Lifelong Planning: A Comparative Evaluation of Route Replanning In Partially-Observable Domains

Hans Walter Behrens, Bharath Gunari, Rishabh Hatgadkar, Nicholas Martinez
Arizona State University
Tempe, Arizona 85281–3673
{1211230537, 1213217679, 1215005506, 1207024399}
{hwb, bgunari, rhatgadk, nlmarti4}@asu.edu

***Abstract*—Route-finding has historically played an important role within computer science, but often assumes that the region or graph to be traversed is fully known. In practice, this requirement can prove difficult to satisfy; no domain better illustrates this challenge than robotics. As autonomous agents traverse unknown environments, their route-finding must adapt to incorporate new information as it becomes available. In this work, we explore several route-finding algorithms from the literature, including lifelong-planning A*, D* Lite, and naive-replanning A*. We discuss our implementations of these approaches, and assess their strengths, weaknesses, and appropriateness for various scenarios. We conclude by comparing their relative efficacy for autonomous agent pathfinding in partially-observable terrain, as represented by the Pacman pathfinding problem.**

***Index Terms*—path planning, shortest path problem, robot learning, observability**

I. Introduction

One of the major domains of Artificial Intelligence is search, i.e. how a rational agent, given some sensor information and possibly some domain knowledge, searches for a goal state. Within search, there are two types of search problems. In uninformed search, the rational agent doesn't have any heuristics, i.e. it has no idea of which states are "closer" to the goal state. In informed search, the rational agent does have this information. One classic example of informed search is A*, wherein the rational agent uses both backward cost and a heuristic function to prioritize nodes in the fringe. A* search operates under the assumption that the entire search space is known. Therefore, in applications such as robot exploration, where the entire search space is not known, the rational agent must search, carry out the generated search plan, and re-adjust (if necessary) based on discovered obstacles or other environmental factors.

The goal of this research project is to understand the different types of searches that can be utilized in a "goal-directed robot-navigation task" in unknown terrain. In particular, this research project aims to compare these searches in terms of their intuition, implementation, and performance. The three methods that will be compared are the Naive Re-planning A* (NRA*), Lifelong Planning A* (LPA*), and D* Lite.

II. Background

This section will contain a brief technical discussion and intuition for each of the three approaches.

A. Naive Re-planning A*

The NRA* algorithm is an adaption of the classic A* Search to an environment that is partially observable. In an A* Search, a rational agent prioritizes nodes s in the fringe based on a value $f(s) = g(s) + h(s)$, which corresponds to the cost to reach the current node, plus the heuristic cost to reach the goal. In NRA*, the only information that the agent can perceive are the four adjacent cells in the assumed grid-world environment. Therefore, the NRA* agent generates an optimal path, and follows it until one of two things occur: (a) it has reached the goal or (b) it encounters a previously-unobserved obstacle. In this second case, it updates the edge weights into the obstacle square, re-runs A* search from its current location, and repeats the process as before. N.B. that when this re-planning takes place, any previous route knowledge that might have been computed previously does not carry over to the next search, resulting in significant recomputation.

B. Lifelong Planning A*

The LPA* search algorithm is based on an incremental heuristic search [1] [2]. Unlike NRA*, LPA* is able to use information from previous routefinding computations to inform new searches. To do so, LPA* keeps track of $g(s), rhs(s)$ 2-tuples for each node s in the graph. Intuitively, $g(s)$ is an estimate of the cost so far, and corresponds to the same value from A* search. $rhs(s)$ is a one-step look-ahead estimate that takes into consideration the predecessors s' of s, potentially providing a better estimate than $g(s)$. More formally, the rhs is defined as 0 if $s = s_{start}$ and $min_{s' \in pred(s)} g(s') + c(s', s)$ otherwise. The queuing strategy for fringe nodes is more complex than in (NR)A*, as it uses a tuple of keys that establish a dual-priority system. Since the $rhs(s)$ values are a "potentially better informed" version of the $g(s)$ values, a node is called locally inconsistent if $g(s) \neq rhs(s)$. When a node becomes locally inconsistent, either through initialization or replanning, it is added to this queue.

Initially, only the start state is locally inconsistent. By making the nodes locally consistent, and therefore making sure that the rhs values are precisely the same as the g values, an optimal path can be found based on the currently-known edge weights. Upon discovering environmental changes which affect the edge weights between nodes, it updates those affected nodes, and makes them locally inconsistent (or in other words,

adds them to the priority queue) if necessary. It then repeats the pathfinding process until the graph is once again consistent. Note that edge weights can increase (locally overconsistent) or decrease (locally underconsistent) in general, but in our example scenario, edges can only increase. This corresponds to the agent encountering an impassable barrier. Once the graph is locally consistent, the shortest path to the goal can be found by starting at the goal state, transitioning back to the predecessor through the nodes which minimize the cost, continuing until the start state is reached.

