Name: Eric Chase
UNI: eac2242

COMS W4995 Project Proposal

Idea

For my project, I plan to write a Haskell implementation of the D* Lite algorithm. Developed by Sven Koenig in 2002, D* Lite is a search algorithm in the vein of A* search whose aim is to determine a path between an agent's location and a designated goal state. Unlike A* search, however, D* Lite can importantly be used for navigation in unknown terrain, meaning that the traversability of states are not known ahead of time. For this reason, D* Lite is utilized predominately in applications of robot navigation, whereby the locations of obstacles are determined on-the-fly using the robot's sensors.

As a high-level description, the D* Lite algorithm initially assumes that the state space does not contain obstacles. With this assumption in mind, it calculates an ideal shortest path to the goal state. The agent then follows that path until a new obstacle is encountered. At this point, heuristic information is updated and new arc costs are calculated between locally affected states. The shortest path is replanned, and the search continues until the goal state is reached.

This is an algorithm that I learned of through my research with Professor Tony Dear on multi-agent collision resolution. I believe that writing a Haskell implementation of it would be fun, a step more rigorous than implementing plain A* search, and most importantly, a good candidate for parallelization like other search algorithms.

Roadmap

For the purposes of this project, I plan to make a few simplifying assumptions about the problem setup, namely that the map adheres to a simple grid structure and the agent is limited to 4-way movement only (north, south, east, and west). Additionally, I intend to use Manhattan distance as the heuristic utilized by the algorithm.

As for implementation details of my project, the priority queue of states used by the algorithm can be modeled using a module like Data.Heap. Furthermore, because this queue is only updated occasionally in the case of an obstacle encounter, its state can be maintained efficiently using the State monad. Additionally, a map keyed by coordinate-pairs can be used to store the state-specific statistics used by D* Lite. With exception to the details listed above, I want to use functions from the Standard Prelude as much as possible. For user input, the program's top-level function will require arguments corresponding to the size of the map, the obstacle locations, and the start and goal states of the robot. In order to highlight the fact that the solution returned by D* Lite changes

dynamically as new objects are discovered, the successive paths proposed by the algorithm and the final solution (if there is one) will be logged to the console at termination.

Finally, I will discuss some points where I believe parallelism could benefit my project. Though D* Lite proposes a new path to the goal with each obstacle encountered, printing these paths is not a trivial matter. This is because, under normal execution of the algorithm, obsolete paths are never fully revealed, as the solution is updated in place, overwriting part of the previously proposed path. Thus, in order to recover the proposed paths in their entirety, each must be followed to the goal before updating the priority queue (which effectively creates a new path). Though this added procedure is not part of a vanilla D* Lite implementation, I believe that it is important in order to better illustrate what the algorithm does. Luckily, this is the kind of job that could be run in parallel perfectly using a copy of the old priority queue, while the other threads continue with the execution of the algorithm. There are also some smaller optimizations within the D* Lite algorithm that parallelism allows for. For instance, the fact that the priority queue is formed according to a tuple means that each element of this priority value could be evaluated simultaneously in parallel.

Additionally, when it comes to search algorithms, there is generally a lot of room for improvement via parallelization due to their recursive nature. For example, when a state is removed from the priority queue in D* Lite, all of its successor states must be updated and potentially removed as well. Each of these are operations than can safely be performed in parallel, with a separate thread handling each successor. I suspect that these parallelization efforts will result in much faster execution times, especially for larger maps and maps with many obstacles.