, and not by the map size.

Corridor restriction for CPR LRTS

As CPR LRTS uses a windowed temporal-spatial search, it is possible to restrict the corridor to just be long enough for a search with the specific WHCA* window size to work. A longer corridor is still calculated with LRTS, but the corridor used for the WHCA* search is limited to the current abstracted node which is a parent of the current node, and the few next abstracted nodes.

This naturally leads to the reverse A* heuristic being recalculated more times, but the LRTS corridor will be recalculated just as often as before, as the corridor used for the WHCA* search is calculated from the LRTS corridor.

Shared distance heuristic

LRTS constructs an improved heuristic while traversing an environment. This heuristic is lost when the algorithm is done for the agent and results from one agent is not shared with other agents while it is running.

An abstracted map typically has much fewer nodes than the original map has. Thus depending on how much memory is available, it could be justified to store an all-pairs heuristic in memory. This enables heuristic values to be globally stored and retrieved, and all agents will be able to learn from each other and use knowledge of local minima in the heuristic function that other agents learned.

Direction maps

As with LRTS, direction maps will be evaluated for PR LRTS. Direction maps have not been evaluated for use with temporal-spatial search yet, and thus it will be interesting to see how it will perform with CPR LRTS.

Present problems

Both PR LRTS and CPR LRTS are restricted by their corridors. As corridors are not guaranteed to be traversable, there's no guarantee that a path will be found through the corridor. But CPR LRTS has a bigger chance, as it uses temporal-spatial planning. However, CPR LRTS is limited by the window size and corridor size.

Also, as the paths of agents are planned independently, all situations cannot be solved.

3.3 Hypotheses

Temporal-spatial reservation search through PR LRTS corridor will decrease path lengths, reduce deadlocks, but not eliminate them.

More situations will be solvable with temporal search through corridors. But the algorithm will still be able to deadlock, when there is no possible solution available in the corridor, or the window size is too low. Temporal-spatial search also has the disadvantage of increasing memory usage and calculation time.

Continuous temporal-spatial reservation search reduces path length and deadlocks, without increasing maximum calculation time.

Agents which have arrived at their destination and are idle can still cooperate with other agents by moving out of the way when needed.

LRTS agents disturb each other's search even when they do not collide, by rapidly altering the environment.

LRTS uses a fixed lookahead search to calculate what direction it should take at the current step, and this lookahead search is also used to construct a new heuristic value for the current node. In a non-static environment, this value may change uncontrollably when nodes change between being reachable and not reachable. This may make the heuristic function h at least partially nonadmissible, which may increase path lengths.

A way of avoiding this is to separate calculation of the new heuristic value from the calculation of what node to travel to – nodes that one can travel to would only include those that are actually reachable from the current position.

Weighting decreases path length and memory usage.

This has been discovered by Bulitko et al[3]. It will be tested to see whether this also works in a cooperative environment.

Shared distance heuristics can be used to decrease overall path length, by gradually learning the environment.

When an agent traverses an environment, it builds an improved heuristic of the environment. It would be desirable that accumulated knowledge is shared between agents.

Deadlocks and path length can be reduced by using direction maps.

If agents are encouraged to use specific routes which cause agents travelling in opposite directions to have less of a chance of blocking each other's paths, deadlocks and distances travelled may be reduced. However, distances travelled may also be somewhat increased, in case of non-congested areas.

Updating h of all intermediary nodes in LRTS will improve convergency times.

LRTS only updates the heuristic value of the current node. It is possible to also update

the heuristic values of intermediate nodes. However, this would have the disadvantage of using much more memory, but it may reduce total path length.

Deadlocks in PR LRTS can be reduced by not using the nearest unabstracted node of the abstracted goal node as a subgoal.

When the nearest unabstracted child of an abstracted goal node is used as a subgoal, this node will be a very common intermediary node. This may cause additional congestion around such nodes.

Instead, the corridor can be cut off at the abstract node right before the abstract goal node, and the abstracted goal node can be kept as goal node. In this case, the node that is nearest to this goal node has to be calculated, as the abstract goal node won't be reached.

A problem with this approach and a better solution was discovered: the full corridor will be searched when the goal is not part of the corridor. A better solution would be to search to any node that is a child of the abstracted goal. The implementation of an A* search in a corridor in HOG does precisely this.

3.4 Theoretical analysis

3.4.1 LRTS

Computational cost

The computational cost depends mainly on lookahead. If occupied cells are treated specially, two searches are performed, and this doubles the computational cost that comes from lookahead.

A generated path may not be traversable, as no reservation table is used, thus in the worst case the second node in the path will be blocked, and the path will have to be recalculated every second time step.

Using an octile map and a Dijkstra search with a maximum depth or lookahead d, at most $n = (1 + 2(d-1))^2 = 4d^2 - 4d + 1$ nodes are expanded, but all eight neighbouring nodes of them are touched. Handling occupied nodes specially simply doubles expanded and touched nodes. Updating intermediate nodes can be modeled as adding a cost of d to the touched nodes. d is the maximum number of intermediate nodes that need to be updated.

As a modified Dijkstra search with a maximum depth and a binary heap is used, it takes $O(\log n)$ time to expand a node, where n is the maximum number of nodes in the heap. Time is thus bounded by $O(n \log n)$.

Maximum memory usage

This is the sum of the size of the map (worst case memory usage of the stored heuristic) and the worst case of the memory used for the closed and open list, which is $(1+2d)^2 = 4d^2+4d+1$ entries.

Examples

Lookahead	Nodes expanded	Memory usage from open & closed lists
1	1	9
4	49	81
8	225	289
12	529	625
16	961	1089
20	1521	1681
24	2209	2401
28	3025	3249
32	3969	4225
36	5041	5329
40	6241	6561

3.4.2 PR LRTS

Computational cost

PR LRTS incorporates the computational cost of LRTS, but in addition the computational cost of refining a path in the corridor is added.

A path in a corridor has to be calculated between 1 and an infinite number of times, and this may happen at every second time step, as a calculated path may become blocked when other agents have moved before the agent takes its second step on the calculated path.

With map clique abstraction, at most 4 nodes can be abstracted into one, unless adoptees are allowed. I suspect that it may be possible to abstract arbitrarily many nodes into one when adoptees are allowed, and if so, map clique abstractions with adoptees are not suitable for real-time algorithms. However, it may be possible to put an upper bound on it anyway; I don't feel qualified to determine this.

In this theoretical analysis, I will restrict myself to map clique abstractions without adoptees, and thus an abstracted node may be an abstraction of at most 4^a nodes, where a is the corridor abstraction level.

In the absolute worst case, all nodes in a corridor will be expanded, and as usual, all neighbouring nodes of the expanded nodes are touched.

A wider corridor adds to the number of nodes in the corridor, by the following amounts, where n is the original amount:

Corridor width	Fix single connectivity?	Maximum amount of nodes in corridor
0	Yes	$3n - 2$
1	No	$5n + 4$
1	Yes	$6n + 6$
2	No	$8n + 20$
2	Yes	$8n + 26$

These worst-case figures are obtained for diagonal corridors.

Memory usage

The memory usage of PR LRTS is the LRTS memory (lower than pure LRTS due to abstraction level) plus memory used by the corridor and memory used for the open and closed lists in the corridor.

3.4.3 CPR LRTS

Computational cost

As with PR LRTS, it incorporates the computational cost of LRTS, but in addition the computational cost of refining a path in the corridor is added.

A path in a corridor has to be calculated between $\lceil c/w \rceil$ times and an infinite number of times, where c is the minimum corridor length and w is the window size. As long as no agents which are outside our control are moving on the map and the map doesn't change, a reserved path will always be valid. Otherwise reserved paths may very well be blocked when it's time to traverse them, and the current path may have to be recalculated as often as every second time step.

In the absolute worst case, all nodes in the corridor will be expanded by the reverse A* heuristic. However, if abstraction is used for the reverse A* heuristic, the computational cost for calculating it can be cut down significantly. The number of nodes inside the corridor is guaranteed to be at least almost halved between two neighboring levels of abstraction, as intermediate abstracted nodes are guaranteed to be at least 2-cliques.

In addition, nodes within the specified window size will be expanded by the temporal-spatial A* search. $\frac{(2w+1)(w+1)(2w+3)}{3}$ nodes will be expanded by the temporal-spatial search, where w is the window size.

Memory usage

The memory usage of CPR LRTS is the LRTS memory (lower than pure LRTS due to abstraction level) plus the memory used by corridor, the memory used for the open/closed list in the corridor and the memory used for the reverse A* heuristic.

4 Implementation & measuring

4.1 Implementation

The implementation was based on Nathan Sturtevant's HOG (Hierarchical Open Graph[14]) framework. LRTS, PR LRTS and CPR LRTS were implemented in it by me. PRA*, CPRA* and WHCA* implementations were provided by Sturtevant, and were used as comparisons.

For all the algorithms, direction maps can be used.

4.1.1 LRTS

LRTS was implemented as described in the paper[3], with weighting, lookahead and learning amount as parameters.

In addition, it can be specified whether to:

- Treat occupied nodes specially? This option is called *handleOccupied*.
- Update heuristic values of intermediate nodes between the current node and the one that gave the updated heuristic value? This option is called *updateIntermediate*.
- Always greedily go towards the goal, if it is found within the expanded nodes? This option is called *greedyGoal*.

4.1.2 PR LRTS

PR LRTS uses LRTS and thus inherits the parameters available for LRTS, except that treating occupied nodes specially won't have any effect, as abstracted nodes are never marked as occupied.

In addition, the following parameters are available:

- Abstraction level of corridor;
- Should corridor A* fall back to nearest node found if goal cannot be reached in the current corridor? If it doesn't and the goal cannot be reached, the agent will simply stop and wait until it can. This option is called *fallbackToNearest*.
- Should the corridor A* search to the node closest to average node that is a child of abstracted goal, or just to any child of the abstracted goal? This option is called *searchToMiddleChild*.
- Should single connectivity be fixed in the corridor?
- How many nodes around the corridor should be part of the corridor? That is, what corridor width should be used?
- Should a shared distance heuristic be used?

4.1.3 CPR LRTS

All parameters that are available for PR LRTS are available for CPR LRTS, except whether to search to the middle node of the abstracted goal or not. CPR LRTS will always do this, as the WHCA* search needs to search towards a specific node. Also, falling back to the nearest node available doesn't make sense for WHCA*, as a windowed temporal-spatial search doesn't require the goal to be reachable.

In addition, the following parameters are available:

- WHCA* window size;
- Abstraction level of the reverse A* heuristic (also called the WHCA* abstraction level);
- Should the corridor be restricted to nodes between the current and what is needed for WHCA* window size? This option is called *restrictCorridor*.

4.2 Measuring

The following aspects have been measured, among others:

- Average & maximum nodes expanded & touched (this is approximately the computational cost needed);
- Average & maximum memory usage;
- Average & maximum distance travelled (of agents that reached their goal);
- Percentage of agents that reached their goal.

Path quality can be measured and quantified in several ways. I've chosen to primarily look at the maximum distance travelled and the percentage of agents that reached their goal. The maximum distance travelled indicates the time it took to solve the problem, when all agents did reach their goal.

As it is not possible to know whether all agents will reach their goal or not, a deadline had to be introduced. This deadline was somewhat arbitrarily set to 4 times the map width, in this case $128 * 4 = 512$ steps. If an algorithm is well-behaved, it should be able to deliver a solution within this time limit, for the maps that were used. This deadline also serves to decrease the time taken to run the tests; it would not be feasible to let the tests run with a very long deadline.

When some agents fail to reach their goal, all other measurements may be affected by this, and that may cause them to not be comparable. For example, the algorithms either do not perform any more calculations or perform fewer calculations after an agent has reached its goal, thus skewing average computational cost and memory usage. Also, as the pathfinding problems assigned to the agents may vary widely in how hard they are, only looking at the distances travelled of a fraction of the agents may skew the results. However, maximum computational cost and memory usage is unaffected by the percentage of agents that reached their goals, unless very few agents did.

Of the measurements, the maximum nodes expanded, maximum memory usage and maximum path length and percentage of agents that reached their goals are by far the most interesting from a real-time pathfinding perspective, although the average measurements are also interesting. As touched nodes correlate closely with expanded nodes and expanded nodes require more computational power to process, the expanded nodes measured are primarily what is examined out of these two measurements.

To test the shared distance heuristics and direction maps thoroughly, several runs will have to be run, where agents are removed from the map after every run, but the stored data is kept. Also, the same trials have to be run without shared distance heuristics or direction maps, to be able to evaluate the results fairly.

