Improved Path-planning in Zones for Marvin, an Autonomous Vehicle

Elben Shira <eshira@cs.utexas.edu>

May 8, 2008

1 Introduction

This paper is split into two main sections:

1. A formal description of my research and findings. This includes my work on:
   - A*, Adaptive A* and implementation speedups
   - Progress since poster and future work
2. A less formal description of:
   - A comprehensive guide of the planning code base What Ive learned
   - Comments on robotics, autonomous vehicles, and research

2 Abstract

Path-planning is a large and extensively researched problem. From the original Dijkstra algorithm [1] created in the 1960s to the latest Generalized Adaptive A* [3] created in 2008, researchers are still attacking and solving path-planning problems. My main focus was on improving A* [2], mainly through Adaptive A*, a new algorithm put forth by Sven Koenig (USC) et al. A* is an improvement of the Dijkstra algorithm, which was an improvement of the standard breath-first search. A breath-first search (Figure 1) involves looking at every level of the tree until the wanted node is found. Dijkstra improved this by adding a cost-of-path-already-taken value in the algorithm. A* improves on this by adding a heuristics, most common being

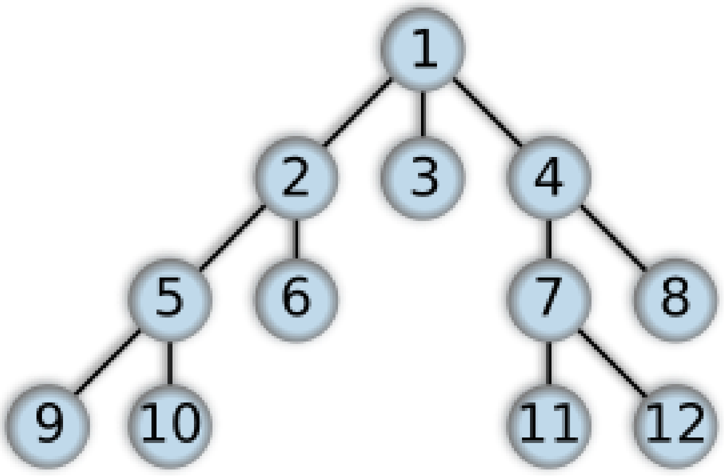


Figure 1: Breath-first search, numbered by order of expansion.

the Euclidean distance from the current state to the goal state. A* always finds the optimal (shortest) path.

Implementing a simple A* approach on a real vehicle, however, is inefficient because A* is a slow algorithm. In a real-world environment, planning every cycle is required to detect changes in the environment such as just-revealed or moving obstacles. Thus, we look at other algorithms such as Adaptive A* and RRT-Connect. Through testing, we find that Adaptive A* and RRT-Connect perform much better than a standard A* in all tested situations including size of environment, density of obstacles, and radius of environment known.

3 Notation

We define the following notation:

- a state is a cell or node in the environment, containing x, y, θ, cost of travel so far, and estimated future cost
- S is the set of states (cells) in the environment
- $x_s \in$ S is the start state
- $x_g \in$ S is the goal state

- $x_n \subseteq S$ are the states that are neighbors of $x \in S$.
- $x \in S$ is the state the vehicle is currently on
- $g(x)$ is the cost of traveling from x_s to x
- $g(a, b)$ is the cost of traveling from a to b
- $h(x)$ is the heuristics function, representing the cost (in Euclidean distance) from x to x_g
- $h(a, b)$ the heuristics from a to b
- $f(x) = g(x) + h(x)$

4 A*

A* finds the shortest path by adding the states surrounding the current state into an open list (a priority queue). Each state inserted into the open list contains a pointer back to the original state. It then pops the state with the lowest f(x) value and expands that state, adding its surrounding states into the open list. Once the goal state is popped, then we have found the shortest path.

The heuristics function of A* is admissible, meaning that it must always be less than or equal to the actual cost to get to the goal state:

$$g(x) + h(x) \leq g(x_g) \tag{1}$$

```
insert x_start into open;
while not a goal state:
    x := open.pop();
    for each n in x_neighbors
        if g(n) < g(x) + g(x, n):
            g(n) := g(x) + g(x, n);
            update back pointer;
            if open.contains(n): delete n
            open.push(n) with f(n) := f(n, x_goal);
follow back pointers from x_goal to x_start to find path;
```

Figure 2: A*

At first, we implemented A* as described in Figure 2. This, however, proves to be inefficient due to two problems:

1. The same states were being pushed in the open list multiple times. If we are at state A with a neighbor B, then if B is popped from the open list, then it would insert A back into the open list.
2. Checking to see if a state n is in the open list (line 8) is a costly linear operation.

