5-24-2019

# Performance of A* Based Search Algorithms in Grid-Based Stochastic Multi-Agent Systems with Static Obstacles and Non-Static Goal States

Kallen M. Harvey
*Portland State University*

# Performance of A* Based Search Algorithms in Grid-Based Stochastic Multi-Agent Systems with Static Obstacles and Non-Static Goal States

by

Kallen Harvey

An undergraduate honors thesis submitted in partial fulfillment of the

requirements for the degree of

Bachelor of Science

in

University Honors

and

Computer Science

Thesis Adviser

Dr. Marc Goodman

Portland State University

2019

# Performance of A* Based Search Algorithms in Grid-Based Stochastic Multi-Agent Systems with Static Obstacles and Non-Static Goal States

Kallen Harvey
Portland State University
kaharvey@pdx.edu

## Abstract

In this paper, I compare the performance of various A*-based search algorithms in a search space where the assumption that goal states must be static is removed. I provide a basic overview of the algorithms used and explain how I created the environment in which the agents will move. Both off-line/incremental and real-time A*-based searches are compared to their original published versions in a grid-based environment similar to those described in each. I explain my results and make suggestions for future research.

## Introduction

Many problems within Artificial Intelligence can be characterised as state-space search problems [Russell & Norvig, 2010], and the ability to understand how various search algorithms perform in a search space is paramount in selecting which algorithm to use. Specifically, the performance of an agent within the domain it is situated in is dependant on the properties of the domain, and the search algorithm used. Though research related to traditional state-search spaces is primarily focused on discovering means to effectively search larger and larger spaces, I was interested in search spaces able to fit into computer memory without negatively impacting the performance of the system. Looking further, I found that there has been significant research completed around agents with static goal states in incremental searches [Dijkstra, 1959], [Hart et al., 1968], [Hart et al.,1972], and real-time searches [Koenig & Sun, 2009], but could not at the time identify any situated around agents trying to locate a non-static goal state in either type of environment. After the system and data presented here were complete, I was made aware of

Moving Target Search [Ishida & Korf, 1995], and MtsCopa [Baier et al., 2015], and other associated work dealing with non-static goals. While research into stochastic search algorithms such as [Abdolmaleki et al., 2017] and [Olson & Shehu, 2013] exists, these focus more on fitness functions in Expectation-Maximisation based search and navigating high-dimensional search spaces through evolutionary based stochastic search. To keep the research within a reasonable scope for the time I had available, I did not use or adapt these algorithms.

The apparent lack of research during my initial background search into dynamic goal states thus invited inquiry. This research will therefore be investigating how a selection of algorithms, all based on A*, perform in a search space with non-static goals. There are many real-world motivations for this inquiry, such as common navigation tasks in real-time environments and computer game pathfinding. Previous research by [Koenig & Sun, 2009] and [White, 2007] have explored the capabilities of real-time computer game agents to move within both known and unknown domains towards a static goal. One issue not explored in these papers is how the performance of these algorithms changes when the requirement for the positioning of the goal state to remain static is removed. A real-world example of this situation is the oft-decried ‘follow’ mission found in many modern role-playing video games. These missions require the player to follow a non-player character (NPC), an agent, through a search space towards a common, static, goal. One possible method to solve the problem is to provide a command that automates the player character’s pathfinding, setting the goal state as the current state the NPC occupies. Also of interest is the simulation of predatory NPCs such as the Alien in Alien:Isolation, or other games in which an agent must locate the player character. In these situations, pathfinding with static goals rely on the ability to intermittently update the goal state to simulate the player's position. As the responsiveness of these systems increases, the ability to dynamically set and reach a goal state in real time without loss of performance will become more relevant. Researchers in robotics may also find this research of use in the future when selecting search algorithms to base an agents navigation through a 3-dimensional space with low latency on.