4.3 Maps used for measurement

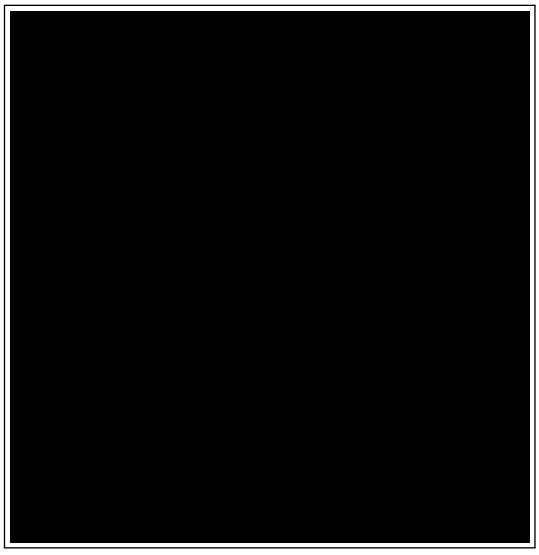
I have used 4 maps, all provided by HOG. In the map graphics below, white means untraversable, while black and gray are traversable. Agents cannot have a start or a goal which is on a grey tile, however.

The maps were scaled to 128×128 before being used. Larger maps were not used, as the tests already took a long time to run. All the maps are octile maps; that is, agents may move in all 8 directions (straight and diagonal directions) between nodes.

More complicated maps, as for example mazes, were not measured, as they were not felt to represent real world problems.

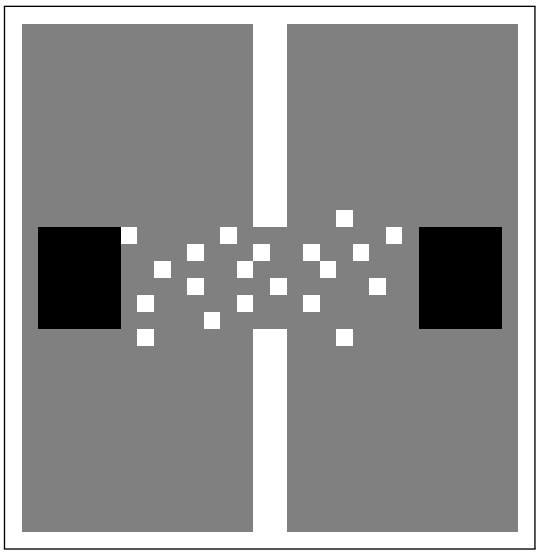
empty.map

This is a 100×100 empty map which tests basic operation of algorithms. This is the easiest map used.



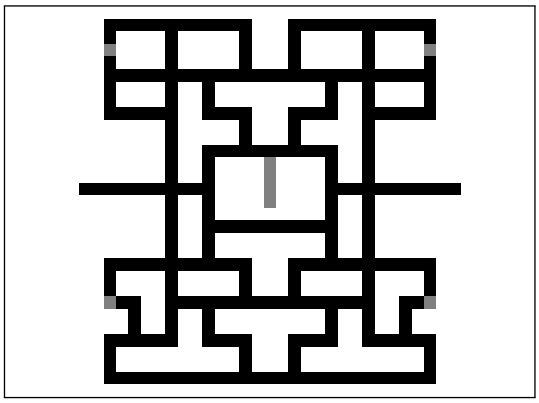
test_s3.map

This 32×32 map tests cooperativity by having two areas which agents travel between, and where the path between the areas has random obstacles in it.



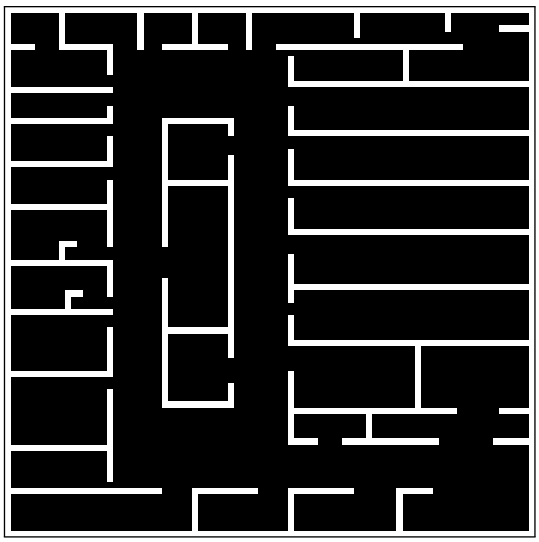
pacman.map

This is a 43×31 pacman-like map with corridors which are just a few tiles wide. This challenges cooperativity.



CSC2F.map

This is a 88×86 map with room structures and some bottleneck areas which cause local minima in the heuristic function. This is the hardest map tested.



4.4 Tests performed

When not otherwise noted, all combinations of the parameters mentioned were tested.

All tests were performed with 32, 64 and 128 agents, and on all 4 maps. That is, for every combination of algorithm and algorithm parameters measured, 12 unique tests were performed. If all agents reach their goal for all 12 tests, an algorithm is considered to be well-behaved, and solves all problems presented.

When a test is started, all agents are assigned random starting positions and goals, with the restriction that the starting position is required to be in the left half of the map and the goal in the right half, or vice versa. Note that to make test data comparable, all agents in all tests with the same map and number of agents (and the number of the run, for the tests that test direction maps or shared distance heuristics) are assigned the same problems.

The simulation is then run until all agents have reached their goals, or until 512 time steps have passed. Data is collected while the tests are running.

All tests took approximately 2 days to run on an Intel Core i5 750 (4 cores at 2.67 GHz each, all utilized). The test data amounted to measurements for 88416 tests in the end, and was imported into a MySQL database to make analysis easier. Specifically, to summarize results from several tests.

To summarize average values, the SQL function avg() was used in conjunction with GROUP BY. For maximum values, for example maximum nodes expanded, either avg() or max() was used, depending on the situation. If all values were measurements of the same algorithm with the same parameters, max() was typically used. Otherwise, avg() was used. As an exception to this was when summarizing maximum distance travelled, as taking the maximum of these wouldn't say much about how the algorithm performed on other maps than the hardest one. Instead, the average of the maximum distance travelled was used.

4.4.1 LRTS

- Tried different amounts of lookahead (1, 4, 8, 12 ... up to 40 in steps of 4) and γ (0.3, 0.7, 1.0);
- Toggled handleOccupied/updateIntermediate/greedyGoal.

4.4.2 PR LRTS

For PR LRTS, two separate runs were performed: One that focused on testing a diversity of on/off options, and one that focused on testing how LRTS lookahead affects the algorithm. Also, LRTS γ was fixed at 1.0, and other LRTS parameters were also kept constant: updateIntermediate at false, and greedyGoal at true (handleOccupied was kept at false, but this parameter doesn't affect PR LRTS). This was to keep the number of combinations down to an acceptable level, to allow the tests to complete within a reasonable amount of time.

Run 1

- LRTS lookahead was kept at 16;
- Different abstraction levels (1 - 4) were tried;
- Different corridor parameters (width: 0, 1, 2; toggle fixSingleConnectivity) were tried;
- searchToMiddleNode and fallbackToNearest were toggled.

Run 2

- LRTS lookahead of 4 to 12 in increments of 4 (16 was already tried in run 1 above) were tried;
- Different abstraction levels (1 - 4) were tried;
- Different corridor parameters (width: 0, 1; toggle fixSingleConnectivity) were tried;
- searchToMiddleNode was fixed at false and fallbackToNearest was fixed at true.

4.4.3 CPR LRTS

The CPR LRTS tests were similar to the PR LRTS tests in that two separate runs were performed.

Note that the WHCA* abstraction level is required to be below the corridor abstraction level, thus combinations where this wasn't the case were skipped.

Also, WHCA* was found to behave erratically when using a too low window size with a too high WHCA* abstraction level, so those combinations were also skipped. When the

window size was not big enough to reach the neighbouring abstracted nodes in the reverse A* heuristic, WHCA* would often prefer to stay at the current position rather than move, as no better node than the current one was found.

This was the case when the window size was below 2^a, where a is the abstraction level.

Run 1

- LRTS lookahead was kept at 16;
- Different corridor abstraction levels (1 - 4) were tried;
- Different corridor parameters (width: 0, 1, 2; toggle fixSingleConnectivity and restrictCorridor) were tried;
- WHCA* abstraction levels of 0 - 4 were tried;
- Different WHCA* window sizes (4 - 20 in increments of 4) were tested.

Run 2

- LRTS lookahead of 4 to 12 in increments of 4 (16 was already tried in run 1 above) were tried;
- Different corridor abstraction levels (1 - 4) were tried;
- Different corridor parameters (width: 0, 1; toggle fixSingleConnectivity) were tried;
- restrictCorridor was fixed at false;
- WHCA* abstraction levels of 0 - 4 were tried;
- Different WHCA* window sizes (4 - 20 in increments of 4) were tested.

4.4.4 Direction maps and shared distance heuristics

Direction maps (DM) were tested with LRTS, PR LRTS and CPR LRTS. Shared distance heuristics (SDH) were tested with PR LRTS and CPR LRTS.

A number of runs were made for each algorithm and parameter combination, to see if and how much of an improvement was made over successive runs, with data being collected over all runs. Note that a specific run number has the exact same search problem as other runs with the same run number, and that this search problem is different from that of other runs.

For direction maps, an α of 0.2, 0.5 and 0.7 was tested, and a w_{max} of 10.0 was tested. I intended to also test a w_{max} of 5.0 and 20.0, but due to an unfortunate bug in the code which was found too late, this was not done.

Shared distance heuristics do not have any additional parameters that can be altered, thus performing more runs than with direction maps was feasible.

LRTS

- Lookahead of 4, 12, 20 was tried;
- γ of 0.3, 1.0 was tried;
- 4 runs per parameter combination was run for DM.

PR LRTS

- A corridor abstraction level of 1 - 4 was tested;
- searchToMiddleChild was fixed at false, fallbackToNearest was fixed at true, fixSingle-Connectivity was fixed at true;
- For abstraction level 1, a corridor width of 2 was used, for the rest a corridor width of 1 was used;
- 8 runs per parameter combination was run for DM, 16 for SDH.

CPR LRTS

- A corridor abstraction level of 1 - 4 was tested;
- restrictCorridor was fixed at false, fixSingleConnectivity was fixed at true;
- For corridor abstraction level 1, a corridor width of 2 was used, for the rest a corridor width of 1 was used;
- WHCA* abstraction levels of 0 - 4 were tried;
- 4 runs per parameter combination was run for DM, 16 for SDH.

4.4.5 PRA*

PRA* was tested with window sizes of 4 to 10 in increments of 2.

4.4.6 CPRA*

CPRA* was tested with window sizes of 4 to 10 in increments of 2, and with and without an adaptive window.

4.4.7 WHCA*

WHCA* was tested with window sizes of 8 to 20 in increments of 4 and with an abstraction level between 0 and 4.

4.5 Comparing to previous results

LRTS and PR LRTS were measured, with the modifications described above to work in a cooperative environment. Also PRA* with local repair, CPRA* and WHCA* were measured. Unfortunately, memory usage reporting wasn't implemented in the PRA* and CPRA* implementations, so to compare with those algorithms, memory usage cannot be weighed into the results. However, computational cost often correlates with memory usage.

CPR LRTS can be compared to all of them, although it cannot be completely fairly compared to the non-real-time algorithms.

While the real-time algorithms have an absolute upper bound on them, specified by their parameters, the map size also influences the non-real-time algorithms.

The way that was settled on to compare the algorithms was to take the algorithm configurations that solved all problems and filter out those that were outside a specific amount of suboptimality for the maximum path lengthes. The remaining algorithms were sorted by a linear combination of the maximum nodes expanded, maximum nodes touched, maximum memory used and the average of the maximum distance travelled.

The algorithms were also compared by just sorting by one of the aspects of the algorithms, for example average maximum expanded nodes.

5 Experimental results & observations

Note that a large amount of tests were run, and that the results only can be presented in a heavily summarized form.

All the results can be downloaded as a compressed MySQL database dump. Where to download it and how to interpret it can be read about in Appendix A.

