

Final Report

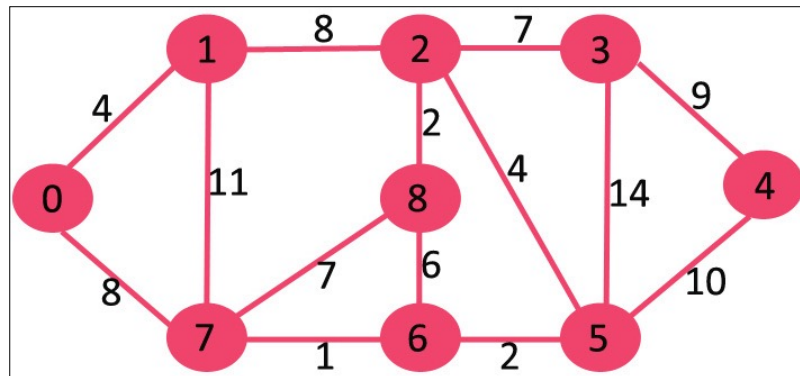
Prepared for: Sonia Martinez, Professor, UCSD

Prepared by: Brahm Prakash Mishra

Preparer PID: A53214048

9 December 2016

MOTION PLANNING OF A ROBOT USING DIJKSTRA'S ALGORITHM - IMPLEMENTATION, SIMULATION AND ANALYSIS



Introduction

A wide range of motion planning problems reduce to finding the shortest path from a point in a graph to another. It was Dijkstra's algorithm (invented by computer scientist Edsger W. Dijkstra) laid the foundation for many of the algorithms in use today. This project was aimed at exploring a serial and parallel implementation of this algorithm. With Dijkstra's algorithm, one can find the path from starting node (source) to destination node (goal), while simultaneously determining shortest path to all other nodes.

The input for the algorithm is a graph - a mathematical object defined as a set of vertices V , considered together with a set of edges E . Each edge connects a pair of vertices. Additionally, a weight can be assigned to each edge that physically represents the variation of a quantity over the graph. Often, this weight is a representation of the 'cost' of traversing the path. The problem then reduces to minimizing the cost, by determining the shortest path possible. The project was implemented using MATLAB, and led to a surprising result - parallel implementation is 'not' superior to serial implementation. For parallel implementation to actually be beneficial, one has to implement the algorithm using data structures that aren't easily accessible over MATLAB. Therefore, the only parallel implementations of this algorithm in use today are often written in medium level languages such as C and C++, where the programmer has complete control over data structures, pointers, and flow of the program. Instead of spending time optimizing the parallel implementation to surpass the serial one, the project shifts its focus to applications of Dijkstra's algorithm in Motion Planning and simulations.