## Background

Throughout the history of the field, there have been several seminal means of searching through a space to locate a goal. These algorithms, while focusing on static-goal navigation, are well-researched and their performance is well documented. Of these algorithms, this research is

focused on the A* family, born from Dijkstra's shortest path algorithm, and the FastMap pre-processing algorithm for A*-based searches. A* and Dijkstra's search were selected due to their prolific use in the robotics and video game industry for pathfinding, as well as their history. Local Search Space Real-Time A* search [Koenig & Sun, 2009] was selected as the most modern of the A* family to analyse for this paper due to its use of both Dijkstra's and A*, and the desire that this research be relevant to real time systems. The similarities this introduces reduced the complexity of the code and system, something desperately needed as the initial scope of the project included many other algorithms and search spaces. Incorporation of these algorithms and alternate spaces was abandoned to create a manageable scope and ensure the system was completed in a timely manner.

## Dijkstra's Shortest Path

As the earliest search algorithm this research is relaxing assumptions in, Dijkstra's algorithm [Dijkstra, 1959] is arguably the foundation on which this paper, and all the other algorithms investigated, is based on. Used extensively as the introduction and baseline for pathfinding in video games and tutorials, it is one of the most well-known search algorithms. While the original only finds the distance from the source waypoint to the goal, I implemented a more commonly used version that finds and maintains a tree of shortest paths rooted at the initial waypoint. It starts by creating a table for the visited waypoints, and fixing the initial waypoint as the source. Iterating through the neighbors of the current waypoint and updating the max distance to that waypoint found so far. The high rate of expansion in search spaces with a moderate branching factor or a moderate number of unique waypoints can lead to lower performance and higher memory requirements, and make this algorithm less suited for large or real-time systems.

## A*

One of the most well-known and seminal search algorithms in the field, A* [Hart et al., 1968], [Hart et al., 1972] uses the cost between the starting waypoint to each adjacent waypoint and the heuristically estimated cost to the goal. As the basis for a large section of the algorithms used in research and industry, A* is applied in this research as part of the baseline against which to measure the performance of other algorithms in the family of A*. It is also used in LSS-LRTA* to create local search spaces. In order to simulate a priority queue in python, I used

pythons heapq heapify functionality on the keys of a dict. This ‘priority queue set’ was used to hold the waypoints and improve the ability to easily remove the waypoint with the lowest estimated cost. Because the environment was a grid-based waypoint graph, I used the Manhattan distance for my heuristic function. As the heuristic function has a large impact on the performance and the ability of an agent using A*, I have used the same heuristic as [Koenig & Sun, 2009] in their analysis of A* for creating LSS-LRTA*, and the FastMap algorithm.

## FastMap

The FastMap algorithm is a preprocessing heuristic algorithm for A*-based searches on undirected, weighted edge graphs. [Cohen et al., 2018] introduces the algorithm and proves it is both admissible and consistent, better performing than Differential heuristic, and generalizable for all graphs. FastMap functions by finding the two nodes on a graph furthest from each other, then calculating the distances from these nodes to every other node in the graph, using trees rooted at the two nodes. These trees offer the querying search algorithm lower response time during search in exchange for some additional overhead and memory complexity before the first search occurs. The algorithm was selected for this research to incorporate the current state-of-the-art heuristic for A*-based searches to control for the impact heuristic selection makes in a real-time system.

## LSS-LRTA*

Local Search Space Learning Real Time A* search [Koenig & Sun, 2009] uses A* to determine a non-disk-shaped local search spaces in a fine-grained way and updates the h-values of all states in the local search spaces to learn quickly. First, LSS-LRTA* uses A* to choose its local search spaces. A* searches from the current state of the agent toward the goal state until lookahead > 0 states have been expanded or the goal state is about to be expanded. The states expanded by A* form the local search space. LSS-LRTA* then uses Dijkstra’s algorithm to update the h-values of all states in the local search space. Dijkstra’s algorithm replaces the h-values of all states in the local search space with the sum of the distance from the state to a state s and the h-value of state s, minimized over all states $s \in S$ that were generated but not expanded by A* (= that border the local search space). LSS-LRTA* moves the agent along the

path found by A* until it reaches the end of the path (and leaves the local search space) or action costs on the path increase. If the current state of the agent is different from the goal state, then LSS-LRTA* repeats the process, otherwise it terminates successfully. This algorithm was selected for this research for its incorporation of Dijkstra's and A*, the ability to measure the performance of my system against the published data within the paper, and the advent of using a search designed for real-time environments.

## Methodology

The domain used for this research is an undirected waypoint graph [White, 2007] representation of a grid-based environment with static obstacles, pinch and ambush points [White, 2007] for the agents to navigate.