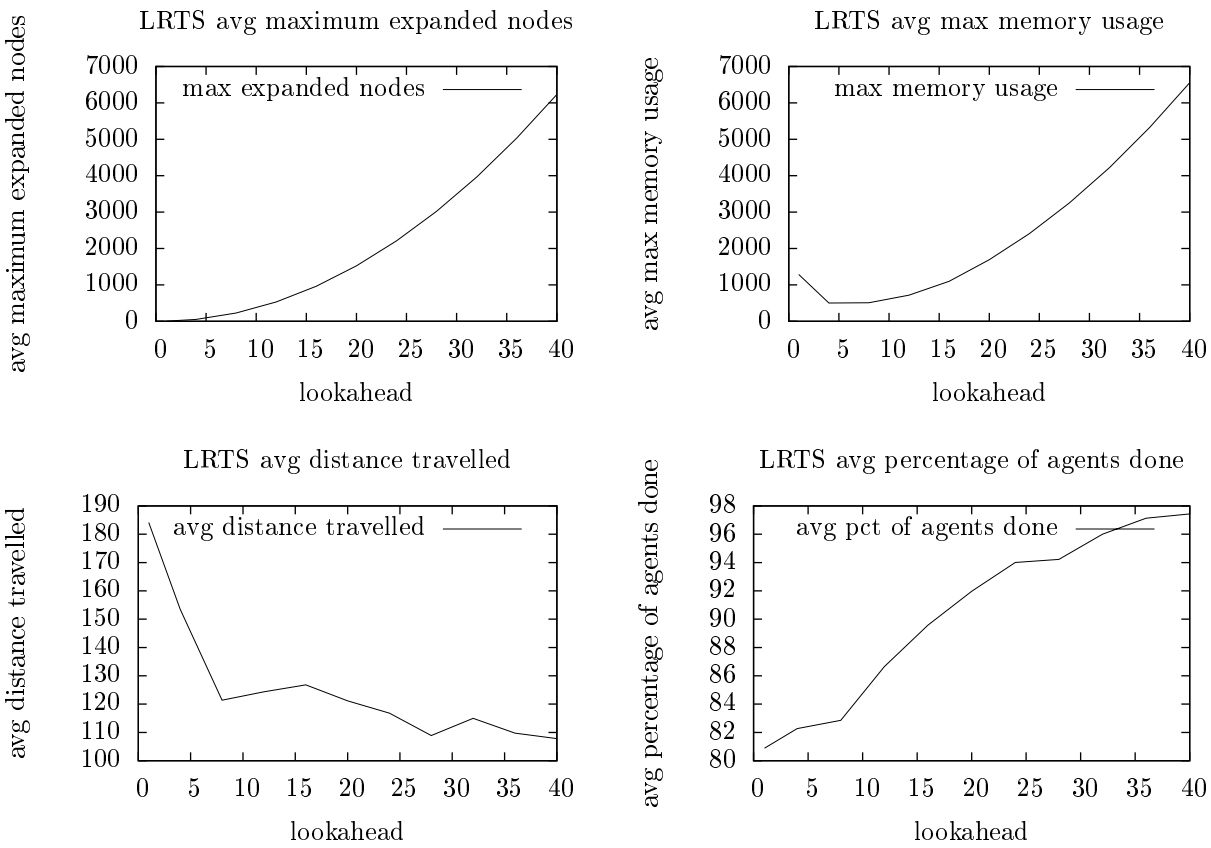
Appendix B contains the MySQL queries used to generate the summarized results seen in this chapter.

For graphing corridor widths, fixSingleConnectivity is considered to contribute a value of 0.5 to the corridor width when it is enabled. Thus when fixSingleConnectivity is enabled and a corridor width of 1 is used, this is graphed as if a corridor width of 1.5 was used.

An agent is 'done' if it reaches its goal within the deadline.

5.1 LRTS

Influence of lookahead



All the boolean parameters of the LRTS implementation were turned off.

The average distance travelled, instead of the average maximum distance travelled was measured. The latter measurement tended to be random, as all agents rarely reached their

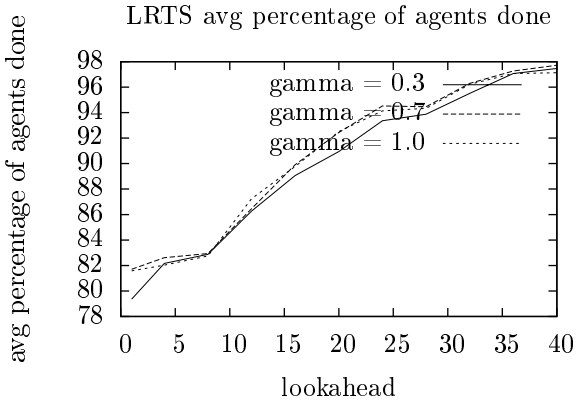
goals on the harder maps. Note that in the other graphs, the average maximum distance travelled was used.

A higher lookahead increases the percentage of agents reaching their goals and decreases distances travelled, but increases computational costs and memory usage. The exception is when going from a lookahead of 1 to a lookahead of 4; then memory usage decreases steeply. The likely reason is that with a lookahead of 1, many more heuristic values are stored in the hash map of heuristic values than with a lookahead of 4. With a higher lookahead than 4, the increase in memory usage due to deeper lookahead outnumbers the decreased memory usage due to fewer stored heuristic values, making memory usage always increase with increased lookahead.

LRTS never managed to solve all problems. Even a lookahead of 40 doesn't enable LRTS to solve all problems, but it gets closer the higher lookahead it uses.

It can be noted that the theoretically calculated maximum expanded nodes and maximum memory usage matches the empirically attained results.

5.1.1 Influence of weighting



As above, all boolean parameters were turned off.

Using a γ of 0.3 over a γ of 1.0 causes fewer agents to reach their goal in most cases. In the cases where it brings an improvement, the improvement is negligible.

Using a γ of 0.7 instead of 1.0 makes more agents reach their goals when a lookahead of 8 or less or 16 or more is used.

5.1.2 Influence of boolean parameters

γ= 0.3

greedyGoal	updateIntermediate	handleOccupied	avg(exactPercentDone)
0	0	0	90.859583
0	0	1	90.970167
0	1	0	91.126333
0	1	1	91.191500
1	0	0	90.859583
1	0	1	90.970167
1	1	0	91.113333
1	1	1	91.152583

γ= 0.7

greedyGoal	updateIntermediate	handleOccupied	avg(exactPercentDone)
0	0	0	91.464667
0	0	1	91.503750
0	1	0	91.094500
0	1	1	84.244917
1	0	0	91.464667
1	0	1	91.503750
1	1	0	91.862000
1	1	1	91.920750

γ= 1.0

greedyGoal	updateIntermediate	handleOccupied	avg(exactPercentDone)
0	0	0	91.321667
0	0	1	91.334750
0	1	0	91.068333
0	1	1	82.414833
1	0	0	91.315167
1	0	1	91.334750
1	1	0	92.258917
1	1	1	92.337167

The tests which were run with a lookahead of 1 were excluded from this test, as they skewed the results, and most of the boolean parameters had no effect on them.

As can be seen, a combination of updating intermediate nodes and handling occupied nodes is catastrophal when using a γ of 0.7 or 1.0.

If all three options are enabled, this creates an algorithm that performs well with all values of γ tested. Using all three options except handling occupied nodes, one gets an

algorithm that still performs very well, but which will use about half the computational cost of having all three activated.

I suggest using deeper lookahead instead of handling occupied nodes specially, as that will give a much bigger and guaranteed advantage.

lookahead = 1

greedyGoal	updateIntermediate	handleOccupied	avg(exactPercentDone)
0	0	0	80.881389
0	0	1	75.672222
0	1	0	80.881389
0	1	1	75.672222
1	0	0	80.881389
1	0	1	75.672222
1	1	0	80.881389
1	1	1	75.672222

With a lookahead of 1, only handling occupied nodes affects the results, and it consistently makes fewer agents reach their goals.

5.1.3 Influence of direction maps

Without direction maps

lookahead	gamma	avg(maxDistanceMovedDone)	avg(exactPercentDone)
4	0.3	138.439167	100.000000
4	1.0	136.445833	100.000000
12	0.3	138.454583	100.000000
12	1.0	137.108750	100.000000
20	0.3	139.082917	100.000000
20	1.0	137.810417	100.000000

With direction maps

lookahead	gamma	avg(maxDistanceMovedDone)	avg(exactPercentDone)
4	0.3	155.944167	100.000000
4	1.0	160.892639	100.000000
12	0.3	141.410556	100.000000
12	1.0	141.200139	100.000000
20	0.3	140.880972	100.000000
20	1.0	140.639167	100.000000

To get a fair measurement of distances travelled, measurements for the two harder maps ('CSC2F' and 'pacman') were excluded, leaving 'empty' and 'test_s3'. Thus all agents reached their goals.

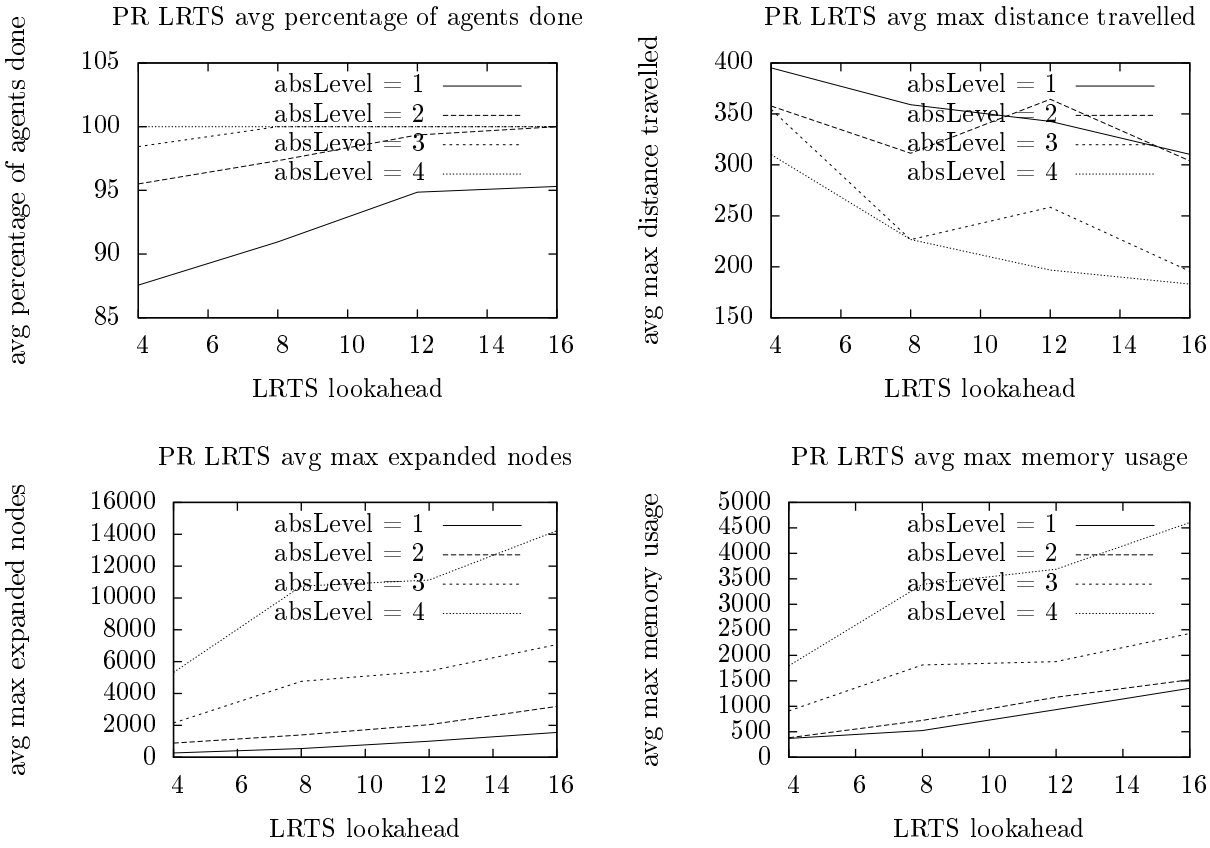
Direction maps consistently increase distances travelled of LRTS.

Note that the direction maps results were averaged over all tested parameter values of direction maps, but the results do show the general trend.

5.2 PR LRTS

In the following graphs, the corridor abstraction level is used as the x axis, but how the corridor abstraction level affects the algorithm will not be covered until later on.

5.2.1 Influence of lookahead



These results were created when using a corridor width of 1, with single connectivity-fixing turned off, searching to the middle child of the abstracted goal node turned off, and with falling back to the nearest node in the corridor turned on.