A key drawback for LPA* is that it does not allow for changing start positions. For our motivational scenario, where the agent must directly observe the updated edge weights, this can cause significant backtracking.

C. D* Lite

D* Lite [3] [4] aims to improve on this by only updating nodes on the path from the in-progress current position to the goal state. To do so, modest changes to the LPA* algorithm are required; however, the core replanning approach of LPA* is maintained.

LPA* iteratively attempts to find shortest paths from the start to the goal, incorporating new edge costs as it observes them through the local consistency approach described above. In contrast, D* Lite iteratively attempts to find the shortest path from the $current$ node to the goal, also incorporating new edge costs as it observes them, but only towards the goal. There are some other differences as well. In particular, the search direction in now switched, and the $g(s)$ values now represent the goal distance, not the backwards cost. Similarly, the rhs values are now forward looking, i.e. one step look-ahead with respect to successors and their respective goal distances. Similar to LPA*, D* Lite also prioritizes nodes based on a tuple that is defined in terms of $g(s), rhs(s)$. However, the definition of this tuple differs from that of LPA* in that its priorities are effectively lower bounds on those priorities from LPA*. It also maintains a priority offset k_m, which allows it to bound the replanning propagation. This re-formulation, along with a less strict terminating condition allow D* Lite to have better performance than LPA*.

The framework for D* Lite is as follows: it first attempts to compute the shortest path from the start to the goal node. It then takes steps along this path, and makes any changes to the edge costs and associated priority queue values when necessary. It then computes the shortest path using this new information, and repeats. The currently-optimal shortest path to the goal can be found by starting at the current start state, transitioning to the successor which minimizes the rhs condition, i.e. one step look-ahead, and continuing until the goal state is reached. However, to extract the actual path taken by the agent including replanning, this information must be extracted retrospectively, by recording the steps taken.

III. Implementation Details

All implementation was done in Python2, for ease of integration with existing systems, namely the Pacman domain.

A. Pacman

A common context for measuring pathfinding and route-planning algorithms is through the classic arcade game, Pac-man. By adjusting different aspects of gameplay, such as the presence of ghosts, fore-knowledge of wall locations, density and placement of food pellets, and so on, a wide variety of heterogeneous environmental configurations can be used to explore different avenues of route planning.

In this scenario, we assume a grid-world that can be traversed in 4 directions, $\{N, S, E, W\}$, which is surrounded by impassable barriers. The agent is unaware of any additional walls, but can observe walls immediately adjacent to it. Since there may be large numbers of walls in complex configurations, this is roughly analogous to a blind person finding their way through a labyrinth by touch, a daunting task.

B. Naïve-Replanning A*

Naïve-Replanning A* describes a strategy where the agent first generates a shortest-path estimate to the desired goal using the well-known A* algorithm [5], and then follows that path. As the agent moves through the partially-observable environment, if (and only if) the route is interrupted by any impassable barriers (such as walls), it updates an internal representation of the environment with that information. It then generates a new shortest path from scratch using A*, using its current location as the start location, the original goal as the new goal, and the updated environmental representation as a modified graph.

This approach is called "naïve" because it does not re-use any of the previous planning computations for later replanning, which can lead to re-exploration of many states, and unnecessarily duplicated computation. Additionally, although A* is guaranteed to find the best route given the known environmental information, NRA* is not; this is because it operates with missing information, and can lead the agent down a suboptimal path.

C. Lifelong-Planning A*

Lifelong Planning A* was implemented by creating a generalized implementation, then by making small changes to interface it with the Pacman domain. This was accomplished using object-oriented design principles. In particular, the LPA* object must be instantiated with the maze dimensions, the start location, and goal location. It maintains a list of so-called naive walls, which are the walls the agent is certain exist. We have implemented a mechanism with which the agent can "make", or learn, a wall in the naive walls so that its knowledge about the maze can be updated and persist. This wraps the LPA* object: initially supplying information, iteratively extracting path steps from the object, and supplying information about the true walls to the object. Since the LPA* agent must backtrack to compensate for the unchanging start position in LPA* (i.e. revisit parts of the maze multiple times), care had to be taken to make sure that this backtracking behavior was captured in the final path returned for visualization. Each time new information was supplied to the LPA* object, edge

weights were updated, and the LPA* object would generate a new route automatically if needed.

One of the challenges that the group faced with this part of the implementation was representing the search process that the Pacman took. Instead of simply returning the final path, if there was a backtracking situation, the backtrack path had to be appended to the existing path without any overlap. Rather than forcing the agent to return to the start position in these cases, we instead intersected the inverse paths to the furthest downstream point, reducing extraneous movement if possible. Extensive debugging was required to provide this functionality.