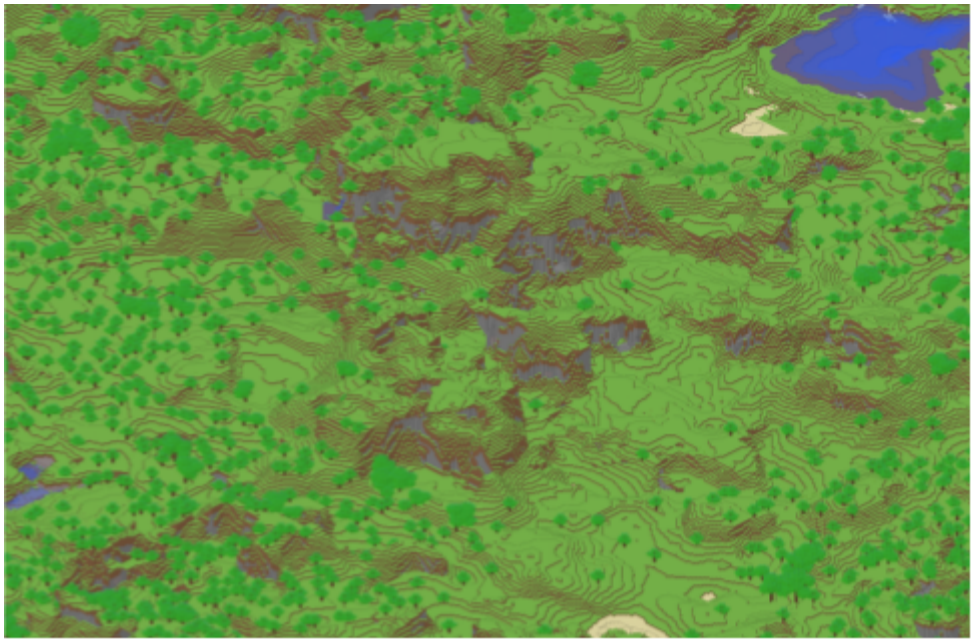

Figure 1: Isometric 3D perspective of the Minecraft 1.3.2 world used to generate the grid waypoint graph

The environment was generated by mapping a 320 x 320 grid of the map generated by Minecraft Java Edition 1.3.2 to a waypoint graph and removing edges between nodes with a heightmap difference greater than 1. These removed edges are the abstractions representative of static obstacles in the search space.

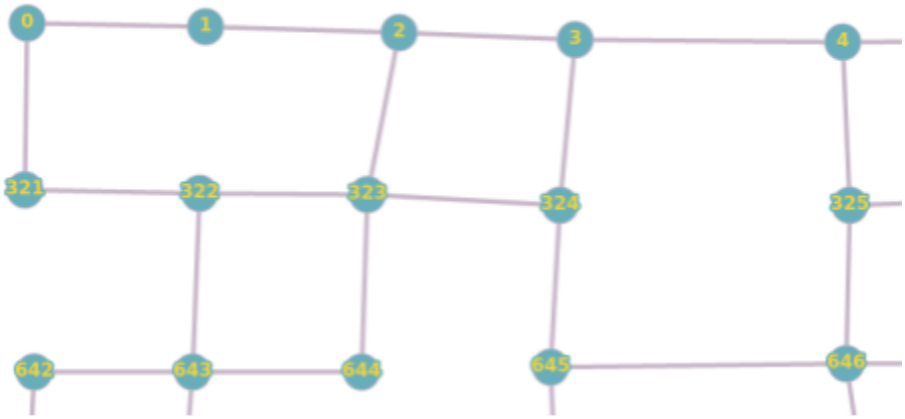


Figure 2: Visual Representation of the original heightmap waypoint graph.

In order to guarantee the presence of ambush and pinch points, the algorithm from [White, 2007] was applied to the grid, searching for waypoints with 3 neighbors instead of 2.