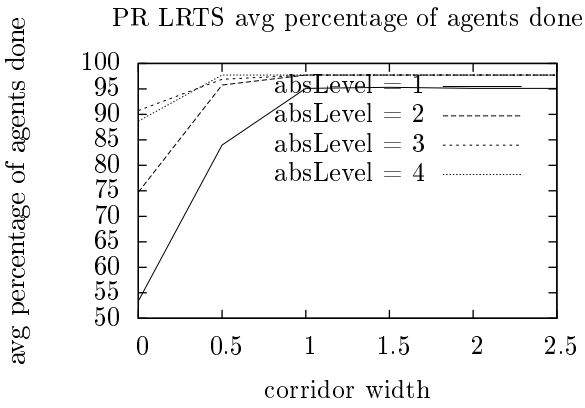
It can be seen that distances travelled generally decrease with higher lookahead. However for abstraction level 2 and 3, the distances travelled are increased when a lookahead of 12 is used, as compared to when a higher or lower lookahead is used.

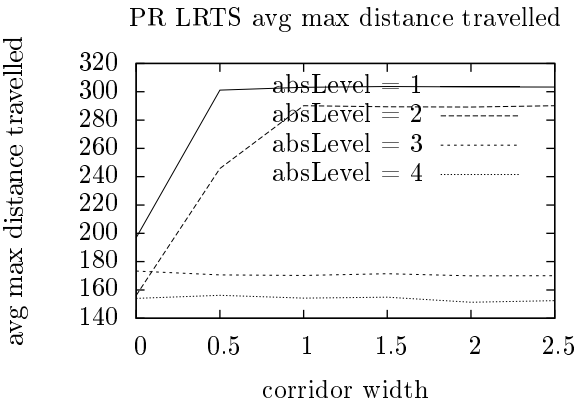
Also, it can be seen how computational cost and memory usage increases with higher lookahead.

5.2.2 Influence of boolean parameters and corridor width

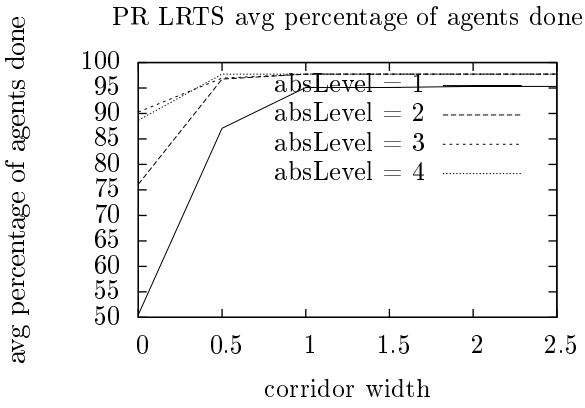
The following results were created with an LRTS lookahead of 16, searching to the middle child of the abstracted goal node turned off, and with falling back to the nearest node in the corridor turned on.

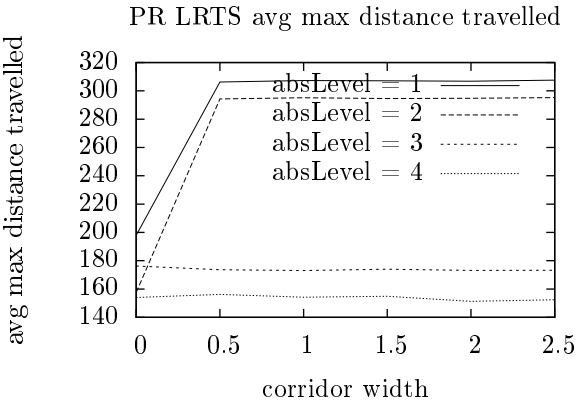
Falling back to nearest turned off, searching to middle child turned off



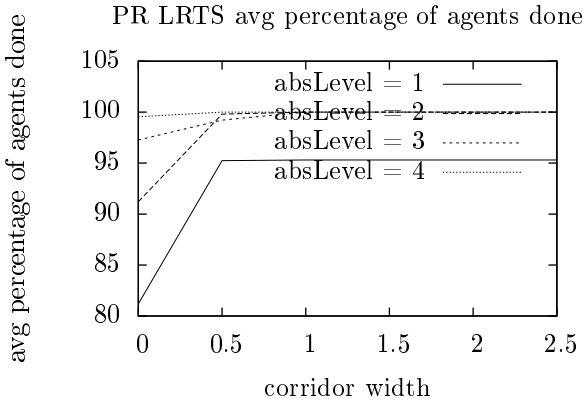


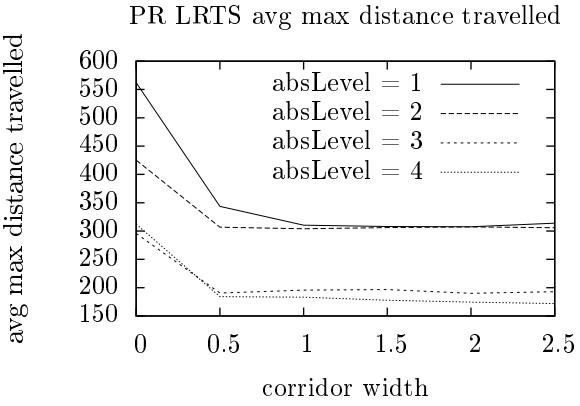
Falling back to nearest turned off, searching to middle child turned on



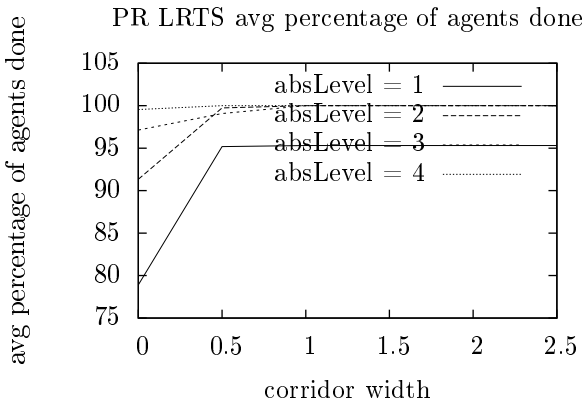


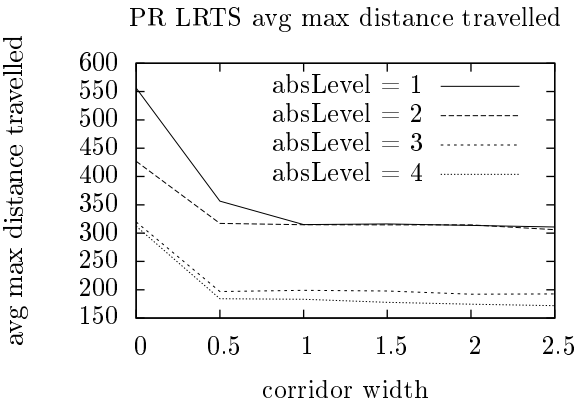
Falling back to nearest turned on, searching to middle child turned off



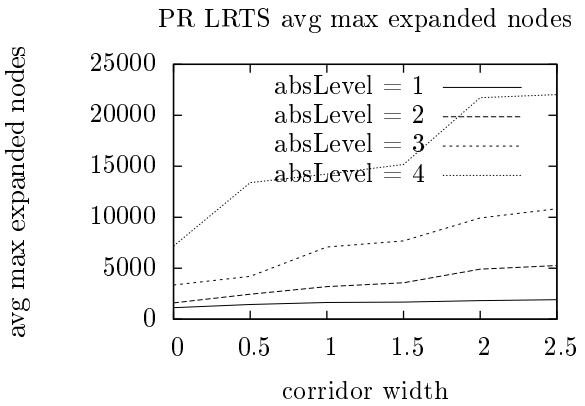


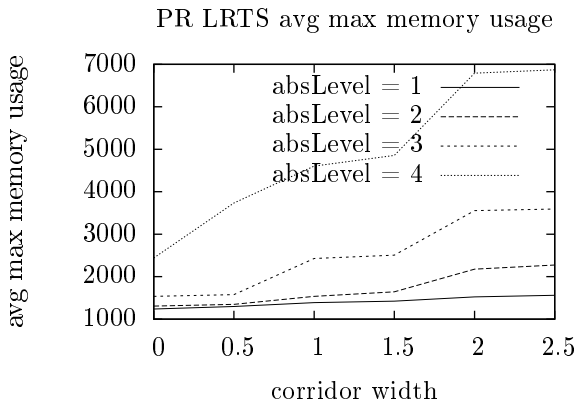
Falling back to nearest turned on, searching to middle child turned on





Computational cost and memory usage





When falling back to the nearest node is turned off, the algorithm performs drastically

worse than with it turned on. Specifically, PR LRTS never manages to solve all presented problems with it turned off, but turning it on enables it to do so.

Although it is hard to see in the graphs, searching to the middle child makes the algorithm perform a tiny bit worse.

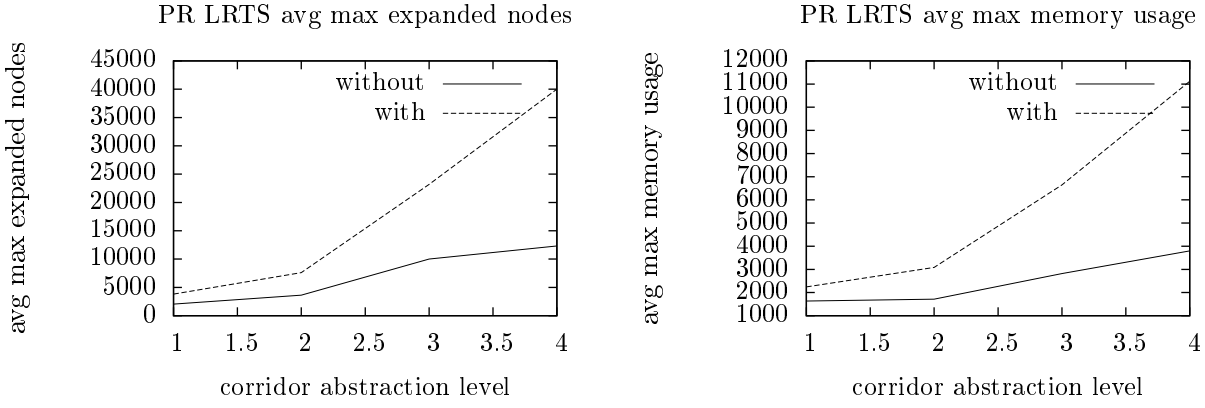
Note that how the algorithm performs with a non-widened corridor is not representative for how it performs with a widened one, and that distances travelled when all agents did not reach their goal should be taken with a big pinch of salt.

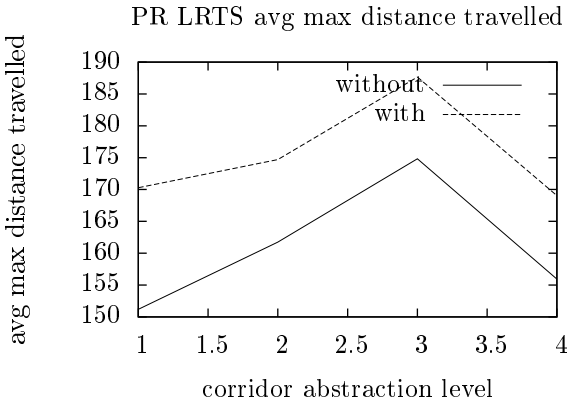
It can be seen clearly how wider corridors make the algorithm perform better, but also that a corridor wider than 1.0 will help relatively little, and that wider corridors correlate with larger computational costs.

5.2.3 Influence of abstraction level

As can be seen in the graphs in the preceding sections, what abstraction level is used plays a vital role in how well the algorithm will perform, but will also make the algorithm more computionally expensive and make it use more memory.

5.2.4 Influence of direction maps





To get a fair measurement of distances travelled, measurements for the two harder maps ('CSC2F' and 'pacman') were excluded, leaving 'empty' and 'test_s3'. Thus all or very close to all of the agents reached their goals.

Direction maps consistently increase the distances travelled.

Two other interesting observations was how direction maps also cause a drastic increase in computational cost and memory usage, and how the distance travelled is affected by the abstraction level used. I think the former is caused by A* re-expanding nodes due to inconsistency caused by direction maps, but I have no explanation for the latter.

Note that the direction maps results were averaged over all tested values of α and w_{max} for the direction maps, but the results do show the general trend.

5.2.5 Influence of shared distance heuristics

Shared distance heuristics did not make any difference for PR LRTS.