D. D* Lite

Similar to the LPA* implementation, the D* Lite algorithm was also implemented in an object oriented manner. The overall search again acted as a wrapper to the D* Lite object, initializing it, and iteratively extracting information from and updating information in the object. However there was a subtle difference in how the D* Lite object operates. With the D* Lite object, since the start position corresponded to the agent's current position, we could simply take a step with the object, learn information from the perceived walls, and repeat. Therefore, no backtracking had to be taken into account because the D* Lite algorithm calculates the shortest path from the current node rather than from the start location.

Another specific challenge that the group faced with both LPA* and D* Lite integration were the initial differences between the core algorithm and Pacman domain. In particular, the core algorithm used a coordinate system that was transposed with respect to the Pacman domain. Additionally, the core implementation returned a list of coordinate points, rather than actions. Both of these discrepancies had to be accounted for in the code. Finally, the Pacman domain maintains a count of nodes expanded, each time a call to get successors is made. Since our LPA* and D* Lite core did not interact with get successors, an internal expansion count had to be maintained so that it could be tracked as a metric for evaluation.

E. Test Coverage

To ensure correct execution of the implementations, unit tests were created in conjunction with the PyCoverage tool. These units tests were used to test the correctness of the implementations against a series of inputs and expected outputs for both the underlying algorithmic implementations, as well as the supporting data structures and algorithms used.

Our test suite provided complete (100%) coverage for all core implementations and supporting libraries which were implemented in the course of this project. Supporting code, such as that provided through the Pacman project, system libraries, or other external sources was judged as outside the scope of our testing strategy.

IV. Evaluation

A. Configuration and Metrics

The four metrics used to measure the performance of the search algorithms were path length, nodes expanded, memory usage, and running time. Path length describes the length of the final path that Pacman took in order to find the goal state. This path is the same path that is visualized in the Pacman environment. Nodes expanded measures the number of nodes that needed to be expanded, i.e. popped from the priority queue in order to complete the search. The nodes expanded metric is linked to both time and space complexity. This is because the more nodes expanded, the longer the search will take. Also, the more nodes expanded, the deeper the queue, and the more space is needed to store those nodes in the queue. Memory use describes the maximum amount of memory used at a given point in time (in MB) by the Pacman Python process. The memory usage was captured using the `valgrind` utility. Running time captures the effective amount of time was needed to guide the agent to the goal location. The runtime was captures by using the linux `time` command.

All of these metrics were measured as a function of maze size, i.e. the effective area of a the Pacman maze in test. Note that there is not a direct linear relationship between maze size and search complexity, since the maze size does not capture other factors, such as complexity or wall count. However, the maze size is correlated with search complexity and in general, the larger the maze size, the more complex the search problem.

A* search was also included as a baseline comparison. Note that the A* algorithm has a totally observable environment, whereas the other three algorithms only have a partially observable environment, namely the 4 adjacent cells in the Pacman environment.

The experiments were run using a machine with 8-core Intel i5 1.6 GHz processor and 12 GB of RAM.

B. Experiments and Discussion

We executed and collected metrics for each search algorithm against four different environments: the pre-defined tiny (7x7), small (22x10), medium (36x18), and big (37x37) mazes of the Pacman domain. Results were tabulated, graphed, and the decision was made to switch the units to logarithmic for better visibility. Each approach was executed once, as the pathfinding process for these algorithms is deterministic.

For all algorithms, the path lengths increase with respect to the maze size (see Figure 1). The A* algorithm has the best performance as expected, followed by NRA* and D* Lite, and finally be LPA*. The explanation for the low performance of LPA* is that it continually backtracks to the intersection point, which contributes to the path cost.

For all algorithms, the number of nodes expanded increases with respect to the maze size (see Figure 2). The order of performance from best to worst is as expected: A*, D* Lite, LPA*, and NRA*. The low performance for NRA* can be explained by the re-planning nature of its implementation. Every time it hits a wall on its expected path, it re-plans from scratch, i.e. the nodes explored and nodes in its fringe are not carried over when calling A* again. This is what LPA* and D* Lite aim to avoid.

For all algorithms, the amount of memory increases with respect to the maze size (see Figure 3). The order of performance