We improve this by adding two features into the implementation:

1. Use a closed grid (a 2d array of booleans) to stop multiple insertions of the same state back into the open list. Once a state is popped from the open list, we close that cell.
2. Use a open grid (a 2d array of booleans). When we insert a state into the open list, we mark that cell as true. Checking if a state is in the open list (line 8) now becomes a constant operation.

Testing showed that these two improvements cut the A* runtime by at least half.

5 Adaptive A*

Adaptive A* is similar to A*; is uses a path cost and a heuristics to find the optimal path. Adaptive A*, however, improves on A* by remembering the path costs of previous searches, $g(x_{gprev})$. It uses this knowledge to improve the heuristics, $h(x)$, of the current search:

$$h(x) := g(x_{gprev}) - g(x) \tag{2}$$

```
do A* search;
path_cost := path cost of the A* search
while not at goal:
    for every s expanded by the A* search:
        h(s) := path_cost - g(s)
    do A* search using updated h(s);
```

Figure 3: Adaptive A*

Figure 3 shows a simple implementation of Adaptive A*. Several problems, however, arise with this implementation. Like A*, the heuristics used must underestimate the cost to the goal. If the goal changes (from x_g to

x_{gnew}), we must make sure that the heuristics is still consistent. That is, it still underestimates the goal. This is done by, after (2), doing the following:

$$h(x) := max(h(x, x_{gnew}), h(x) - h(x_{gnew}, x_g)) \tag{3}$$

While Adaptive A* needs no modification when the environment is static or when costs to states increase, an addition is needed for dynamic environments where costs to states decrease. This is called the Generalized Adaptive A* [3]. We did not implement this and as a result, our implementation will not find the optimal path if obstacles move.

```
main():
    counter := 1;
    for all states s, last_update(s) := 0;
    while not at goal:
        init_state(x_start);
        init_state(x_goal);
        g(x_goal) = 0;
        open.pop(x_start);
        A*_search();
        path_cost(counter) := path cost from find_path()
        counter++;
        check environment for new obstacles;
init_state(s):
    if last_update(s) != counter and last_update(s) != 0:
        if f(s) < path_cost(last_update(s)):
            h(s) := path_cost(last_update(s)) - g(s);
        h(s) := max(h(s), h(s, x_goal));
    else if last_update(s) = 0:
        h(s) := h(s, x_goal);
    g(s) := infinity;
    last_update(s) := counter;
```

Figure 4: An improved Adaptive A*

Another problem with the above implementation is that it runs the heuristics modifier greedily. That is, it updates every expanded state, even if it will not be used for future searches. This is inefficient. Thus, we introduce a lazy approach where the h-value of a state is not updated until it is expanded by a future search. We do this by keeping a counter, which keeps track of the number of searches we have done. Using this counter, we keep track of the path cost for each search. We also keep track of which search (via the counter) a state was last expanded by. With this data, we then update the states heuristics only if necessary.

The same problems described above in the A* implementation also applies to Adaptive A*. Though not shown in Figure 4, we included these improvements into our actual implementation and, once again, found great improvements in our runtime.

The following notation corresponds to Figure 4:

- counter is the number of complete searches we have done
- $path\_cost(n)$ is the path cost of the n^{th} search
- $last\_update(x)$ is the last search number some state x was updated