5.3 CPR LRTS

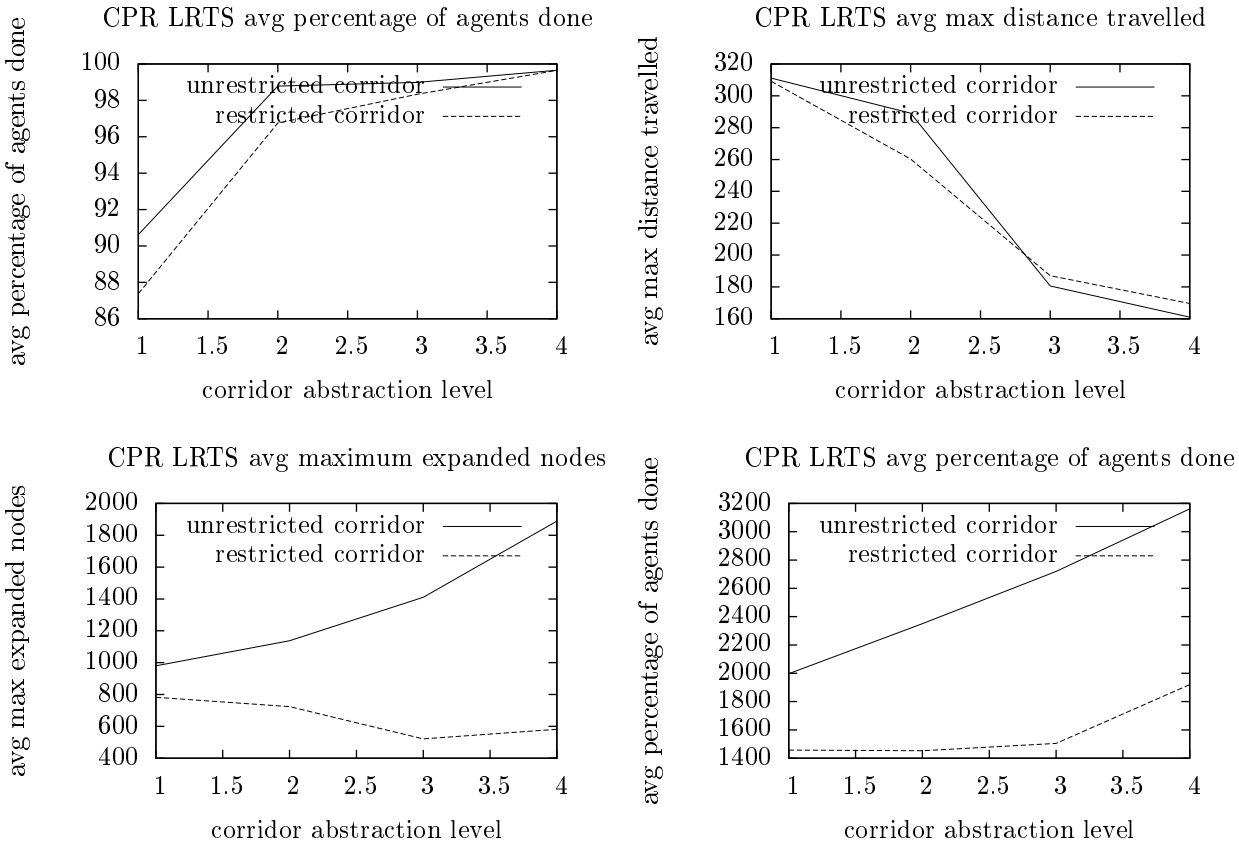
As CPR LRTS takes so many parameters, presenting results in graphs becomes quite hard. The corridor abstraction level, corridor width, WHCA* abstraction level and WHCA* window size all are critical to how well CPR LRTS works. There is no simple relation between the parameters and how the computational cost and memory usage is affected, and often, the minimum or maximum of a measurement might not correlate with the minimum or maximum value of a parameter. However, there is a general trend of how the different parameters affect computational cost and memory usage, and that is summarized in the graphs below. Do note, however, that these values are averages over many different trials where the other parameters varied wildly, and the relationships do not necessarily hold when zooming onto specific combination of parameters.

Averaging maximum distances travelled is not guaranteed to produce a valid summary when all agents did not reach their goals, and for some of the graphs, the effect of this can be clearly seen. Thus those graphs should be taken with a pinch of salt, especially when many agents didn't reach their goals.

In the graphs in the following sections, plots for how CPR LRTS performs both with and without restricting corridors are included. How restricting corridors affects the results is not covered in the text, this is covered in a section that follows it.

In all the tests below, an LRTS lookahead of 16 was used, except for the tests that tested LRTS lookahead.

5.3.1 Influence of corridor abstraction level



To enable a fair comparison of corridor abstraction levels, tests with a WHCA* abstraction level above or equal to 2 were filtered out, as such tests did not exist for all corridor abstraction levels.

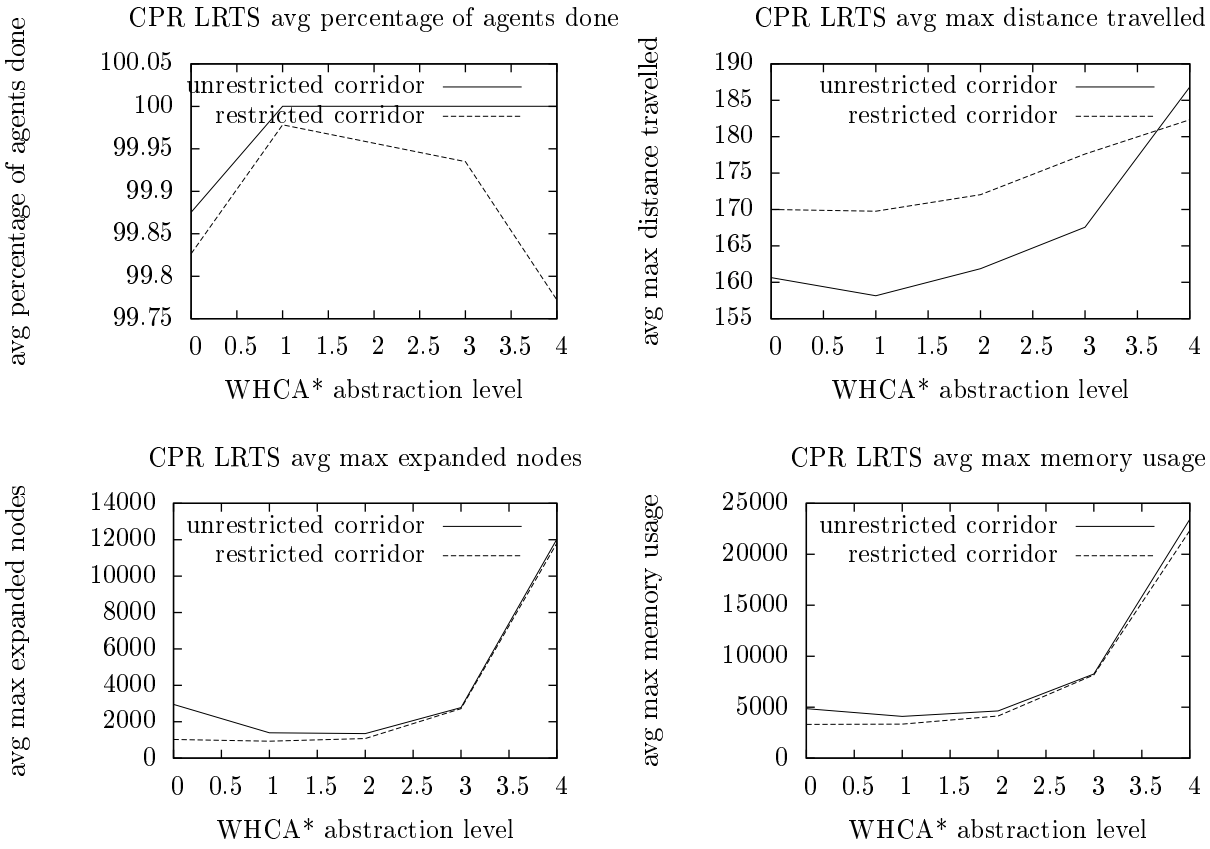
An abstraction level of 2 enables the algorithm to solve many more problems than with an abstraction level of 1. Using a higher abstraction level than 2 enables the algorithm to solve a few more problems, but the difference is far from as dramatic as between abstraction level 1 and 2.

There is however a dramatic difference between abstraction level 2 and 3 when looking at the distance travelled by agents, where abstraction level 3 has a big advantage compared to lower abstraction levels. Abstraction level 4 improves on this slightly, but the influence is far lower.

A higher abstraction level incurs a large penalty in computational cost and memory

usage, thus abstraction level 2 or 3 are probably the optimal choices.

5.3.2 Influence of WHCA* abstraction level



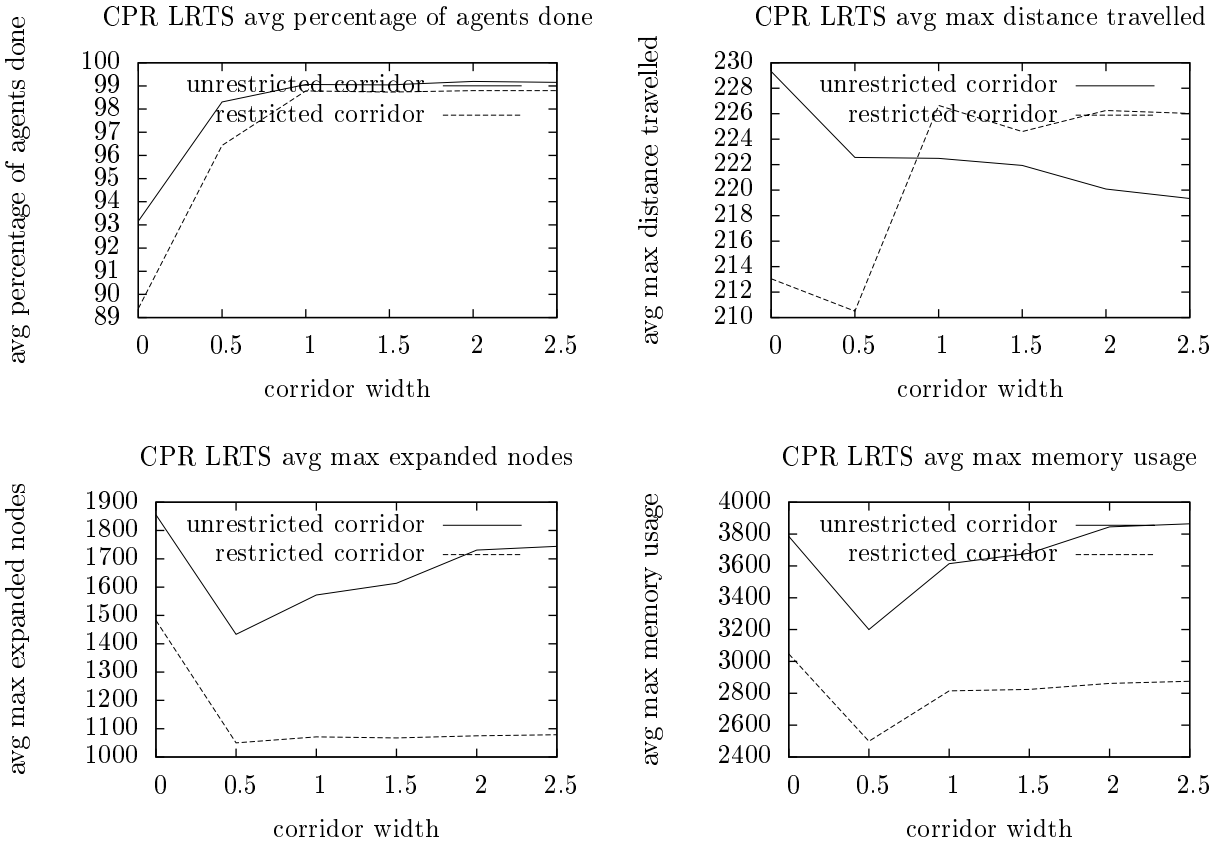
To enable a fair comparison, tests that did not use a corridor abstraction level of 4 and a WHCA* window size of 16 or above were filtered out; such tests did not exist for all WHCA* abstraction levels.

The WHCA* abstraction level barely affects how many agents reach their goals.

The distance travelled is however affected negatively by a high WHCA* abstraction level, as expected. Curiously, the minimum distance travelled is seen when the WHCA* abstraction level is 1 and not 0, but this may just be a statistical anomaly, as I find no plausible explanation for it.

Computational cost and memory usage benefit from a WHCA* abstraction level of 1 and 2, but higher values than that causes them to both become larger again.