**Find_Pinch_Points**(Graph *G*)
Create an empty set *P* for storing pinch points
For each waypoint *n* in *G*
If *n* has exactly 2 neighbors *a* and *b*
Create *G'* = *G* – *n*
If no path exists from *a* to *b* in *G'*
Add *n* to Set *P*
Return *P*

Figure 3: Pinch Point algorithm from [White, 2007]

**Find_Ambush_Points**(PinchPoint *p*, Graph *G*)
Create an empty set *A* for storing ambush points
For each neighbor *m* of *p*
For each waypoint *n* in *G*
If *m* is visible to *n* and *p* is not visible to *n*
Add *n* to Set *A*
Return *A*

Figure 4: Ambush Point algorithm from [White, 2007]

The waypoints found were added to a python Set. As the waypoints were popped off the set, there was a 33% chance that one of their edges would be removed, and a 5% chance two of their edges would be removed. The use of a Set ensured that no waypoint would be present more than once, and the stochastic edge removal ensured the presence of at least one point of interest. The algorithms in figures 3 and 4 were then applied to the graph, and the results (figure 5) were the final representation of the environment.

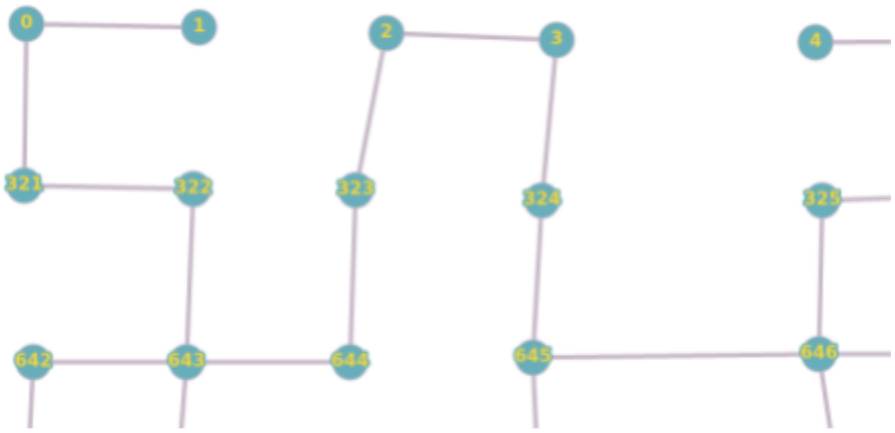


FIgure 5: Visual representation of waypoint graph after applying changed pinch point algorithm
Pinch Points: 0, 321, 322
Ambush Points: 322 (0), 643 (321)

In order to effectively measure the run-time performance of the search algorithms, I created two classes of agents able to navigate the environment. As the environment is small and able to fit into computer memory, it is important to keep the implementation and hardware details as similar as possible across iterations. To ensure this, the agents are implemented in an object-oriented hierarchy, and implementation details are kept as similar as possible. The abilities of these agents are delineated by their knowledge of the domain, the mobility of their goal state, and the search algorithms available for their use. Each agent has precepts for local, agent-centered search [Koenig, 2001a], and the ability to move within the domain, limited by the available edges (= static obstacles). The precepts allow every agent to observe the state of the waypoint they are in including: the location (x,y) within the grid; the available neighbors and their h-value/cost; the presence of other agent(s); and if the waypoint is near a blind corner. Each agent has a parameterized lookahead initially set to 3.

Hunters, utility-based agents, have a lookahead of 5 and use a different A*-based search algorithm each iteration to search the domain. In the first three iterations the Hunters goal state is the Hunter agent, and the Hunter has a limited knowledge of the domain. When, in the final two iterations, the hunter agents apply the pinch/ambush point and FastMap preprocessing algorithms, they have a primary and secondary goal state, instances of the Hunted agents and points of interest, respectively. The system allowing Hunters to find the points of interest require the Hunter agents to have a complete knowledge of the environment, making the development of the last two iterations much more difficult.

Hunter agents started as simple reflex agents [Russell & Norvig, 2010], and in the final iteration were controlled by a player. The precept-action rules for the first four iterations were simple: see an agent in a visible neighbor waypoint, try to move to a waypoint not in its neighbors; see one of two waypoints marked as an exit, move to them; pick a random waypoint not in the last 5 visited. The exit waypoints, waypoints 0 and 102399, were selected to ensure that in the worst case (from the perspective of the Minecraft world) the Hunted agent would need to traverse an absolute diagonal of 227 blocks. Given that the maximum available movement options along the waypoint graph are the four cardinal directions, the exits were not ideally located.