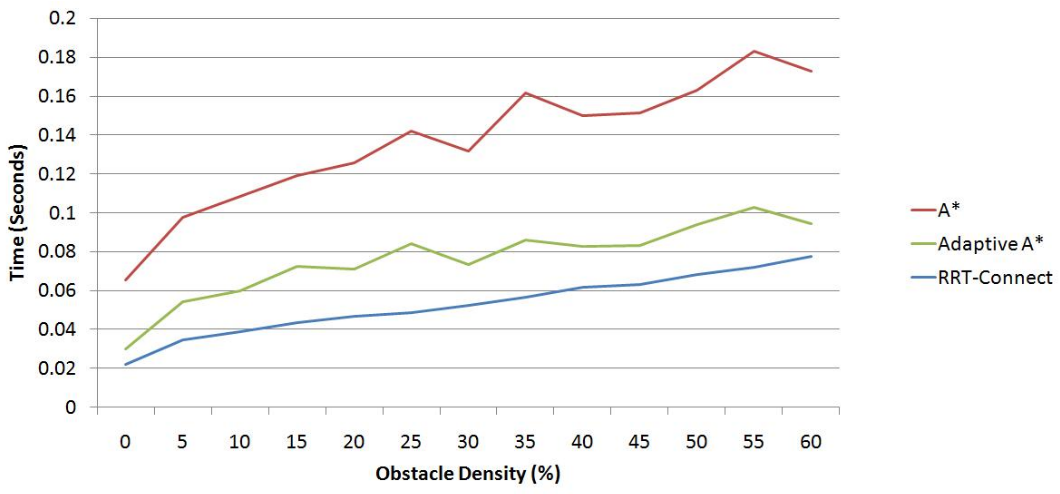
6 Experimental Findings

We now test A* versus Adaptive A* versus RRT-Connect. Our testing environment was defined as:

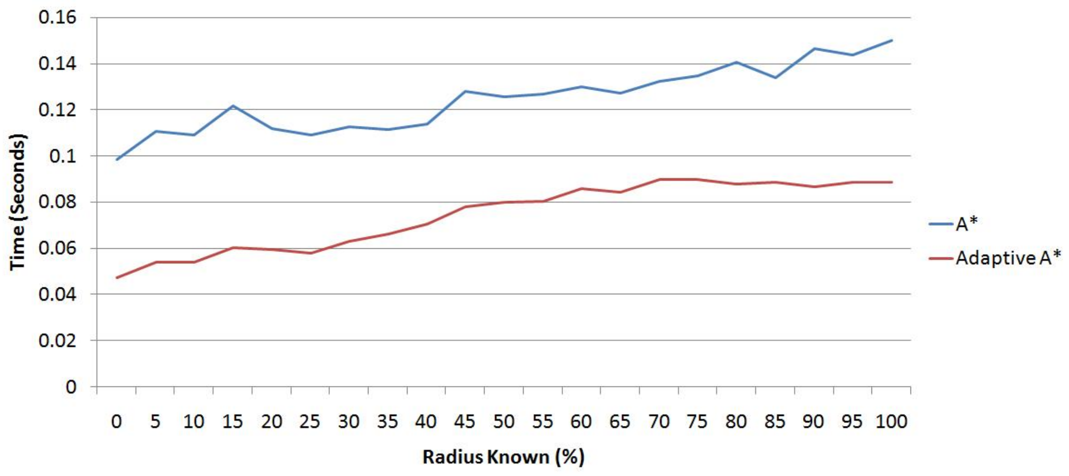
- environment size: 100 x 100
- obstacle density (percentage of obstacles over free space): 30
- re-plan every: 1 step
- runs per environment: 10

The steps per re-plan is always fixed at 1 and we always use the average of 10 runs per test. We then let one variable be a dependent variable for each of our tests. We measured this against time and/or states expanded.

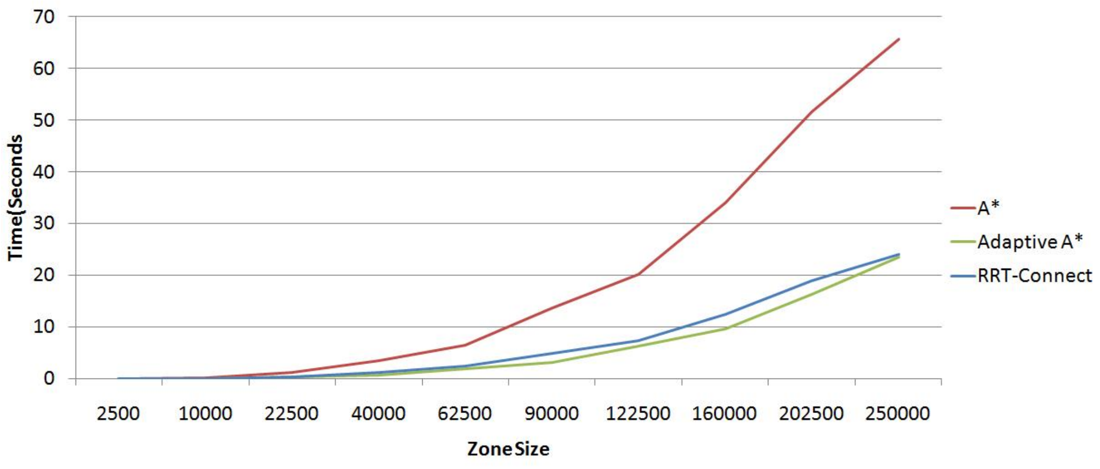
Adaptive A* and RRT-Connect outperforms A* in all situations. We find, however, that RRT-Connect is slightly faster than Adaptive A* in general. RRT-Connect, however, does not find the optimal path.



