Name: Eric Chase
UNI: eac2242

COMS W4995 Project

Overview

For my project, I wrote 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, state-specific 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.

Implementation

This section details the implementation specifics of my program.

First, the D* Lite algorithm uses a priority queue to hold states whose heuristics may need to be updated. I found that none of the readily available priority queue implementations for Haskell fit my needs for D* Lite's priority queue, so I wrote my own priority queue data type called MyPrioQueue in myPrioQueue.hs (using a hash map and set), along with a number of functions to modify an instance of my queue type. Importantly, writing a custom priority queue implementation allowed me to define and optimize certain operations that the algorithm performs frequently:

- The *remove* function removes an arbitrary state from the queue in O(log*n*) time, where *n* is the number of items in the queue. There is no analogous function defined in the Data,Heap or Data.PQueue packages.

- The *insert* function adds a state-priority pair to the queue in O(log*n*) time (same as Haskell’s existing implementations).
- The *findMin* function returns the state-priority pair in the queue with the minimum priority. It runs in O(*n*log*n*) time. The analogous function in Data.Heap runs in constant time, but only returns the minimum value without its associated priority.
- The *member* function returns a boolean indicating whether a state is present in the queue in constant time. There is no analogous function defined in the Data,Heap or Data.PQueue packages.
- The *empty* function returns a boolean indicating whether the queue in empty in constant time (same as Haskell’s existing implementations).

My implementation of the actual D* Lite algorithm is defined within dstar.hs. As outlined in my proposal, the file contains a top-level function that expects input from the user corresponding to the size of the map, the start and goal states of the search problem, and the obstacle locations (specifics on how to run the program are presented in the README). It prints all of the successive paths proposed by the algorithm during the course of its execution. The final path printed to the console is a path from the start to goal states that navigates around all of the map’s obstacles.

Additionally, using record notation, I defined a data type called SearchState that contains all of the parameters of the search problem. The state monad is crucial to my implementation, allowing the search parameters to be passed between the D* Lite functions in the form of a SearchState. Each of the search parameters are described below:

- s_queue is the priority queue of the algorithm.
- s_stats holds state-specific statistics. It is a map between states (coordinate pairs) and pairs of doubles (g and rhs values) that are used to determine priority.
- s_obs is a set containing the locations of all obstacles in the map.
- seen_obs is a set containing the locations of all previously-encountered obstacles.
- s_km is a value that is added to priorities in order to prevent having to reorganize the priority queue at any point.
- s_start is the current position of the robot.
- s_start0 is the starting position of the robot.

- s_goal is the goal location of the robot.
- s_last is the previous position of the robot.
- s_height is the height of the map.
- s_width is the width of the map.

Finally, I tried to stick relatively closely to Koenig's original outline in my implementation of the D* Lite algorithm. The key functions of the algorithm are described below:

- *driver* is the main driving function of the algorithm. Each iteration, it moves the robot to a new state and scans for obstacles. If any obstacles are found, the queue is updated by calling *updateQueue*. Once updated, the algorithm may propose a new path (if the previously unknown obstacle blocked the progress of the previous path). The new path is determined by calling the *computeShortestPath* function. This process continues until the goal state is reached, when all of the proposed paths are returned.
- *updateQueue* is the function that updates the algorithm's priority queue when an obstacle in encountered. It loops until there are no longer any states in the queue that need their information to be updated.
- *updateVertex* receives a state as input and updates the specific attributes of that state. This function is called for each state that *updateQueue* iterates over. It can modify the state's statistics (g and rhs values) or update its placement in the priority queue.
- *computeShortestPath* receives a copy of the search parameters and builds the algorithm's current proposed path by following the cheapest transitions from the start to goal state. It returns this proposed path.

It is important to note that the program expects proper input from the user. Namely, the start, goal, and obstacles must reside within the extents of the map (for example, (11, 11) is not a valid location if the map size is 10 x 10), and there must be a valid path to the goal (meaning that no obstacles reside on the start or goal states, and there exists a path such that the robot can reach the goal from the start state). Cases of malformed input will result in undefined behavior.

Parallelization

NOTICE: Although I was unable to achieve a speedup of my program through parallelization, the following section will serve as documentation of what I attempted.