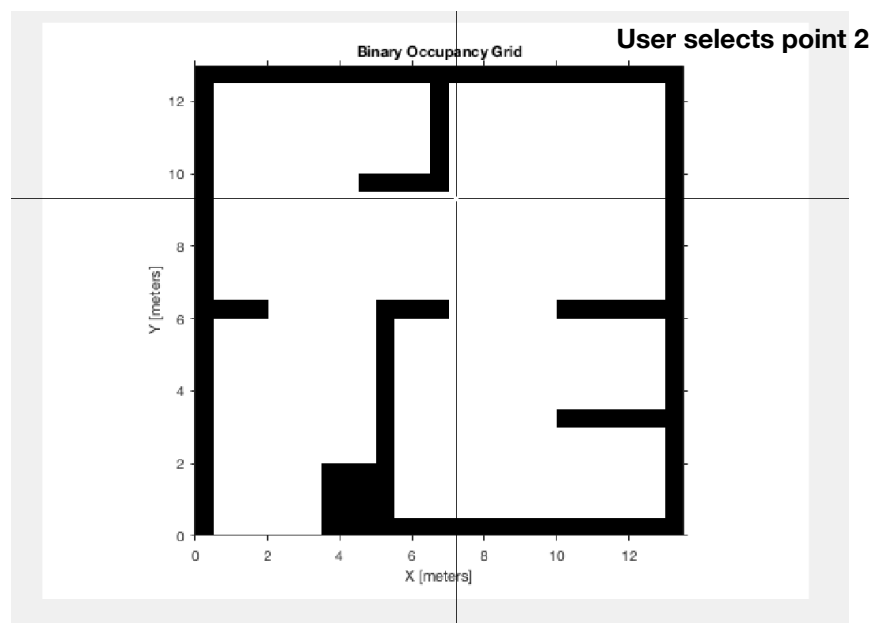
Objectives/Problem Statement

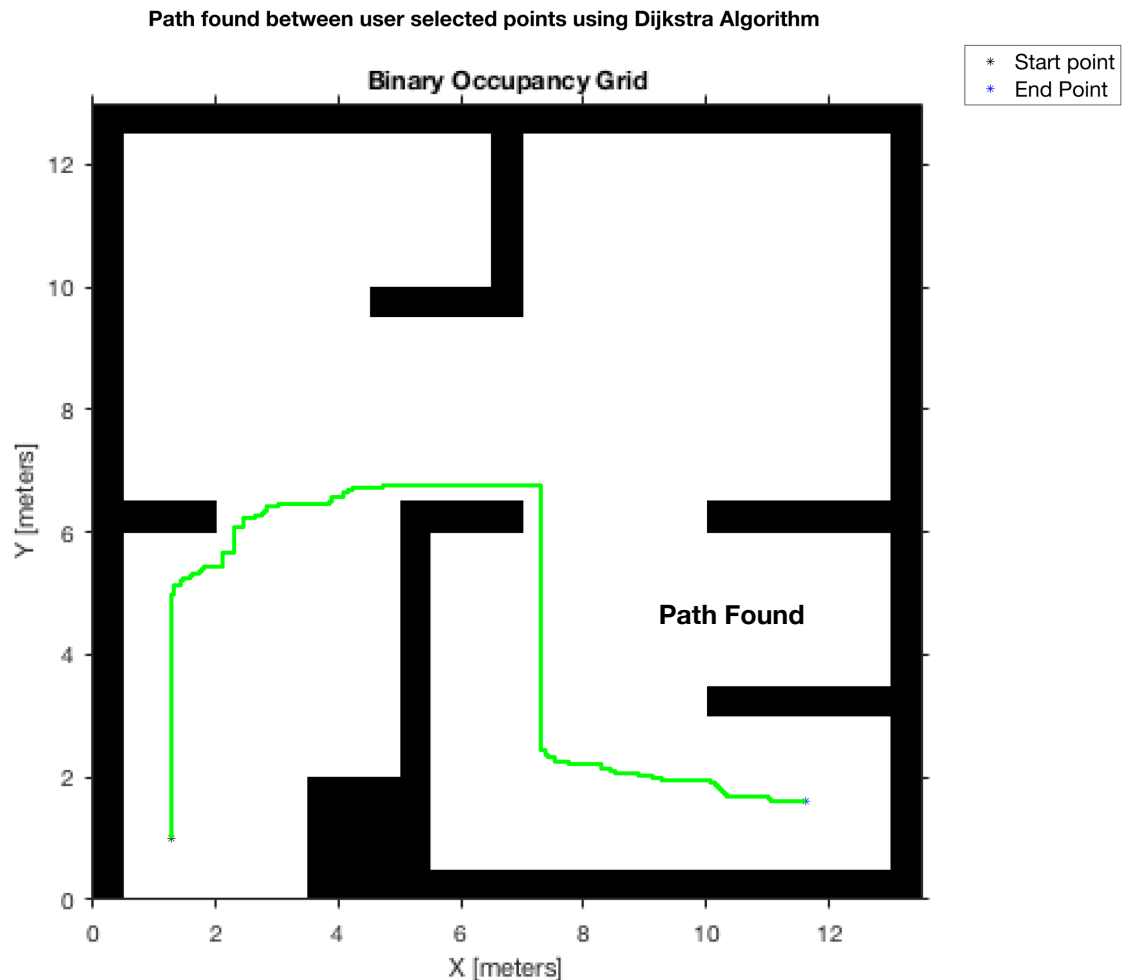
The project started out with the following basic objectives-