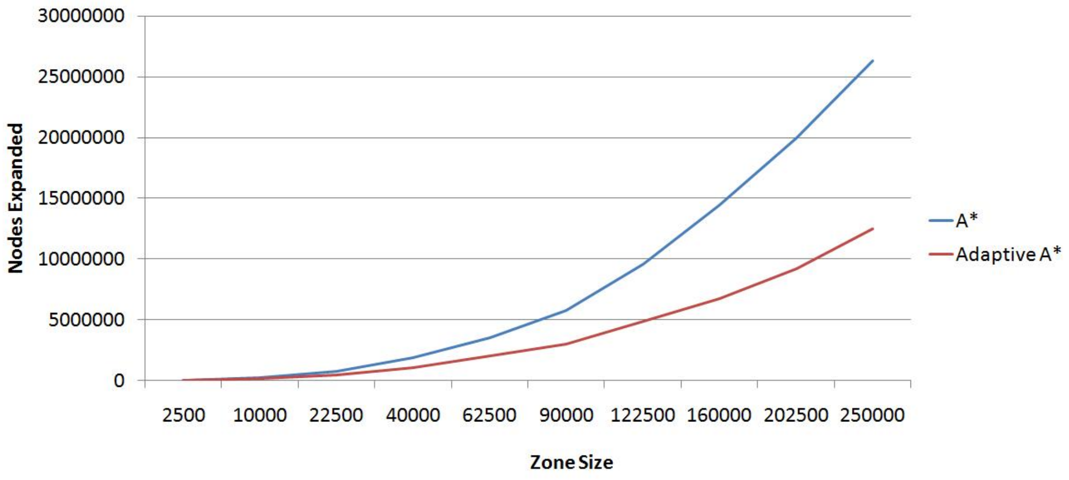
Both Adaptive A* and RRT-Connect outperform A* as obstacle density increases. RRT-Connect is faster than Adaptive A*.



Adaptive A* is constantly faster than A* no matter the percentage of environment revealed to the vehicle at every step.



The best improvement is found as zone size increases. A* degenerates much more quickly than Adaptive A* and RRT-Connect.



As expected, Adaptive A* expands less nodes than A*. This is the true reason why Adaptive A* outperforms A*.

7 Progress Since Poster and Future Work

Most of the semester was spent on finding, learning, implementing, and testing different algorithms on a theoretical and simulated environment. The last two weeks was spent on implementing the Adaptive A* algorithm on the vehicle codebase itself. I first figured out how the algorithm was to be injected into the current system. The vehicle currently uses a Voronoi-type algorithm. It is given a world map and from there, creates a skeleton using the Voronoi algorithm, which consists of possible paths the vehicle can take. The system then runs an A* search on this skeleton. We aim to replace this current system with Adaptive A*.

I ported the current Adaptive A* implementation from Java to C++ (a language I was not familiar with). This took longer than expected because I also had to port the testing tools, which is much harder to port than the algorithm itself.

I will continue working on the vehicle this summer, though I'm not sure what part of the vehicle I want to concentrate on. Seeing that we plan on building the codebase from the bottom up, finishing the zone planner may not be the best idea. Perhaps we would find it best to start on the code rebuild.

8 Miscellaneous Contributions

While I concentrated my work primarily on path-planning, I did look into tool development early in the semester. This included implementing the red/green color scheme found in visualVelodyne, as seen in Figure 5. The idea and implementation was simple, but the results were useful. Not only did coloring the data made visualizing what ground and obstacles were, the color scheming proved that the Velodyne was tilted, which was a problem that the Totem team concentrated on.

9 The Planning Codebase

I will only document the code I uploaded.

- AStar.java - An A* algorithm. The method itself is static, so it is not optimized for replanning.
- AdaptiveAStar.java - Adaptive A* algorithm. Broken into three methods as described in [3]. You must initiate an AdaptiveAStar object.
- Grid.java - Has many useful methods for manipulating 2d int arrays. One of the methods can load and image and return a 2d int array.
- Node.java - Represents a state of the world. Used by AStar.java and AdaptiveAStar.java.

