The Quadcopter World

Julian Straub - JulianStraub@gatech.edu

Problem Definition

In this project we want to investigate in the problem of autonomously exploring unknown terrain with a quadcopter that is able to perform Simultaneous Localization and Map building (SLAM) using feature extraction from the images collected by a downward-facing camera. Such a system could for example help to build a fast map of a disaster zone like a collapsed building to provide first responders with the necessary overview over the situation. Another possible area of deployment would be the military, where it could help reduce the risk for soldiers and the danger for citizens in a battle region by providing a better overview for the operation controllers.

While it is not necessary for this mini project to know exactly how the feature extraction and SLAM computation is done, it is important to realize that the ground over which the quadcopter navigates has different textures. Depending on those, the features extracted from the camera-images are of different quality - "good", "okay" or "bad". Since the quality of the SLAM output depends on the quality of the features, the quadcopter should fly over good features as long as possible. Also obstacles to either side of the aerial vehicle can be detected and have to be avoided as they are explored on the way to a goal position in unknown environment.

Related Work

The problem described above falls into the category of goal directed path-planning in an unknown environment. After discretizing the world, this problem can be solved in a naive approach by re-planning the path with a search algorithm like A*[1], whenever a new observation is made. Such an approach generates a lot of overheat computations since not necessarily all parts of the path need to be re-planned. The sensors' range is limited and thus new observations only effect the cost of nodes in very close proximity to the quadcopter. Hence the path near the goal usually needs not to be re-planned. Several algorithms like Adaptive A* (AA*), Lifelong Planning A* (LPA*) and D* Lite[2] have been proposed exploiting this fact to reduce the time for re-planning.

Approach

In the following it will be described how the environment of the quadcopter was abstracted and simulated with Matlab and C++ code. After that two algorithms will be proposed to solve the problem of finding a path to a goal through an unknown environment.

Feature quality	Good	Okay	Bad
Cell cost	1	2	4

Table 1: Feature quality to cell cost relation

Simulating the Environment

For the purpose of this mini project, the world is represented by a grid of cells. Each of those is either occupied by an obstacle or is free and has a texture with "good", "okay" or "bad" features. The cost of a cell has to be understood as the cost for entering this area of the world. Impassable cells are associated with infinite cost and therefore avoided by the search algorithms. To account for the unwanted traversal of cells with "bad" features, a higher cost will be assigned to those once they were discovered. Consequently cells with "good" features will be associated with a low cost. In this example the costs were picked such that it is for example cheaper to move around a "bad" cell if it is surrounded by "good" ones. For a real deployment one has to determine how strongly "bad" cells should be avoided by measuring the negative impact of those on the overall performance of the SLAM system. Table 1 sums the relation between feature quality of a cell and the cost for entering it up and gives the quantities used in the implementation.

In reality the viewing angle of the downward facing camera is limited. This is simulated in that the quadcopter can only sense the status of the cell he is on and the stati of the directly adjacent ones.

The robot always knows its position in the grid-world precisely due to a perfectly working SLAM system. Out of convenience the movement was restricted to one cell per time-step in the East, West, South or North direction. However, the system could be adapted to a higher degree of freedom in the movement easily.

In the concrete implementation a Matlab script is used to generate the grid-world with its cells of different feature quality randomly. The ratio of feature quality in the grid is good:okay:bad = 1:2:1. Obstacles are added around the square sized world and at predetermined positions as can be seen in Figure 4. The side length of the world can be chosen to an arbitrary number of cells. Once this map is generated it is fed into a C++ program that simulates the step by step discovery of the world for the search algorithms. Unexplored cells are assumed to have features of "okay" quality. The robot updates its grid-world representation once a cell's feature quality was observed and does not forget about that anymore.

Searching for a Path

The objective is to reach a specific given goal position without any prior knowledge of the environment. The fact that the robot gains information about the world only incrementally and in the limited area of its sensor, makes it necessary to obtain a new path whenever new observations are made. This is because it is not guaranteed anymore that the old path is viable or optimal under the new information.

In order to be able to search for a path to the goal the representation of the world must be reinterpreted as a graph where a node represents a cell in the grid-world. In our case the cost for entering a node can be different from leaving it. The cost of entering a node is determined by the feature quality of the respective cell in the grid.