Originally, I proposed two main areas of interest in my D* Lite implementation that I believed could benefit from parallelism:

1. Parallelizing actions that must be performed on all child/successor states.

When a state is removed from the priority queue in D* Lite, its four successor states must be updated and potentially removed as well. In my project proposal, I speculated that each of these operations could safely be performed in parallel, with a separate thread handling each successor. In my code, the action of updating a state's children is performed using map operations. Thus, my attempt to parallelize this process involved parallelizing these map operations. Two different types of map operations were present in my code:

A. Calls to *mapM_*

Outside of printing all proposed paths at the end of the program's execution, all calls to *mapM_* are used to iteratively apply the *updateVertex* function to a state's children. These calls appear in the *scanUpdate* function, which is called when new obstacles are discovered, and the *updateQueueHelper* function, which is called by *updateQueue*.

Once the algorithm was implemented, I realized that these map operations are not actually valid candidates for parallelization like I originally thought because of the way that search parameters are shared across function calls via state. The successors need to be updated is sequential order. Otherwise, if the operations are performed in parallel, updates to the state contents (the search parameters) by one thread can be overwritten by another thread depending on when the threads access and save state. If having multiple threads was absolutely necessary, the state could be shared across the threads using something like an MVar, which essentially forces sequential behavior between threads anyway using a locking mechanism. Thus, attempting to parallelize these calls was pointless.

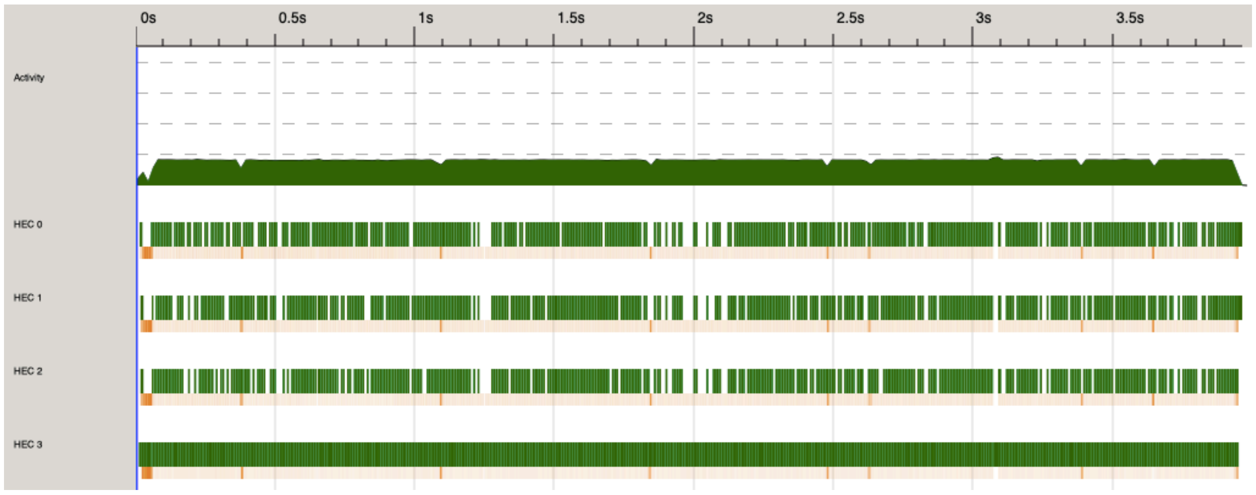
B. Calls to *map*

These calls appear in three locations (the *driver*, *computeShortestPath*, and *updateVertex* functions), and are used to calculate a list of costs for moving to each of a state's successors. This is done so that the least expensive successor state can be selected, which the algorithm moves the robot to.

Originally, I figured that the pure map operations could be parallelized using the *parList* strategy such that a thread could calculate each of the successor costs simultaneously. The issue with the *mapM_* calls is not present this time around because computing the cost of a state does not require modifying the search parameters. When all of the map calls were evaluated using the *parList* strategy, this was the outcome of an example run on four cores (I tried combining *parList* with a number of different strategies, but the results were the same each time):

```
SPARKS: 638405(2016 converted, 0 overflowed, 0 dud, 630223 GC'd, 6166 fizzled)

INIT    time    0.001s  (  0.006s elapsed)
MUT     time    3.602s  (  3.396s elapsed)
GC      time    1.497s  (  0.487s elapsed)
EXIT    time    0.000s  (  0.006s elapsed)
Total   time    5.100s  (  3.895s elapsed)
```