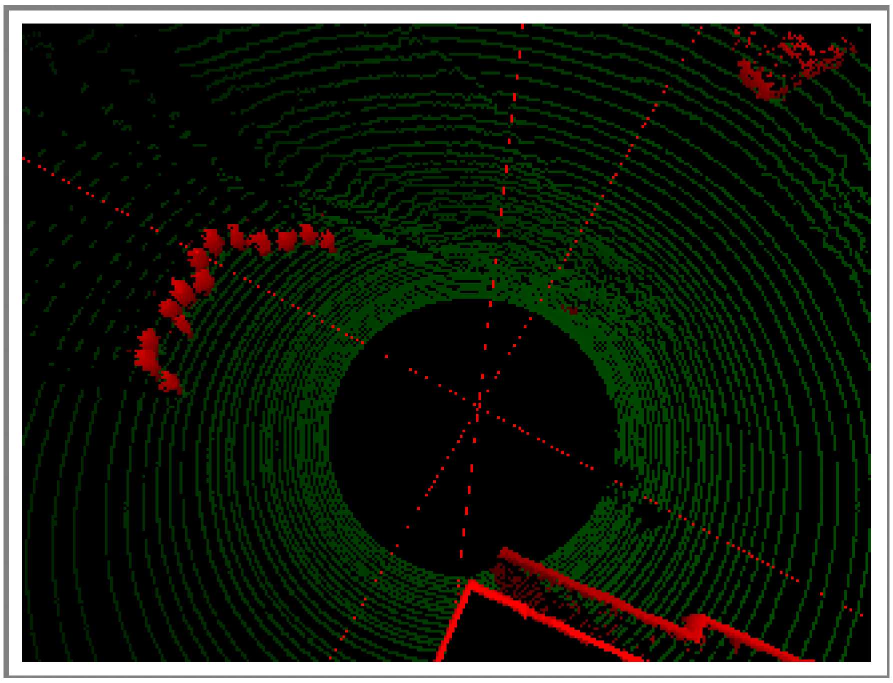


Figure 5: An improved visualVelodyne

- Observer.java - Used to get data (states expanded, run time) from the searches. Used by AStar.java and AdaptiveAStar.java.
- Stopwatch.java - Used by Observer.java to get run time.
- TestEnvironment.java - Main testing suite for A* and Adaptive A*. Can run multiple tests automatically. Currently does not output margin of errors (this can be easily implemented).
- **/astar_cpp**
  - test.cpp - Testing suite. Used while porting.

10 Comments About This Course

In the beginning of the semester, this course both excited and terrified me. The topic seemed fascinating and stirred me; there was so much I wanted to know and learn about. I was also intimidated. Research, after all, is what graduate students and professors do, not an inexperienced and naive freshmen. But I decided to stay in this class because, well, I like being challenged. I enjoy the thrill of entering strange and foreign lands, of exploring the unknown.

I have learned many things through this class. Though robotics is obviously an extremely large field, this course gave me a good overview of the problems found in robotics. I especially enjoyed the lectures over different topics such as vision, closing the loop, and PID controllers. Yes, the lectures were simple introduction lectures, but they have influenced me in many different ways. I, for example, decided to take M 362K: Probability next semester because the closing-the-loop lecture made me realize how important probability is in robotics (a field I am very interested in) and computer science in general. Our discussions about vision also greatly influenced the way I look at robotics and AI. I never realized how complex and powerful we as humans are. Our ability to subconsciously process data obtained from our sensors is amazing. Computers may compute numbers faster than us, but we still reign in our ability to use heuristics and analyze stimuli. This revelation may not tie with the vehicle and planning algorithms, but its these small new-found perspectives that made this course such a sensational experience.

In the beginning of the semester, I wanted to work on creating and improving the tools. Because of this, I spent most of my time looking at the old tools and the accompanying source code. My most useful contribution here was colorizing visualVelodyne. While it was a simple edit, this idea proved useful, especially for proving that the Velodyne was, in fact, tilted. The Velodyne LIDAR team also found this improvement useful. They now plan on expanding this by adding gradients and coloring negative obstacles blue.

I moved away from tool development because I wanted to work on something more theoretical. Writing a log visualizer would have been a great project (which was what I wanted to do at first), but I wanted to learn more about robotics and AI itself. So I chose path-planning.

11 References

[1] E. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269271, 1959.

[2] 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 SSC4 (2): pp. 100107.

[3] Generalized Adaptive A*, Xiaoxun Sun, Sven Koenig and William Yeoh, Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), Padgham, Parkes, Mller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. XXX-XXX.