Repeated A*

A naive approach to solve this problem is to use the A* algorithm to re-plan a path to the goal as soon as the feature qualities of new cells are observed. In order to get a better feeling for the algorithm, the standard A* [1] was implemented in C++ from scratch. It uses hash maps and a priority queue to efficiently store the nodes expanded by the A* algorithm. The heuristic for the A* search is the L_0, also called Manhattan, distance metric.

The algorithm to travel from the start to the goal position can be summed up as follows:

1. Sense surrounding environment.
2. Run A* to find path from current location to the goal position if unexplored cells were observed.
3. Move one step on the path and start over.

The downside of this approach is that the path is recomputed completely after every new observation, although those are only in immediate proximity to the robot and do not strongly influence the path near the goal.

D* Lite

D* Lite makes effective use of this idea. It keeps a memory of cost estimates from visited cells to the goal. After each new observation only the estimates are updated which have changed. Using those cost estimates a path to the goal is found by a simple greedy search which can be performed very fast. The other idea that makes D* Lite fast is that the search tree is rooted at the goal and not at the current start position of the search. Therefore the search tree remains mainly the same over time. It only changes near the leaves, since observations are made locally around the robot.

The implementation of D* Lite is a modified version of a re-implementation by James Neufeld [3] of the unoptimized D* Lite as described in [2]. The modifications were necessary to allow the use of the specifically weighted graph that represents the quadcopter world. The standard implementation does only distinguish between obstacles and free space. Also the heuristic for the search algorithm was changed to the L_0 distance metric.

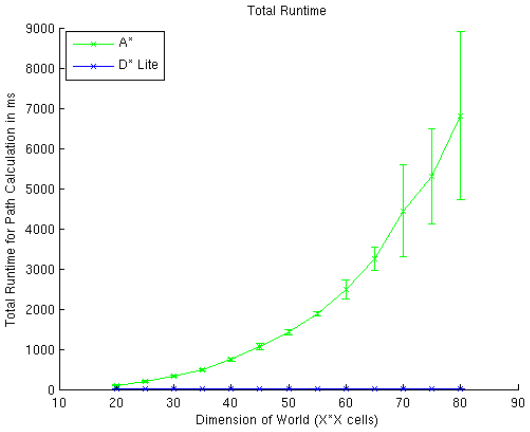

Figure 1: Statistics of the time it took the robot to reach the goal position per run. The lines connect the means and the error bars indicate the standard deviation of the time measurements.

The algorithm to travel in the simulation from the start to the goal position using D* Lite is the same as the one described above for A* but with D* Lite as search algorithm.

Evaluation and Discussion

The evaluation of the two different algorithms, repeated A* and D* Lite, will focus on the speed of the algorithms and on the quality of the paths. These are the two properties that are most relevant for the task at hand. Fast movements and limited computational resources on the quadcopter make it desirable to have a efficient and fast path planning algorithm. But at the same time we would prefer a path that leads through areas with good features to foster the quality of the SLAM system.

The total time it took each of the two algorithms to move the robot in the simulated environment from the start to the goal position was used as the metric for efficiency and speed. In all runs, the starting position was at $[0, 0]$ and the goal at $[worldLengthX - 1, worldLengthY - 1]$. For both algorithms this time was measured ten times each on world sizes from 20*20 cells up to 80*80 cells.

Figure 1 depicts the statistics of those measurements. Clearly, the repeated A* algorithm scales badly with increasing number of cells in the grid-world. The quadratic form of the curve originates from the quadratic increase of cells. In contrast to A*, the total time per run for D* Lite is much less - depending on the number of cells at least two orders of magnitude. In the close up of the curve for D* Lite in Figure 2 it gets clear that D* Lite's total run time also increases.