The player controls were implemented using pynput's [Palmér, 2019] keyboard listeners to detect key presses. This allowed the testing of the best-performing algorithm in an environment closely simulating pathfinding in a real-time video game.

Using the pseudocode presented in [Cohen et al., 2018], [Dijkstra, 1959], [Hart et al., 1968], [Hart et al., 1972], and [Koenig & Sun, 2009], I implemented the following search and preprocessing algorithms in Python 3.6.7. Due to time constraints, I was only able to test the system with one agent/one player and two agents in iterations of an adversarial game. Many of my future research suggestions therefore focus on the planned iterations, features, and improvements I was unable to complete.

## Experimental results

As in [Koenig & Sun, 2009], the main task of this thesis is to compare the performance of various search algorithms. Unlike [Koenig & Sun, 2009], though, I am comparing within the class of A* search algorithms, and have more freedom in comparing through proxies such as waypoint expansions due to the similar basic operations. The data is organised according to iteration, and the first four iterations consisted of 1,000 system runs. The fifth consisted of only 25, as the limited map size and quick movement of the hunter using a real-time search led to a significant player disadvantage. The system ran asynchronously, with the hunter search, game window/GUI, and movement occuring on seperate threads in a multiprocessing pool to avoid GUI updates and other expensive operations forcing one agent to wait until a loop completed before being able to execute an action. This was done to ensure that LSS-LRTA* was able to run in a non-incremental, real-time environment. The later in the per iteration run count the search occurred, the more pronounced the disparity between the GUI and the actual agent

locations became. This was the primary frustration and motivation behind limiting the number of runs during system iteration number five.

| Iteration | Hunter Algorithm | Hunted Control | Average waypoint expansions (Hunter) | Average time to capture (seconds) | Average waypoints traveled (Hunted) |
|---|---|---|---|---|---|
| 1 | A* | Reflux | 47,717.32 | 36.62 | 1,494.15 |
| 2 | Dijkstra's | Reflux | 16,356.49 | 18.54 | 756.43 |
| 3 | LSS-LRTA | Reflux | 705.29 | 4.36 | 130.86 |
| 4 | LSS-LRTA+ FastMap | Reflux | 377.38 | 1.65 | 57.69 |
| 5 | LSS-LRTA+ FastMap | Player Keyboard | 177.40 | 0.58 | 0.84 |

# Analysis

In order to accurately measure the performance of the system and agents in this research, a comparison to the published results in [Koenig & Sun, 2009], summarised in Figure 6, is needed.

(a) = cell expansions per search (= lookahead) (b) = average number of cell expansions, (c) = average number of searches, (d) = average number of action executions (= trajectory length), (e) = average number of action executions per search (= trajectory length per search), (f) = average search time, (g) = average search time per search, and (h) = average search time per action execution.

| look-ahead (a) | cell expansions (b) | searches (c) | trajectory length (d) | trajectory length per search (e) | search time (f) | search time per search (g) | search time per action (h) |
|---|---|---|---|---|---|---|---|
| | | | D* Lite | | | | |
| - | 11424.90 | 72.54 | 308.98 | 4.26 | 36825.63 | 507.65 | 119.18 |
| | | | LSS-LRTA* | | | | |
| 1 | 498.55 | 498.55 | 498.55 | 1.00 | 28279.51 | 56.72 | 56.72 |
| 3 | 622.46 | 207.83 | 377.15 | 1.81 | 28380.11 | 136.56 | 75.25 |
| 5 | 686.46 | 137.77 | 337.67 | 2.45 | 28435.03 | 206.39 | 84.21 |

Figure 6. Table 6 from [Koenig & Sun, 2009] for their results in a Grid with Random Obstacles, truncated to the relevant lookahead value of 5.