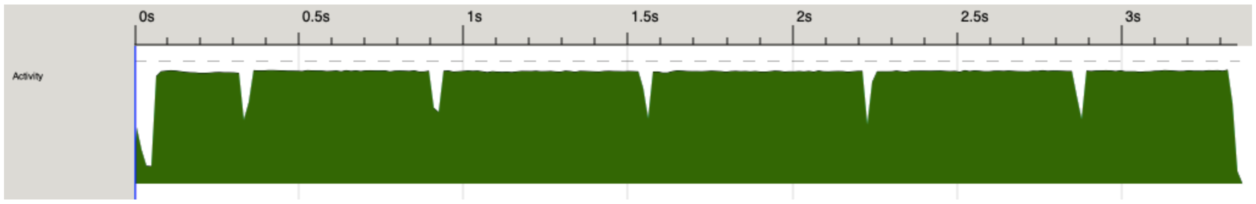


The ThreadScope event log showed decent work balancing across the four cores with occasional garbage collection, and some sparks were even converted into useful work. However, despite this, the vast majority of sparks (around 98%) were garbage collected, and sequential execution of the program was still faster (for reference, the same search problem on a single core ran in 3.569 seconds). In an attempt to address this issue, I reasoned that far too many sparks were being created, and that having fewer sparks would minimize garbage collection time. My idea was to have only the top-level *driver* function create sparks (such that only four sparks are created on each iteration of the algorithm). While this worked in reducing the number of sparks created, the results were similar to before, with the majority of sparks being garbage collected:

```
SPARKS: 1394(4 converted, 0 overflowed, 0 dud, 1382 GC'd, 8 fizzled)

INIT    time    0.002s  (  0.016s elapsed)
MUT     time    3.467s  (  3.363s elapsed)
GC      time    1.301s  (  0.476s elapsed)
EXIT    time    0.000s  (  0.003s elapsed)
Total   time    4.770s  (  3.857s elapsed)
```

I believe that the cause of this problem is two-fold. On one hand, the threads likely do not have enough time to perform their computations because the main thread requires the results immediately in order to retrieve the minimum cost state. Thus, the main thread may perform the computations before the sparks can be even be assigned to a thread, which is why many end up getting garbage collected. Secondly, the problem may also be an issue with granularity of the work being assigned to threads. Because the operation of calculating a state's cost can be done very quickly, sparking a thread to do it in parallel is overkill in a sense, and takes away from the ability of the main thread to stay busy. If we look at the activity section of the Threadscope output for one core (sequential execution), we can see that the activity level is much higher than when using four cores:

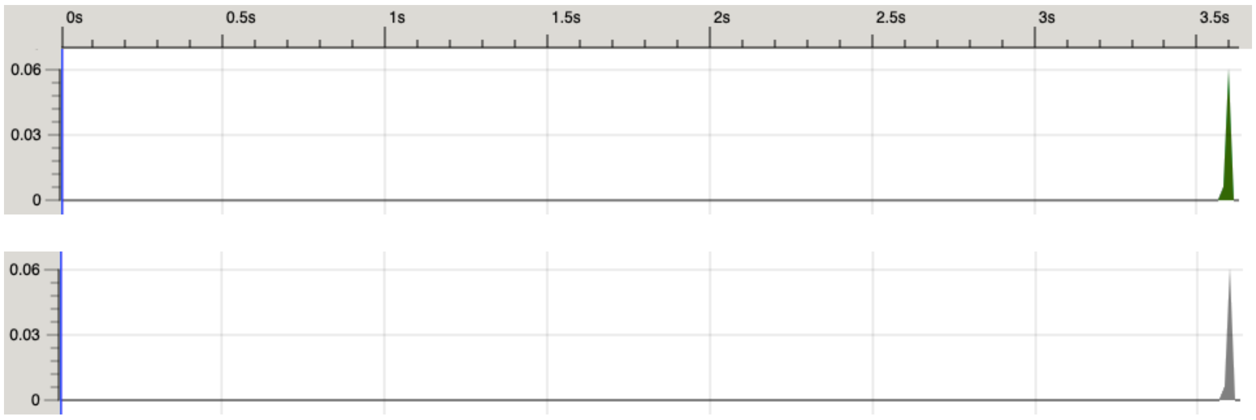


This told me that parallelizing any of the map operations in my program was excessive and would likely not result in the speedup that I desired.

2. Parallelizing the action of computing the current proposed path:

The priority queue is updated in D* Lite each time a previously unknown obstacle is encountered, implicitly creating the next path proposed by the algorithm. However, under normal execution of the algorithm, paths are never fully revealed, as the solution can simply be retrieved by following the cheapest transitions from the start to goal state using the search parameters. Unlike the mapping operations discussed previously, the process of uncovering this path is a perfect candidate for parallelization because it is a “fire-and-forget” activity that has no impact on the execution of the main driving function. Thus, the *computeShortestPath* function can be sparked in parallel once the queue is done updating after having encountering an object.