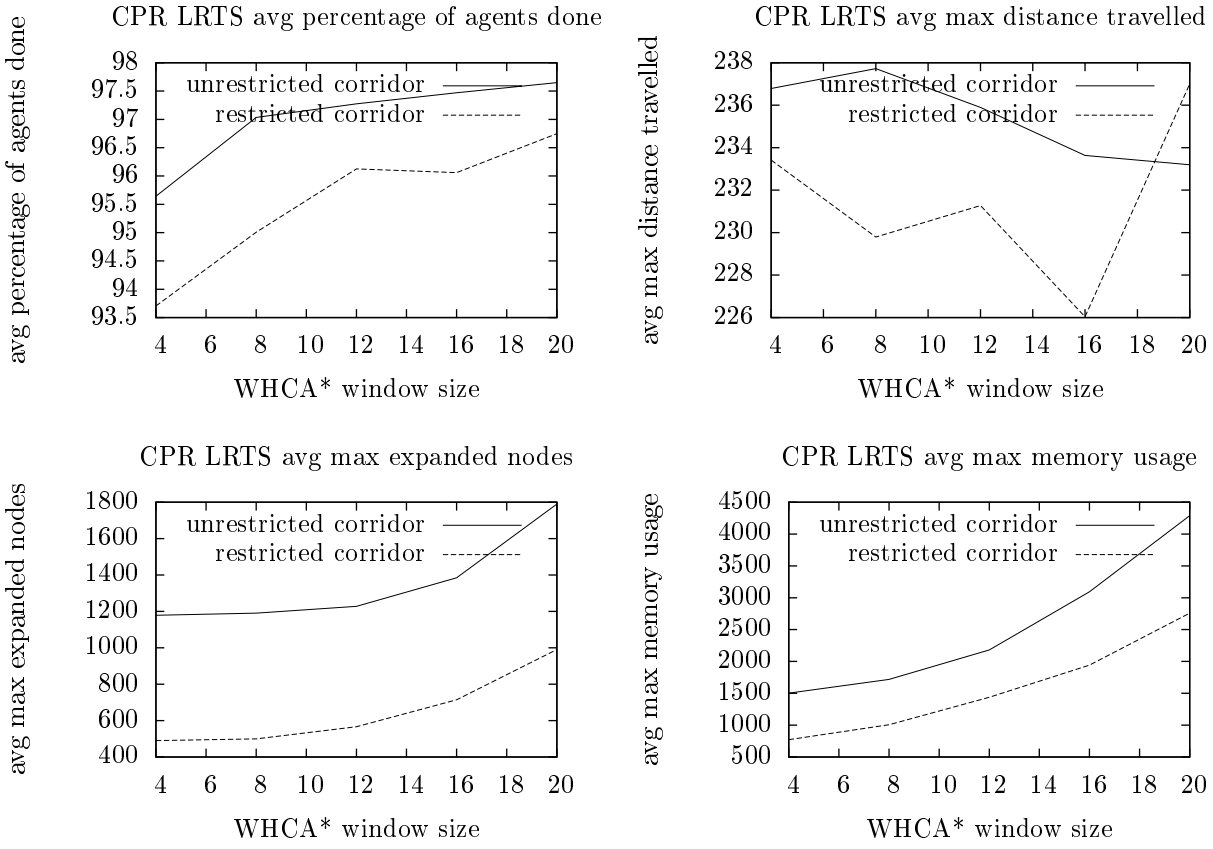
5.3.3 Influence of corridor width



As with PR LRTS, wider corridors give better path lengthes and enable more agents to reach their goal, but also increases computational costs and memory usage. Interestingly, a non-widened corridor also increases both computational costs and memory usage; I assume this is caused by the increased amount of bottlenecks.

The corridor parameters needed for CPR LRTS to solve all problems depend upon all of the the corridor abstraction level, the WHCA* abstraction level and the WHCA* window size.

5.3.4 Influence of WHCA* window size

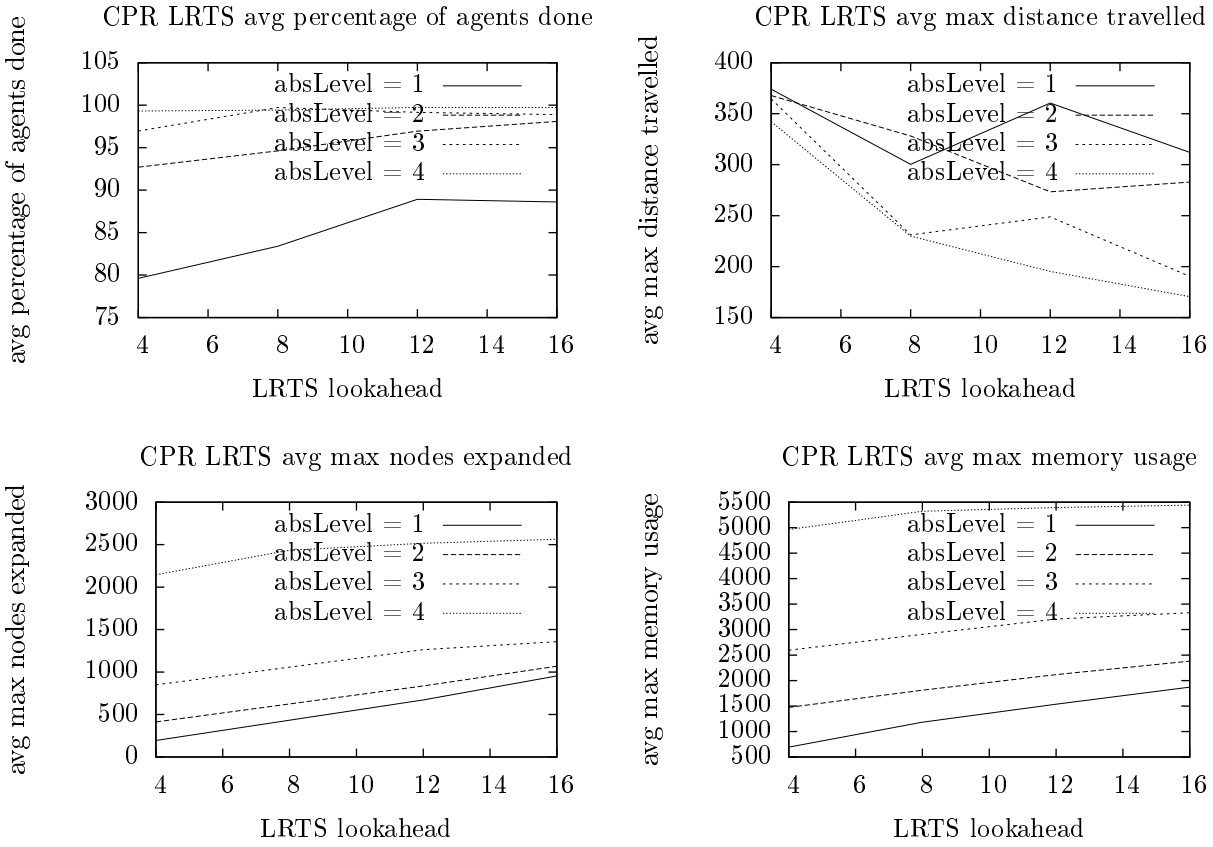


To enable a fair comparison, only tests with a WHCA* abstraction level of 1 were included, as other WHCA* abstraction levels do not have tests for all WHCA* window sizes.

A larger window size enables slightly more problems to be solved.

A larger window size also decreases path lengthes slightly and increases computational cost and memory usage.

5.3.5 Influence of LRTS lookahead

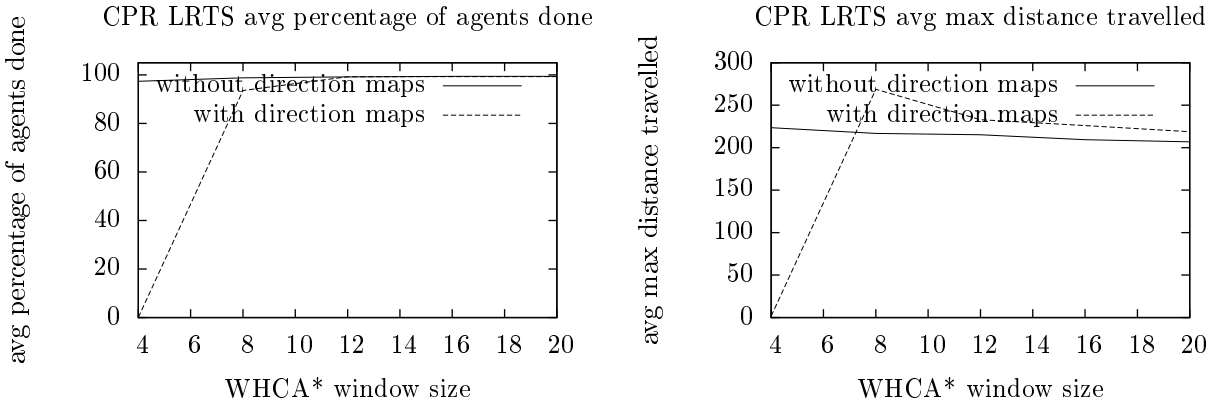


The LRTS lookahead used plays a big role in the amount of agents that will reach their goals. However, with a higher corridor abstraction level, this effect becomes smaller.

It also affects the distances travelled drastically. Usually, higher LRTS lookahead means lower distances travelled, but at corridor abstraction level 1 and 3, a LRTS lookahead of 12 produces much longer distances travelled than an LRTS lookahead of 8 or 16. I cannot explain this.

A higher LRTS lookahead also causes higher computational costs and higher memory usage.

5.3.6 Influence of direction maps



As with PR LRTS, direction maps consistently increase the distances travelled of CPR LRTS and decreases the number of problems solved.

However, with low WHCA* window sizes, the addition of direction maps makes CPR LRTS perform horribly, so that very few of the agents reach their goals. The likely reason for this is that the penalty for moving is too large, and agents thus decide to stay at their current position instead of moving, but it is not clear why this only happens with a low WHCA* window size.

Note that the direction maps results were averaged over all tested parameter values for the direction maps, but the results do show the general trend.

5.3.7 Influence of shared distance heuristics

As with PR LRTS, shared distance heuristics do not make any statistically significant difference.

5.3.8 Influence of restricting corridors

Restricting corridors often decreases the percentage of agents reaching their goal and always slightly increases average maximum distances travelled, while decreasing computational costs and memory usage. In the cases where it can be used while all agents still reach their goals, it performs very well, relative to its computational costs and memory usage.

Sometimes, the parameters which enable this follow a somewhat strange pattern. To enable all problems to be solved, the WHCA* abstraction level and window size needs to be one of the following when using an abstraction level of 2:

WHCA* abstraction level	WHCA* window size
0	20
1	4 or 20
2	4, 12 or 20

It cannot be excluded that this cannot be a bug in the code. The same pattern does not show up for a corridor abstraction level above 2, where the algorithm behaves more sensibly. Also worth noting is that fewer problems overall are solved when corridor restriction is used.

5.4 Comparing the algorithms

In the tables below, algorithms and parameter combinations that did not solve all presented problems were filtered out. Thus LRTS is not included, as it did not solve all presented problems with any combination of parameters it was given.

The algorithm parameters are shown within parentheses after the algorithm name. To make sense of them, follow this key:

- ***WHCA**** *(window size, abstraction level)*
- ***CPRA**** *(window size, adaptive window?)*
- ***PRA**** *(window size)*
- ***PR LRTS*** *(lookahead, corridor abstraction level, corridor width, fall back to nearest?, search to middle child?)*
- ***CPR LRTS*** *(lookahead, corridor abstraction level, WHCA* abstraction level, WHCA* window size, corridor width, restrict corridor?)*

As CPR LRTS with and without corridor restriction behaves so differently, both will be listed.

5.4.1 Best algorithm by travelled distance

The best algorithm when looking at travelled distance is WHCA*(12, 1) with an average maximum distance travelled by the agents of 147.54:

Algorithm	Avg max distance travelled	Max expanded nodes	Max memory usage
WHCA*(12, 1)	147.54	1984	2788
CPRA*(10, no)	149.36	4172	Unknown
PRA*(10)	149.46	1032	Unknown
CPR LRTS(16, 4, 1, 12, 2.0, no)	149.71	1781	2784
CPR LRTS(16, 4, 1, 12, 0.5, yes)	163.18	960	4157
PR LRTS(16, 1, 2.5, yes, any)	172.02	22017	6863

5.4.2 Best algorithm by memory usage

As mentioned earlier, PRA* and CPRA* cannot participate in this test.