(b) as in Figure 6, (f) converted from microseconds to seconds.

| Iteration | Algorithm | Look-ahead | Waypoint Expansions | (b) | Search Time/ Goal | (f) |
|---|---|---|---|---|---|---|
| 3 | LSS-LRTA | 5 | 705.29 | 686.46 | 0.033 | 0.028 |
| 4 | LSS-LRTA FastMap | 5 | 377.38 | 686.46□ | 0.029 | 0.028□ |
| 5 | LSS-LRTA FastMap Player Control | 5 | 177.40 | 686.46□ | 0.69 | 0.028□ |

□: Because the FastMap algorithm was not applied in [Koenig & Sun, 2009], this data is for reference against the improvements made in my system with its application. Similar improvements are expected in the case that the algorithm is applied to their work as well. It exists here as a baseline.

My results were similar to, but slightly worse than, those in [Koenig & Sun, 2009] in the case of the base LSS-LRTA* algorithm. This may be due to the differences in environment, such as their grid obstacles existing at a rate of 25%, my choice of programming language, or even optimizations made in their code not reflected in the pseudocode I based my own on. Differences in implementation of A* and Dijkstra's could also account for the difference in performance. In any case, there does not appear to be any significant downside to removing or

relaxing the assumption that goal states remain static throughout the search process for this algorithm. Given more time and a better understanding of their environment, I would expect that the performance of my own environment, and the agents within it, would more closely match that in [Koenig & Sun, 2009].

When applying the FastMap algorithm, a meaningful reduction in the number of expanded waypoints is seen, along with a slight reduction in the per goal search time that brings the system closer to the time performance reported by [Koenig & Sun, 2009]. The improvements reported in [Cohen et al., 2018] were concerned with comparison against other Euclidean embedding and dimensionality reduction data preprocessing algorithms. Though the dimensionality of my own environment was 2, and no dimensionality reduction was needed for calculating a Euclidean distance from the Hunter to the Hunted, the FastMap algorithm was applied to the system. To limit the project scope, and because the work of comparing FastMap with another heuristic aid was already completed by [Cohen et al., 2018], I did not use the Differential heuristic from [Goldenburg et al, 2011]. To compare the performance of my system with the results for a map with only 5,214 nodes and 9,687 edges reported by [Cohen et al., 2018] with A*, shown in Figure 7, I use the metric of waypoint expansions.

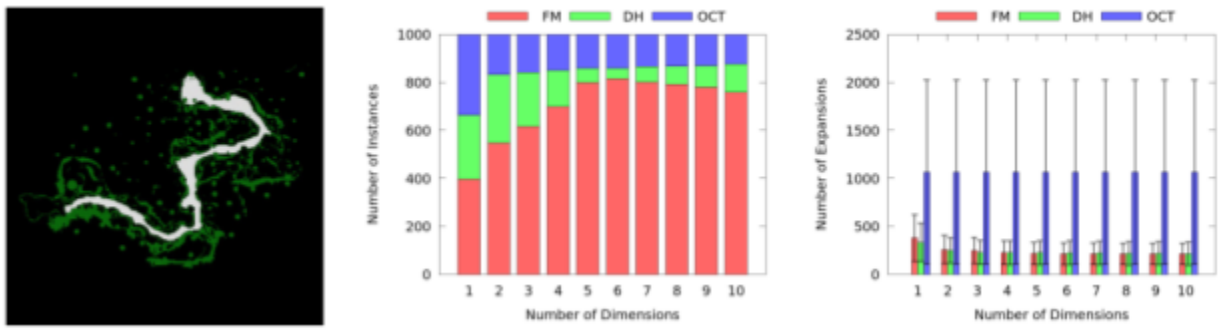


Figure 3: Shows empirical results on 3 maps from Bioware's Dragon Age: Origins. (a) is map 'lak503d' containing 17,953 nodes and 33,781 edges; (d) is map 'brc300d' containing 5,214 nodes and 9,687 edges; and (g) is map 'maze512-32-0' containing 253,840 nodes and 499,377 edges. In (b), the x-axis shows the number of dimensions for the FastMap heuristic (or the number of pivots for the Differential heuristic). The y-axis shows the number of instances (out of 1,000) on which each technique expanded the least number of nodes. Each instance has randomly chosen start and goal nodes. (c) shows the median number of expanded nodes across all instances. Vertical error bars indicate the MADs. The figures in the second and third rows follow the same order. In the legends, "FM" denotes the FastMap heuristic, "DH" denotes the Differential heuristic, and "OCT" denotes the Octile heuristic.