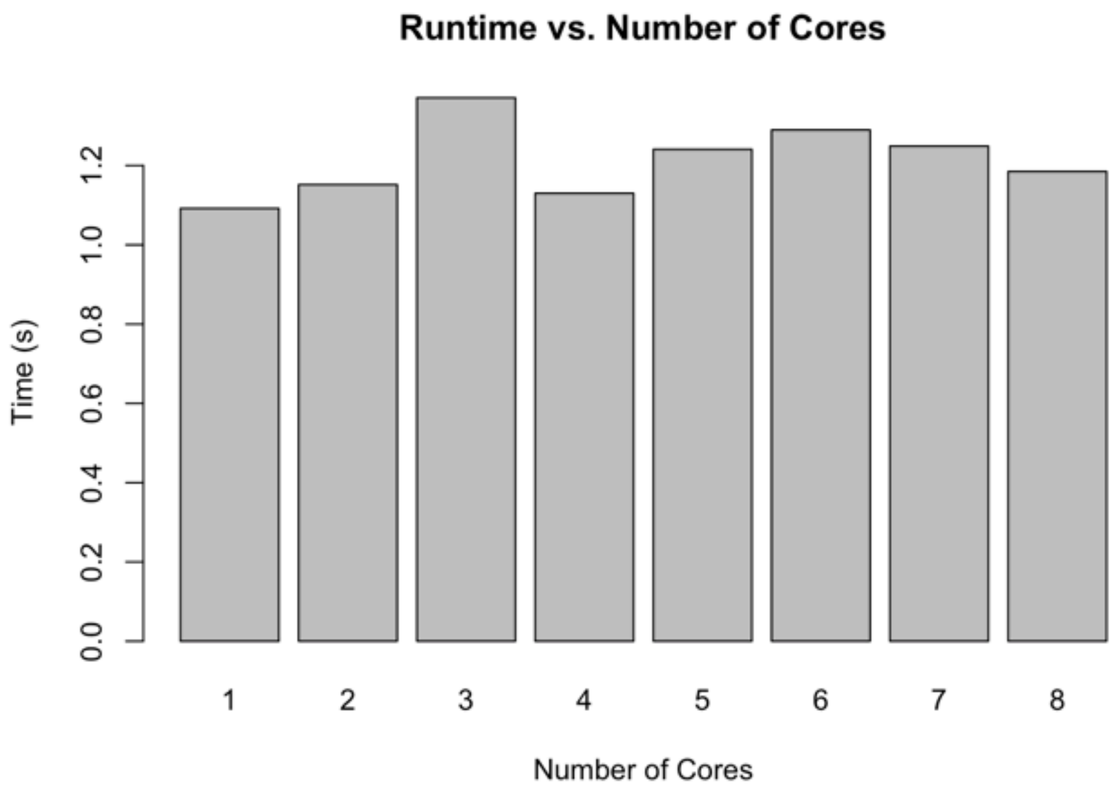
To implement this idea, I tried using a number of different parallel strategies, including *rpar*, *rseq*, *rdeepseq*, and *rparWith* in combination with *rseq* and *rdeepseq*. In each case, I was unable to produce the desired parallel behavior. The *computeShortestPath* function is fired off in parallel for each iteration of the algorithm in which a new obstacle in encountered. Thus, sparks should be created throughout the lifetime of the program. During trial runs, it is clear that my program creates the correct number of sparks (one for each obstacle). However, ThreadScope shows that these sparks are not created until the very end of the program’s execution, where they immediately fizzle out (presumably because the main thread needs to print the paths at that point and so it does the work itself):



Normally, my program checks that the path returned from *computeShortestPath* has not already been proposed by the algorithm so that the list of paths printed to the user does not contain repeats (the new path will be the same as the previous path whenever the discovered obstacle does not block the previous path). This means that the main thread requires the path to be completely evaluated immediately, and I speculated that this could have been the reason for why the sparks were fizzling out. However, removing the check for duplicate paths did not alleviate the issue. While more sparks are converted into useful work than before, they are still not created until the end of the program's execution.

I am unsure why my program ignores the calls for parallelism when sparks are created. It seems to be an issue with Haskell's laziness preventing eager evaluation of the paths. Because no parallel work is ever actually performed, the runtime of my program is once again slower using multiple cores than when ran sequentially, likely due to the overhead of establishing multiple threads of execution and the additional time that is wasted when all of the sparks are pointlessly created at the end of the program's execution.

Finally, the following graph is provided to give an idea of how much slower my program runs on multiple cores (when compiled with optimizations). Note that the test case listed in the README was used for each execution:


Runtime vs. Number of Cores
Time (s)
Number of Cores