Algorithm	Avg max distance travelled	Max expanded nodes	Max memory usage
CPR LRTS(4, 4, 2, 4, 0.5, no)	339.74	243	633
CPR LRTS(16, 4, 2, 4, 1.0, yes)	173.63	258	667
PR LRTS(4, 4, 0.5, yes, no)	316.24	4230	1234
WHCA*(8, 2)	161.30	762	1565

As seen above, CPR LRTS outperforms the other algorithms when it comes to maximum memory usage, and even manages to attain a very good average maximum distance travelled with corridor restriction enabled.

5.4.3 Best algorithm by expanded nodes

Algorithm	Avg max distance travelled	Max expanded nodes	Max memory usage
CPR LRTS(4, 4, 2, 4, 0.5, no)	339.74	243	633
CPR LRTS(16, 4, 2, 4, 1.0, yes)	173.63	258	667
WHCA*(8, 2)	161.30	762	1565
PRA*(4)	151.12	967	Unknown
CPRA*(4, no)	155.36	1037	Unknown
PR LRTS(4, 4, 0.5, yes, no)	304.02	3188	1519

Interestingly, the same two CPR LRTS algorithms come out in the top as in the memory usage test, and after that comes the best WHCA* algorithm in the memory usage test. PR LRTS ends up at the bottom, long after the other algorithms.

5.4.4 Best algorithm by weighted combination of characteristics

The tests above rank algorithms by one specific characteristic. It is typically preferred that an algorithm is reasonably good in many aspects.

If a 20% suboptimality (as compared to 147.54 as obtained by WHCA*) in average maximum distance travelled is allowed, and the expanded nodes, nodes touched, memory usage, and average maximum distances travelled are weighted with the following function:

```
score = travelDistance * 10 + nodesExpanded * 10 + nodesTouched + memoryUsage
```

then one gets the following results:

Algorithm	Avg max distance travelled	Max expanded nodes	Max memory usage
CPR LRTS(16, 4, 2, 4, 1.0, yes)	173.63	258	667
CPR LRTS(16, 4, 2, 4, 0.5, no)	169.31	416	815
WHCA*(8, 2)	161.30	762	1565
PR LRTS(16, 4, 2.0, yes, any)	304.02	21721	6521

The number of nodes touched were not included in the table, to preserve space.

The CPR LRTS algorithm that came out second in the memory usage and expanded nodes test still performs very well. PR LRTS is still at the bottom, and WHCA* keeps in

the middle.

If the weighting function is modified to not include the memory usage, PRA* and CPRA* can be included in the ranking:

```
score = travelDistance * 10 + nodesExpanded * 10 + nodesTouched
```

Algorithm	Avg max distance travelled	Max expanded nodes	Max memory usage
CPR LRTS(16, 4, 2, 4, 1.0, yes)	173.63	258	667
CPR LRTS(16, 4, 2, 4, 0.5, no)	169.31	416	815
WHCA*(8, 2)	161.30	762	1565
PRA*(4)	151.12	967	Unknown
CPRA*(4, no)	155.36	1037	Unknown
PR LRTS(16, 4, 2.0, yes, any)	304.02	21721	6521

PRA* and CPRA* were simply placed in between WHCA* and PR LRTS, and the ordering or best parameters for each algorithm did not change.

5.5 Influence of map size upon the algorithms

As mentioned above, 128x128 maps were used, for performance reasons.

It turned out that this may be too small to fairly measure PR LRTS and CPR LRTS, as at a high abstraction level, there won't be many nodes in the abstracted graph. Specifically, the maximum length of an abstracted corridor will be lower than the LRTS lookahead is. Thus the worst case computational cost and memory usage is not always experienced in the trials above, as both the A* and WHCA* search won't reach their theoretical maximum.

Also, as LRTS expands nodes in all directions and does not focus search towards the goal, it needs a map that is at least twice as wide or high as the lookahead used is, thus LRTS needs a twice as large maximum guaranteed corridor length to reach its theoretical maximum computational cost and maximum memory usage.

The maximum guaranteed length of an abstracted corridor is $128/2^a$,where a is the corridor abstraction level, as on an empty map, 2×2 nodes will be abstracted into 1 node between two neighboring abstraction levels.

Corridor abstraction level	Maximum guaranteed corridor length	Parts of (C)PR LRTS influenced
0	128	None
1	64	None
2	32	None
3	16	LRTS is affected
4	8	LRTS & A*/WHCA* affected

As can be seen, PR LRTS and CPR LRTS corridor abstraction level 3 and 4 may have had unfair advantages in the tests that were run. How large this advantage was is unknown.

To find out, the tests should be re-run with at least a 512x512 map.

Notable is that all of PRA*, CPRA* and WHCA* also are likely to have been affected by this low map size. As they aren't real-time, increasing the size of the map unevitably

also increases the maximum computational cost and memory usage of them.

6 Conclusions and future work

6.1 Conclusions

CPR LRTS is a very competitive algorithm in a coperative environment, both when looking at maximum computational costs and maximum memory usage. It is even competitive when comparing it with non-real-time algorithms like WHCA*, CPRA* and PRA*, but only when allowed some suboptimality in the maximum distance travelled by the agents.

Restricting the corridor to what is needed for the window size used by CPR LRTS makes it behave very differently, and the resulting algorithm has big advantages when comparing maximum computational cost and maximum memory usage. However, maximum distances travelled are increased somewhat, and are much farther away from the best maximum distance travelled achieved.

PR LRTS does not perform so well, when looking at maximum distances travelled, maximum computational cost and maximum memory usage. What is responsible is likely that PR LRTS lacks a reservation table, temporal-spatial search and an abstracted reverse A* heuristic, all of which probably contribute to the success of CPR LRTS.

LRTS performs very badly, and did never manage to solve all maps within the deadline imposed.

Shared distance heuristics did not give a statistically significant advantage.

Direction maps consistently increased maximum distances travelled, but this may be caused by bad parameters given to it.

Conclusions to hypotheses

Temporal-spatial reservation search through PR LRTS corridor will decrease path lengths, reduce deadlocks, but not eliminate them.

Temporal-spatial search in a PR LRTS corridor did perform very well, and did often manage to solve all problems. The congestion in the problems could however have been much higher than in the tests that were run. Also, this hypothesis and the next one were not measured independently, due to technical difficulties, thus the conclusions cannot be drawn independently for them.

Continuous temporal-spatial reservation search reduces path length and deadlocks, without increasing maximum calculation time.

See hypothesis above.

LRTS agents disturb each other's search even when they do not collide, by rapidly altering the environment.

No evidence that this was a major problem was found, even though the solution (handling blocked nodes differently depending on whether they were blocked by an agent or always blocked, performing two separate searches) did increase the amount of agents that reached their goals slightly in some situations. However, this approach doubled the computational cost, and was not worth it.

Weighting decreases path length and memory usage.

Limited success was had with weighting. γ values of 0.3, 0.7 and 1.0 were tested, and 0.7 was performing best out of them.

Shared distance heuristics can be used to decrease overall path length, by gradually learning the environment.

Surprisingly, shared distance heuristics did not bring any advantage or difference to the algorithms. The reason is probably that the propagation of shared heuristics between agents was negligible.

Deadlocks and path length can be reduced by using direction maps.

Also surprisingly, direction maps affected all measured algorithms negatively. As only a w_{max} of 10.0 was tested, other values of w_{max} (in particular, lower values) may give better results.

Updating h of all intermediary nodes in LRTS will improve convergency times.

Updating intermediary nodes had a negligible effect. The likely reason is that lookahead enables agents to attain the same information anyway.

Deadlocks in PR LRTS can be reduced by not using the nearest unabstracted node of the abstracted goal node as a subgoal.

As mentioned, a better way of doing this was discovered: search to any node that is a child of the abstracted goal.

Still, it turned out that this did not have much of an effect over pathfinding to the nearest node of the abstracted goal.

6.2 Future work

Rather small maps (128×128) were tested. Bigger maps should be tested, to see if CPR LRTS still has the same advantages. It will likely perform a bit worse, as the disadvantages of LRTS on a big map will begin to show again, and as the real worst case computational cost and memory usage will be measured. A map with a size of at least 512×512 should be tested, to confirm the results of this thesis.

Other values of w_{max} for direction maps should be tested.

More attention should be put into creating a more rigid theoretical analysis of CPR LRTS. For example, it should be determined whether adoptees in map clique abstraction still enable the algorithm to be real-time, and if it does, how big the disadvantage in maximum computational cost and memory usage it gives is. How the different parameters affect CPR LRTS should also be analyzed theoretically more thoroughly.

Also, the following hypotheses were skipped, but should be verified in the future:

Deadlocks in PR LRTS can be reduced by avoiding planning abstract path through congested nodes.

When planning with PR LRTS, deadlocks may occur. If one avoids congested nodes, one may avoid some deadlocks. This may interfer with temporal-spatial search somewhat, as routes which are traversable may be skipped.

Deadlocks in PR LRTS can be reduced by re-planning agent's search when no path was found on last iteration.

If PR LRTS has deadlocked, one wants to have some way of breaking the deadlock. This could be handled by invoking congestion-avoiding search when this happens, or simply introducing randomness, as in LRA*.

Deadlocks in PR LRTS can be reduced by avoiding planning an abstract path that cannot be traversed.

The abstract map could be held updated with the current traversable abstract nodes. This may interfer with temporal search somewhat, as routes which are traversable may be skipped.

References

[1] Björnsson, Y., Bulitko, V., and Sturtevant, N. R. TBA*: Time-bounded A*. In *IJCAI* (2009), C. Boutilier, Ed., pp. 431–436.

[2] Bulitko, V. Learning for adaptive real-time search. *CoRR cs.AI/0407016* (2004).

[3] Bulitko, V., and Lee, G. Learning in real-time search: A unifying framework. *J. Artif. Intell. Res. (JAIR) 25* (2006), 119–157.

[4] Bulitko, V., Sturtevant, N. R., Lu, J., and Yau, T. Graph abstraction in real-time heuristic search. *J. Artif. Intell. Res. (JAIR) 30* (2007), 51–100.

[5] Bulitko, V. K., and Bulitko, V. On backtracking in real-time heuristic search. *CoRR abs/0912.3228* (2009).

[6] Ebendt, R., and Drechsler, R. Weighted A* search - unifying view and application. *Artif. Intell. 173*, 14 (2009), 1310–1342.

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

[8] Holte, R. C., Grajkowski, J., and Tanner, B. Hierarchical heuristic search revisited. In *SARA* (2005), J.-D. Zucker and L. Saitta, Eds., vol. 3607 of *Lecture Notes in Computer Science*, Springer, pp. 121–133.

[9] Hopcroft, J., Schwartz, J., and Sharir, M. On the complexity of motion planning for multiple independent objects; PSPACE hardness of the "warehouseman's problem". *The International Journal of Robotics Research, Vol. 3, No. 4, 76-88 (1984) 3*, 4 (1984), 76–88.

[10] Jansen, M. R., and Sturtevant, N. R. Direction maps for cooperative pathfinding. In *AIIDE* (2008), C. Darken and M. Mateas, Eds., The AAAI Press.

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

[12] Shue, L.-Y., and Zamani, R. An admissible heuristic search algorithm. In *ISMIS* (1993), H. J. Komorowski and Z. W. Ras, Eds., vol. 689 of *Lecture Notes in Computer Science*, Springer, pp. 69–75.

[13] Silver, D. Cooperative pathfinding. In *AIIDE* (2005), R. M. Young and J. E. Laird, Eds., AAAI Press, pp. 117–122.

[14] Sturtevant, N. Hierarchical Open Graph. `http://webdocs.cs.ualberta.ca/~nathanst/hog.html`. Online; accessed 02-Mar-2010.

[15] Sturtevant, N. R., and Buro, M. Partial pathfinding using map abstraction and refinement. In *AAAI* (2005), M. M. Veloso and S. Kambhampati, Eds., AAAI Press / The MIT Press, pp. 1392–1397.

[16] Sturtevant, N. R., and Buro, M. Improving collaborative pathfinding using map abstraction. In *AIIDE* (2006), J. E. Laird and J. Schaeffer, Eds., The AAAI Press, pp. 80–85.

Appendix A - MySQL database and its structure

The compressed dump of the MySQL database can be found at `http://fileadmin.cs.lth.se/ai/xj/AlexanderToresson/thesis_results.sql.gz`.

tests table

The *tests* table is where all test data is located. It has the following columns per test:

Basic test identification

- **name** - A symbolic string describing the algorithm with parameters.
- **algorithm** - The name of the algorithm.
- **map** - The map the test was run on.
- **numUnits** - The number of agent used in the test.
- **id** - A unique integer ID of the test.