- To implement the serial and parallel Dijkstra's algorithm.
- To analyze the runtimes of above implementations.
- To determine actual improvement of parallel Dijkstra's algorithm over serial Dijkstra's algorithm by simulating motion planning in 2D environments using a bitmap image

Dijkstra in Action

We apply the algorithm to a 2D black and white picture of a room. The user gets to select 2 points within the room, and the algorithm determines the shortest path possible, if it exists. This was programmed on MATLAB, by reading the image of a room, and reading it into an occupancy grid based on a user supplied threshold

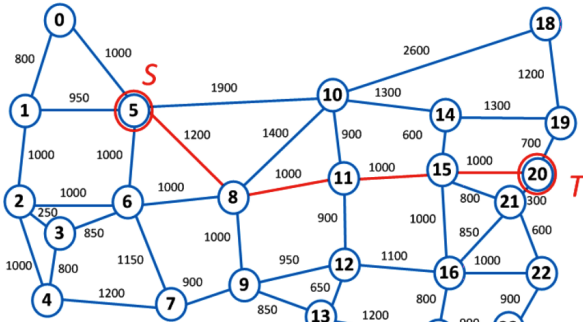




A 2D map of the room can be obtained from devices such as Kinect, using the Robot operating system. It has already been implemented on a number of famous robots. If the problem involves just going from point A to point B, then Dijkstra's isn't very useful (since it has a high time complexity). However, depending upon the robot's purpose, a slight modification of Dijkstra's can be applied to great effect. Even google uses an algorithm based on Dijkstra for its maps. Google Maps' ability to tell users quickly the shortest path between any two points on the map is very well suited for an algorithm like Dijkstra's, since it determines shortest path from a given source (which, in case of google maps, is the user) to all other points. Google's AI keeps track of shortest paths to nearby coffee shops, airports, banks, busses etc - all of which is simultaneously determined by using a modification of Dijkstra's algorithm. That is why, even so many years after its invention, and after many other schemes have been effectively used (such as A*) - it still makes sense to consider the base upon which all of that was built.

Before diving into the details of the project, the following section is aimed at establishing a theoretical context for the analysis that follows.

Theoretical Framework



A 24 node network, showing vertices, edges, edge weights, and shortest path as determined by Dijkstra's Algorithm

We start with the mathematical definition of a graph

$G = (V, E)$, where V is the set of vertices

E is the set of edges.

w is weight associated with each edge.

The shortest path from a given **source** vertex to a given **distance** vertex needs to be determined using Dijkstra's algorithm. The shortest path is one which carries the least weight from **source** to **goal**. In the process, shortest paths to all other vertices are also determined.

The pseudo code for a conventional implementation of the algorithm is as below

1. Initialize the lists as -

Parent P \leftarrow Invalid	//Since no vertex has a parent yet
Distance D \leftarrow Infinity	//Stating that each vertex is at infinite
	//distance from the previous one (i.e,
	//we haven't traversed the graph yet)

2. Empty the set of shortest path **S**

3. While (vertices present in VS) //Continue execution until we have
//traversed all vertices

- (i). Sort the vertices in VS according to their known sum of weights
- (ii). Pick out the vertex from VS that has the least cost-to-go, store the closest one in S
- (iii). If S is connected to a vertex in VS, and has a lower cost-to-go, update the cost-to-go

Theoretical Time Complexity of Dijkstra's Algorithm

From the pseudocode, it can be seen that the algorithm involves

1. Sorting of a priority queue - if implemented to the limit of theoretical efficiency, each update to edge weight takes $O(\log(V)) + O(1) = O(\log V)$ time. However, this can only be achieved if the priority queue being maintained is implemented as a heap.
2. Therefore, for a graph having E adjacent edges, the above update has to run E times, therefore total time complexity associated with maintenance of queue is $E \cdot \log V$
3. There are V such vertices, over which the graph will iteratively traverse. Therefore, the total time complexity is $V \cdot E \cdot \log V$ (alternatively phrased as $E \cdot \log V$ where E = total number of edges of graph)
4. Insertion and deletion from a priority queue are $O(1) \cdot n$ operations once the queue is sorted.