Figure 7: The results from [Cohen et al, 2018], truncated to relevant pieces and label.

As the only published numbers within their paper are for their 10 dimensional waypoint expansions and mean absolute deviation for each algorithm, I had to extrapolate across the provided graph to get a rough estimate of the mean waypoint expansions for 2 dimensions. There is a small difference in the bars on the graph (f) in Figure 7, and the reported mean for

dimensionality of 10 is 205. To my best estimate then, the mean waypoint expansions for the FastMap algorithm in (d) with A* search is 240. This guess is seen below in column (f).

| Iteration | Dimensions | Algorithm | Waypoint Expansions | Expansion per waypoint | (f) | Expansions per node |
|---|---|---|---|---|---|---|
| 1 | 2 | A* | 47,717.32 | 0.4659 | 240 | 0.04603 |
| 3 | | LSS-LRTA* | 705.29 | 0.006887 | | |
| 4 | | LSS-LRTA* & FastMap | 377.38 | 0.003685 | | |

The size of the environments used in gathering the data varied significantly. The larger environment used in this research, with 102,400 waypoints, dwarfed the environment used in [Cohen et al., 2018] with only 5,214 nodes. To control for this size disparity, the number of expansions per waypoint is calculated to give a better sense of how the algorithms performed independently of the search space size. The number of expansions in my implementation of A* without FastMap is an order of magnitude greater than that of A* with FastMap reported in [Cohen et al., 2018]. This is consistent with their claim that FastMap is orders of magnitude faster than the Manhattan distance heuristic I used. Of interest, though, is the improvement seen with the application of the FastMap algorithm with LSS-LRTA*. While LSS-LRTA* is much faster than A*, another order of magnitude in the reduction of number of expansions per waypoint is seen.

## Conclusions

My experimental results are hardly surprising. The small, real-time environment was best navigated by the most recent and advanced of the A*-based searches used designed for real-time environments. LSS-LRTA* is designed to learn an environment and aid in navigation within environments without agents requiring knowledge of the space. This invites the question of what environments the algorithm is unsuited to navigate and learn, and is a good topic for future research.

The FastMap algorithm is effective in real-time environments, and when combined with LSS-LRTA*, can lead to similar performance in environments with several orders of magnitude more nodes as one navigated by normal A* search.

Finally, I want to address the addition of player input for the final iteration of the system runs, and why it is relevant to this body of work. While it added a layer of complexity that was interesting to implement and test, it was one of the major 'sticking' points in the implementation. It is important to include here as an iteration of data gathering and as an example of a real-time system with an active user, because the ability for an agent to navigate such an environment without noticeable delay is a large part of the difficulty in video game pathfinding and 'AI'. Future work and research into why certain search algorithms are better suited for different environments and search spaces should also focus on real-time, user interactive systems. The original goal of this research was to investigate such qualities and aspects of several different environments and algorithms, and the player interactive session stands as a point off which future work more in line with that goal may begin.

## Future work

In future research, I would like to explore not only why relaxing the assumption of static goals had minimal impact on performance, but also incorporate the Moving Target Search [Ishida & Korf, 1995], MtsCopa [Baier et al., 2015], and other associated work dealing with non-static goals. I would also like to expand on this research to incorporate other, more complex, agents and searches such as weighted LSS-LRTA* [Rivera et al., 2013], Minimax Real-time search [Koenig, 2001b], D*-Lite [Koenig & Likhachev, 2002], and D*-Lite in multi-agent systems [Al-Mutib et al., 2011] in investigations into their performance compared to their statically-goaled versions.

Further, an in-depth exploration of why these search algorithms perform better in certain environments and how to identify properties of the search spaces that directly lead to changes in performance also needs to be explored.

Another potential avenue of research would be to expand this research to incorporate various reinforcement learning algorithms such as q-learning [Watkins, 1989] or DeepMind's deep q-learning into the agent models used in this paper. How much impact would these alternate approaches give? How would multiple instances of each agent as in [Wang, 2011] impact the system? If using multiple Hunters, how and why would additional precepts for shared information as in [Yamauchi, 1998] change the dynamic? How would those same precepts impact the Hunted? How do other types of environments, such as mazes and pre-designed

game maps, impact the performance of the Hunter agent? How would changing the lookahead of the agents impact their overall performance?