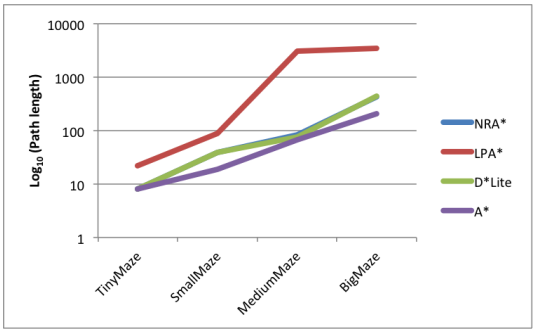

Fig. 1. Path Length v. Maze Size

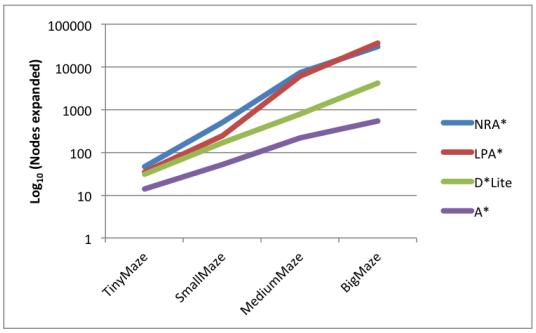

Fig. 2. Nodes Expanded v. Maze Size

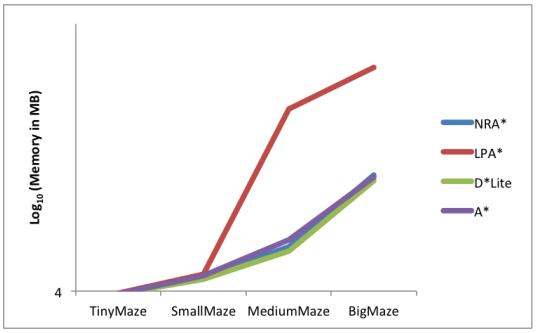

Fig. 3. Memory v. Maze Size

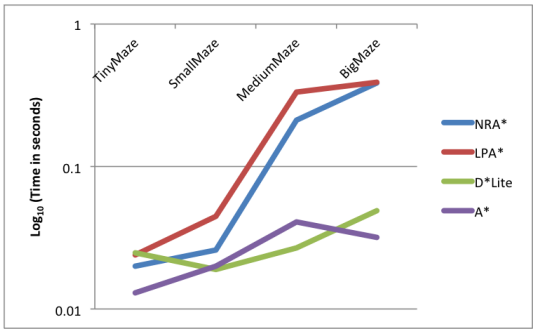

Fig. 4. Runtime v. Maze Size

from best to worst is: D* Lite, NRA*, A*, and finally LPA*. D* Lite slightly out-performs all other algorithms because of specific measures it takes to only re-evaluate those nodes that will have an affect on the final path. Again, the low performance of LPA* is due to the backtracking nature. This is somewhat counterintutive, as A* and NRA* do not maintain in-memory structures of the same complexity as the other two; we conclude that the memory usage is likely dominated by the Pacman domain implementation itself, as memory usage and path length are highly correlated.

For all algorithms, the runtime increases with respect to the maze size (see Figure 4). A* and D* Lite are the best performers, with alternating dominance. Interestingly, this shows that even if the agent has partially observable environment information, it can potentially perform just as fast or faster than an agent with total observability of the environment, using the right strategy, namely D* Lite in this case. This is because A* follows a "perfectionist" strategy, and finding the optimal route may take computational longer than the "good enough" route found by D* Lite. These are followed by NRA* and finally LPA*. These two struggle due to their re-planning or backtracking natures, respectively.

Overall, D* Lite is the most competitive of the re-planning algorithms, especially in computation time. LPA* was not as competitive as we expected, primarily due to the backtracking process necessitated by the Pacman domain integration.

V. Conclusion

In this research, our team explored the intuition, implementation, and evaluation of three search strategies for agents in partially-observable (specifically eight-connected) environments. These search strategies were integrated with the Pacman domain, which is a representative application for robot navigation search tasks. After evaluating the three search strategies (in addition to A*), the group confirmed that the D* Lite strategy is the optimal re-planning strategy. The D* Lite builds off the theoretical foundation of LPA*, but makes signification optimizations to the implementation that allow it to dominate in terms of performance. The NRA* strategy is competitive in terms of path length, but its simple strategy hurts its performance in all other categories. When a totally observable environment is not practical, D* Lite is the best search strategy of the assessed algorithms.

VI. Team Contributions

Hans implemented the core LPA* and D* Lite algorithms, unit tests, provided editing to the report, and assisted with integration with Pacman domain. Rishabh worked on integrating the core algorithms with the Pacman domain, implemented NRA*, and contributed to the report. Nicholas assisted with the Pacman integration, evaluation, tabulation and graphing of results, and was responsible for most of the report. Bharath assisted with a wide variety of project tasks as needed.

Acknowledgment

We would like to thank Mehrdad Zaker Shahrak and Yu Zhang for sharing their valuable insights for this work.

REFERENCES

[1] S. Koenig and M. Likhachev, "D* lite," *Aaai/iaai*, vol. 15, 2002.

[2] S. Koenig and M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," in *ICRA*, 2002.

[3] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning a*," *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004.

[4] R. Simmons and S. Koenig, "Probabilistic robot navigation in partially observable environments," in *IJCAI*, vol. 95, pp. 1080–1087, 1995.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, 1968.