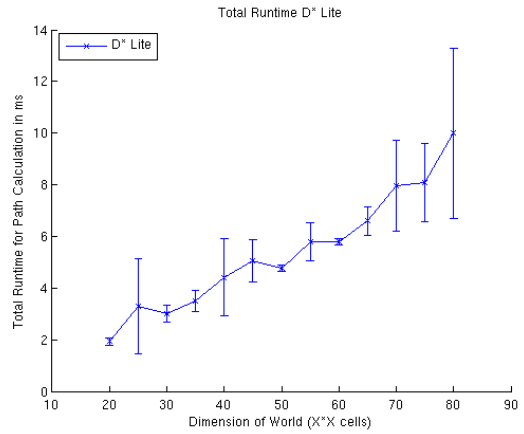

Figure 2: Statistics of the total time per run for D* Lite. The line connects the means and the error bars indicate the standard deviation.

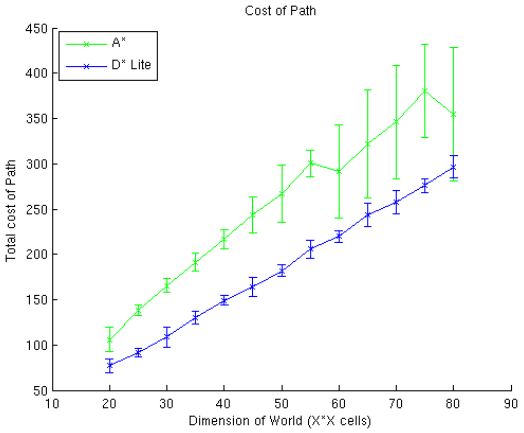

Figure 3: Statistics of the total path costs for both, repeated A* and D* Lite. The means are connected by a line and error bars indicate the standard deviation.

As a metric for the quality of the paths the accumulated cost of the traversed cells was chosen. The less this value is the better are the visited cells and the shorter is the path. The start and the goal positions were picked as in the evaluation of the algorithms' speed. Again, both algorithms were run on randomly generated worlds in the sizes from 20*20 cells up to 80*80 cells. Figure 3 depicts the statistics of the costs of the computed paths. While the path costs naturally increase with the dimension of the world, it gets clear that D* Lite finds generally paths with lower cost.

From Figure 4 we can deduce why A* performs worse. The fact that A* plans from the current position to the goal leads to a longer path since the robot gets stuck behind the upper obstacle. Hence it has to go all the way around it once it discovers that there is no way around it on the right side. In contrast to that D* Lite plans from the goal to the current position and is therefore able to avoid getting stuck at that position.

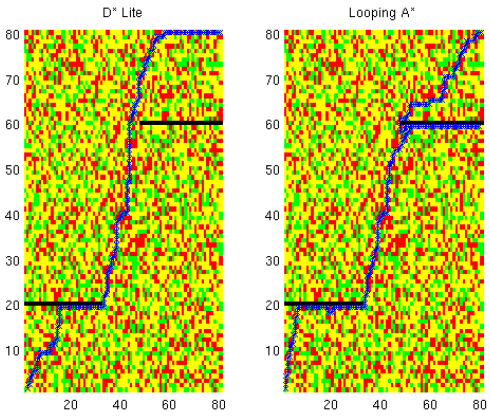

Figure 4: Example paths found by D* Lite (left) and repeated A* (right) depicted in blue. Obstacles are in black, "good" features in green, "okay" features in yellow and "bad" ones in red.

Conclusion and Outlook

All in all two possible algorithms, namely A* and D* Lite, that are able to solve the problem of navigating a quadcopter to a goal in an unknown environment were implemented and evaluated. It was found that D* Lite was superior to a repeated A* with respect to the two performance criteria computational speed and quality of the path. Especially in larger search spaces D* Lite is several orders of magnitude faster than repeated A*. Here D* Lite profits from only having to recalculate the cost of some changed nodes whereas A* does always a complete re-computation of the path. This gets quadratically more expensive with increasing side length of the grid-world.

As future work, it would be interesting to investigate how Rapidly-exploring Random Trees (RRTs) could be used to solve this problem and how efficient they would be. It could be quite challenging to get a path of good quality with them but they would allow to plan in continuos space.

References

[1] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968.

[2] S. Koenig and M. Likhachev. D* Lite. *Proceedings of the National Conference on Artificial Intelligence*, pages 476–483, 2002.

[3] J. Neufeld. C++ implementation of the unoptimized d* lite algorithm. http://code.google.com/p/dstarlite/, 2007.