Also of interest is investigating how the search space configurations arise out of changed properties of said space, and the impact these changes have on the selection of a search algorithm. Furthering such understanding will aid designers and researchers in the field in deciding which algorithm to use given the model and attribute configurations they have.

# References


[1] Abdolmaleki, A., Price, B., Lau, N., Reis, L. P., & Neumann, G. (2017). Deriving and improving CMA-ES with information geometric trust regions. *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '17*, 657–664. https://doi.org/10.1145/3071178.3071252

[2] Al-Mutib, K., AlSulaiman, M., Emaduddin, M., Ramdane, H., & Mattar, E. (2011). D* lite based real-time multi-agent path planning in dynamic environments. *(:Unav)*. https://doi.org/10.1109/cimsim.2011.38

[3] Baier, J. A., Botea, A., Harabor, D., & Hernandez, C. (2015). Fast algorithm for catching a prey quickly in known and partially known game maps. *IEEE Transactions on Computational Intelligence and AI in Games*, *7*(2), 193–199. https://doi.org/10.1109/TCIAIG.2014.2337889

[4] Cohen, L., Uras, T., Jahangiri, S., Arunasalam, A., Koenig, S., & Kumar, T. K. S. (2018). The fastmap algorithm for shortest path computations. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (pp. 1427–1433). Stockholm, Sweden: International Joint Conferences on Artificial Intelligence Organization. https://doi.org/10.24963/ijcai.2018/198

[5] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271. https://doi.org/10.1007/BF01386390

[6] Meir Goldenberg, Nathan Sturte-vant, Ariel Felner, and Jonathan Schaeffer. (2011). The compressed differential heuristic. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*.

[7] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[8] Hart, P. E., Nilsson, N. J., & Raphael, B. (1972). Correction to "a formal basis for the heuristic determination of minimum cost paths." *SIGART Bull.*, (37), 28–29. https://doi.org/10.1145/1056777.1056779

[9] Koenig, S. (2001a). Agent-centered search. *AI Magazine*, *22*(4), 109–131.

[10] Koenig, S. (2001b). Minimax real-time heuristic search. *Artificial Intelligence*, *129*(1–2), 165–197. https://doi.org/10.1016/S0004-3702(01)00103-5

[11] Koenig, S., & Likhachev, M. (2002). D*-Lite. *Eighteenth National Conference on Artificial Intelligence,* 476-483.

[12] Koenig, S., & Sun, X. (2009). Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, *18*(3), 313–341. https://doi.org/10.1007/s10458-008-9061-x

[13] Olson, B., & Shehu, A. (2007). Multi-objective stochastic search for sampling local minima in the protein energy surface. *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics - BCB'13*, 430–439. https://doi.org/10.1145/2506583.2506590

[14] Palmér, M. (2019). Python Software Foundation Library Input Library. https://pypi.org/project/pynput/

[15] Rivera, N., Baier, J., & Hernández, C. (2013). Weighted real-time heuristic search. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems* (pp. 579–586). St. Paul, MN, USA.

[16] Russell, S. J., Norvig, P., & Davis, E. (2010). *Artificial intelligence: a modern approach* (3rd ed). Upper Saddle River: Prentice Hall.

[17] Wang, K.-H. C. (2011). Massively multi-agent pathfinding made tractable,efficient, and with completeness guarantees. In *The 10th International Conference on Autonomous Agents and Multiagent Systems* (Vol. 3, pp. 1343–1344). Taipei, Taiwan.

[18] Watkins, C.J.C.H. (1989). Learning from Delayed Rewards (Ph.D. thesis), Cambridge University.

[19] White, D. (2007). Clarifications and extensions to tactical waypoint graph algorithms for video games. In *Proceedings of the 45th annual southeast regional conference on - ACM-SE 45* (p. 316). Winston-Salem, North Carolina: ACM Press. https://doi.org/10.1145/1233341.1233398

[20] Yamauchi, B. (1998). Frontier-based exploration using multiple robots. In *Proceedings of the second international conference on Autonomous agents - AGENTS '98* (pp. 47–53). Minneapolis, Minnesota, United States: ACM Press. https://doi.org/10.1145/280765.280773