Statistics generated

- **sumNodesExpanded, averageNodesExpanded, stdevNodesExpanded, maxNodesExpanded** - The sum, average, standard deviation and maximum number of nodes expanded per agent per time step.
- **sumNodesTouched, averageNodesTouched, stdevNodesTouched, maxNodesTouched** - The sum, average, standard deviation and maximum number of nodes touched per agent per time step.
- **sumDistanceMovedDone, averageDistanceMovedDone, stdevDistanceMovedDone, maxDistanceMovedDone** - The sum, average, standard deviation and maximum of the distance that agents that reached their goal travelled.
- **sumTimestepNodesExpanded, averageTimestepNodesExpanded, stdevTimestepNodesExpanded, maxTimestepNodesExpanded** - The sum, average, standard deviation and maximum number of nodes expanded per time step.
- **sumTimestepNodesTouched, averageTimestepNodesTouched, stdevTimestepNodesTouched, maxTimestepNodesTouched** - The sum, average, standard deviation and maximum number of nodes touched per time step.
- **sumMemoryUsed, averageMemoryUsed, stdevMemoryUsed, maxMemoryUsed** - The sum, average, standard deviation and maximum number of memory used per agent per time step. Yes, the sum of this does not make sense, as it is akin to the integral of the memory usage.

- **maxFirstTimestepNodesExpanded, maxFirstTimestepNodesTouched** - The number of nodes expanded and touched on the first time step of an algorithm.
- **sumDistanceMoved** - The sum of the distances travelled by the agents.
- **exactPercentDone** - The percentage of agents that reached their goals.

Algorithm parameters

LRTS parameters

- **lookahead** - The LRTS lookahead used.
- **gamma** - The LRTS γ used.
- **handleOccupied** - Whether LRTS treats occupied nodes in a special way.
- **updateIntermediate** - Whether LRTS updates the intermediate nodes between the current one and the one used to get h.
- **greedyGoal** - Whether LRTS always greedily goes towards the goal or not, when it is within reach.

(C)PR LRTS corridor parameters

- **absLevel** - The abstraction level LRTS is run at, to generate a corridor for PR LRTS or CPR LRTS.
- **corridorWidth** - The width of the corridor.
- **fixSingleConnectivity** - Whether to fix single connectivity or not.

PR LRTS parameters

- **fallbackToNearest** - Whether falling back to the nearest node in the PR LRTS corridor was used or not, when the goal could not be reached.
- **searchToMiddleChild** - Whether to perform a search in the corridor to the 'average' child of the abstracted goal node, or to any node that is a child of the abstracted goal node.

CPR LRTS parameters

- **restrictCorridor** - Whether to restrict corridors or not.

WHCA* parameters

- **wHCAAbsLevel** - The WHCA* abstraction level used.

CPRA* parameters

- **adaptiveWindow** - Whether to use an adaptive window or not.

Other parameters

- **windowSize** - The CPRA*, PRA* or WHCA* window size used.
- **dm** - Whether direction maps were used or not. For tests that were not used to benchmark direction maps, this is NULL.
- **alpha** - The α value used for direction maps.
- **wmax** - The w_{max} value used for direction maps.
- **sdh** - Whether shared distance heuristics were used or not. For tests that were not used to benchmark shared distance heuristics, this is NULL.
- **run** - The sequential run number of direction map and shared distance heuristic tests. For tests not used to benchmark direction maps or shared distance heuristics, this is NULL.

Views

The views only provide summarized results for the most important statistics.

- **allruns** - Presents averaged results over all runs in direction maps and shared distance heuristic tests.
- **pertest** - Presents summarized results over all 12 tests done per algorithm and parameter combination.
- **permap** - Presents summarized results over all 4 tests done per algorithm, parameter combination and number of agents.

General notes

Algorithm parameters which do not apply to a specific algorithm are generally set to NULL.

Due to an unfortunate bug, direction maps with a w_{max} other than 10.0 was not tested. Thus the entries in the database where w_{max} is 5.0 or 20.0 are erroneous and should be ignored.

Appendix B - Queries used to generate results

LRTS

Lookahead

```
SELECT lookahead, max(maxNodesExpanded), max(maxNodesTouched), avg(
    averageDistanceMovedDone), max(maxMemoryUsed), avg(exactPercentDone) FROM
    tests WHERE algorithm = 'LRTS' AND run IS NULL AND handleOccupied = 0 AND
    updateIntermediate = 0 AND greedyGoal = 0 GROUP BY lookahead
```

Gamma

```
SELECT lookahead, gamma, avg(exactPercentDone), max(maxMemoryUsed), max(
    maxNodesExpanded), avg(averageDistanceMovedDone), avg(maxDistanceMovedDone
    ), avg(averageNodesExpanded), avg(averageNodesTouched), max(
    maxNodesTouched), avg(averageMemoryUsed) FROM tests WHERE algorithm = '
    LRTS' AND run IS NULL AND handleOccupied = 0 AND updateIntermediate = 0
    AND greedyGoal = 0 GROUP BY lookahead, gamma
```

Boolean parameters

```
SELECT gamma, greedyGoal, updateIntermediate, handleOccupied, avg(
    exactPercentDone) FROM tests WHERE algorithm = 'LRTS' AND run IS NULL AND
    lookahead > 1 GROUP BY gamma, greedyGoal, updateIntermediate,
    handleOccupied
```

Direction maps

```
SELECT dm, lookahead, gamma, avg(maxDistanceMovedDone), avg(exactPercentDone)
    from tests WHERE algorithm = 'LRTS' AND run is not null AND (map = '
    test_s3' OR map = 'empty') AND dm is not null GROUP BY dm, lookahead,
    gamma ORDER BY dm, lookahead, gamma
```

PR LRTS

Lookahead

```
SELECT absLevel, lookahead, max(maxNodesExpanded), max(maxDistanceMovedDone),
    max(maxMemoryUsed), avg(exactPercentDone) FROM tests WHERE algorithm = 'PR
     LRTS' AND run IS NULL AND corridorWidth = 1 AND fixSingleConnectivity = 0
     AND fallbackToNearest = 1 AND searchToMiddleChild = 0 GROUP BY absLevel,
    lookahead
```

Corridor width

```
SELECT absLevel, corridorWidth + fixSingleConnectivity * 0.5, max(
    maxNodesExpanded), avg(maxDistanceMovedDone), max(maxMemoryUsed), avg(
    exactPercentDone) FROM tests WHERE algorithm = 'PR LRTS' AND run IS NULL
    AND lookahead = 16 GROUP BY absLevel, corridorWidth, fixSingleConnectivity
     ORDER BY absLevel, corridorWidth, fixSingleConnectivity
```

Corridor width and boolean parameters

```
SELECT absLevel, corridorWidth + fixSingleConnectivity * 0.5,
    fallbackToNearest, searchToMiddleChild, max(maxNodesExpanded), avg(
    maxDistanceMovedDone), max(maxMemoryUsed), avg(exactPercentDone) FROM
    tests WHERE algorithm = 'PR LRTS' AND run IS NULL AND lookahead = 16 GROUP
     BY fallbackToNearest, searchToMiddleChild, absLevel, corridorWidth,
    fixSingleConnectivity ORDER BY fallbackToNearest, searchToMiddleChild,
    absLevel, corridorWidth, fixSingleConnectivity
```

Direction maps

```
SELECT dm, absLevel, max(maxNodesExpanded), avg(maxNodesTouched), avg(
    maxDistanceMovedDone), max(maxMemoryUsed), avg(exactPercentDone) from
    tests WHERE algorithm = 'PR LRTS' AND run is not null AND (map = 'empty'
    OR map = 'test_s3') AND dm is not null GROUP BY dm, absLevel ORDER BY
    absLevel, dm
```

CPR LRTS

Corridor width

```
SELECT restrictCorridor, corridorWidth + fixSingleConnectivity * 0.5, avg(
    exactPercentDone), avg(maxNodesExpanded), avg(maxDistanceMovedDone), avg(
    maxMemoryUsed) FROM tests WHERE algorithm = 'CPR LRTS' AND run IS NULL AND
     lookahead = 16 GROUP BY restrictCorridor, corridorWidth,
    fixSingleConnectivity ORDER BY restrictCorridor, corridorWidth,
    fixSingleConnectivity
```

Corridor abstraction level

```
SELECT restrictCorridor, absLevel, avg(exactPercentDone), avg(maxNodesExpanded
    ), avg(maxDistanceMovedDone), avg(maxMemoryUsed) FROM tests WHERE
    algorithm = 'CPR LRTS' AND run IS NULL AND lookahead = 16 AND wHCAAbsLevel
     < 2 GROUP BY restrictCorridor, absLevel ORDER BY restrictCorridor,
    absLevel
```

WHCA* abstraction level

```
SELECT restrictCorridor, wHCAAbsLevel, avg(exactPercentDone), avg(
    maxNodesExpanded), avg(maxDistanceMovedDone), avg(maxMemoryUsed) FROM
    tests WHERE algorithm = 'CPR LRTS' AND run IS NULL AND lookahead = 16 AND
    absLevel >= 4 AND windowSize >= 16 GROUP BY restrictCorridor, wHCAAbsLevel
     ORDER BY restrictCorridor, wHCAAbsLevel
```

WHCA* window size

```
SELECT restrictCorridor, windowSize, avg(exactPercentDone), avg(
    maxNodesExpanded), avg(maxDistanceMovedDone), avg(maxMemoryUsed) FROM
    tests WHERE algorithm = 'CPR LRTS' AND run IS NULL AND lookahead = 16 AND
    wHCAAbsLevel < 2 GROUP BY restrictCorridor, windowSize ORDER BY
    restrictCorridor, windowSize
```

LRTS lookahead

```
SELECT lookahead, absLevel, avg(exactPercentDone), avg(maxMemoryUsed), avg(
    maxNodesExpanded), avg(maxDistanceMovedDone), avg(averageDistanceMovedDone
    ), avg(averageNodesExpanded), avg(averageNodesTouched), avg(
    maxNodesTouched), avg(averageMemoryUsed) FROM tests WHERE algorithm = 'CPR
     LRTS' AND run IS NULL AND restrictCorridor = 0 AND corridorWidth <= 1
    GROUP BY lookahead, absLevel ORDER BY lookahead
```

Direction maps

```
SELECT windowSize, dm, avg(maxNodesExpanded), avg(maxNodesTouched), avg(
    maxDistanceMovedDone), avg(maxMemoryUsed), avg(exactPercentDone) from
    tests WHERE algorithm = 'CPR LRTS' AND run is not null AND dm is not null
    GROUP BY windowSize, dm ORDER BY dm, windowSize
```

Comparisons

By travelled distance

```
SELECT * FROM pertest p WHERE `avg(exactPercentDone)` = 100.0 AND run is NULL
    ORDER BY `avg(maxDistanceMovedDone)`
```

By memory usage

```
SELECT * FROM pertest p WHERE `avg(exactPercentDone)` = 100.0 AND run is NULL
    ORDER BY `max(maxMemoryUsed)`
```

By expanded nodes

```
SELECT * FROM pertest p WHERE `avg(exactPercentDone)` = 100.0 AND run is NULL
    ORDER BY `max(maxNodesExpanded)`
```

By score, including memory usage

```
SELECT * FROM pertest p WHERE `avg(exactPercentDone)` = 100.0 AND run IS NULL
    AND `avg(maxDistanceMovedDone)` <= 147.54 * 1.2 ORDER BY `avg(
    maxDistanceMovedDone)` * 10 + `max(maxNodesExpanded)` * 10 + `max(
    maxNodesTouched)` + `max(maxMemoryUsed)`
```

By score, excluding memory usage

```
SELECT * FROM pertest p WHERE `avg(exactPercentDone)` = 100.0 AND run IS NULL
    AND `avg(maxDistanceMovedDone)` <= 147.54 * 1.2 ORDER BY `avg(
    maxDistanceMovedDone)` * 10 + `max(maxNodesExpanded)` * 10 + `max(
    maxNodesTouched)`
```

Appendix C - Terminology

- α - The learning speed for direction maps.
- adoptee - A node that is not formally part of a clique, but has been adopted by it.
- agent - An entity moving in a virtual environment. Can for example ba a robot or a unit in a computer game.
- clique - A group of nodes where an edge exists between all of them. An n-clique is a clique consisting of n nodes.
- expanded node - A node which has been processed and had its neighbours processed.
- γ - Weighting used for LRTS.
- lookahead - The search depth used for LRTS.
- ply - The nodes on a specific search depth.
- spatial - Pertaining to space.
- temporal-spatial - Pertaining to space and time.
- touched node - A node that has been processed.
- time step - A discrete time step in the simulation.
- window size - The search depth used for temporal-spatial search in WHCA*.
- w_{max} - The maximum penalty incurred by direction maps.