From the above analysis, we can expect $O(n^2)$ runtime from the algorithm. This isn't the theoretical minimum, since implementing Fibonacci heaps reduce it to $O(V + \log E)$ (or equivalently, $O(E + \log V)$). However, fibonacci heaps are infeasible, since the gains are marginal with a much higher difficulty in implementation. MATLAB also presents some additional difficulty in this regard, since the default data structures it presents are optimized for matrix operations.

Completeness of Dijkstra's Algorithm

Bully and Smith provide two concise theorems for proving completeness of exhaustive search algorithms

- If a path P is the shortest path from start to goal, each segment of P from node k_n to node k_{n+1} must be itself made up of shortest paths

- Once an element is added to shortest path, no other path in the graph can change its cost-to-go from source.

Suppose that v is the first vertex added to P for which **shortestDistance(source,v) != edgeCost[source,v]**. From there it follows that v cannot be **source**, because **shortestDistance(source)=0**. There must be a path from **source** to v . If there were not, **shortestDistance(v)** would be **inf**. Since there is a path, there must be a shortest path.

Implementation of Dijkstra's Algorithm

Serial Case

Since the project aims at comparing serial and parallel case in a fair manner, a modified version of the above scheme was used. It can be seen that the lists mentioned in the theoretical formulation are essentially tied to the node number. Therefore, all the four lists were merged and one matrix 'path' was used to store all of the necessary info. The rows of the matrix, from top to bottom, denoted the vertex number (node which the vertex forms), distance from start, its parent, and whether the vertex has been visited. Therefore, time complexity lost by not maintaining the priority queue was partially recovered by not having to insert individually into 4 separate queues, and reorder them at each iteration. The reason this

The pseudo code for serial implementation of the modified Dijkstra's scheme is as below -

end

A sample run on an undirected, 10 node graph

Subsequently, the code was profiled to observe its runtime. It was observed that serial implementation took around 0.017 seconds for computation (see graph subref, circled above). Although the overall program execution took well over a second, much of that time was spent in displaying graphs, etc so we do not take those into account.

To observe the effect of number of vertices and edges on the runtime, a second experiment was performed with $n = 100$, and number of edges increasing proportionately. We do not repeat this experiment by increasing edges, since the runtime is proportional to both E and V (*it is either $\log V^2$ or $\log E^2$, depending upon implementation*).















Profile Summary

Generated 09-Dec-2016 13:14:03 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
main	1	1.128 s	0.005 s	
Dijkstra	1	1.115 s	0.019 s	
graph.plot	1	0.485 s	0.014 s	
graph.subsref	35	0.330 s	0.017 s	←
GraphPlot.GraphPlot>GraphPlot.GraphPlot	1	0.309 s	0.022 s	
graph.graph>graph.get.Edges	28	0.284 s	0.006 s	←
tabular.horzcat	28	0.177 s	0.029 s	←
graph.graph>graph.graph	2	0.176 s	0.014 s	
GraphPlot.layout	1	0.163 s	0.001 s	
GraphPlot.layout>layoutauto	1	0.161 s	0.003 s	
table.table>table.table	32	0.159 s	0.025 s	
GraphPlot.layoutforce	1	0.156 s	0.006 s	
makeUniqueStrings	113	0.130 s	0.026 s	
MLGraph.forceLayout	1	0.111 s	0.005 s	
relaxationRoutine	7	0.110 s	0.002 s	←
distance	7	0.108 s	0.003 s	
MLGraph.forceLayout>layoutOneConnComp	1	0.106 s	0.011 s	

Performance Summary

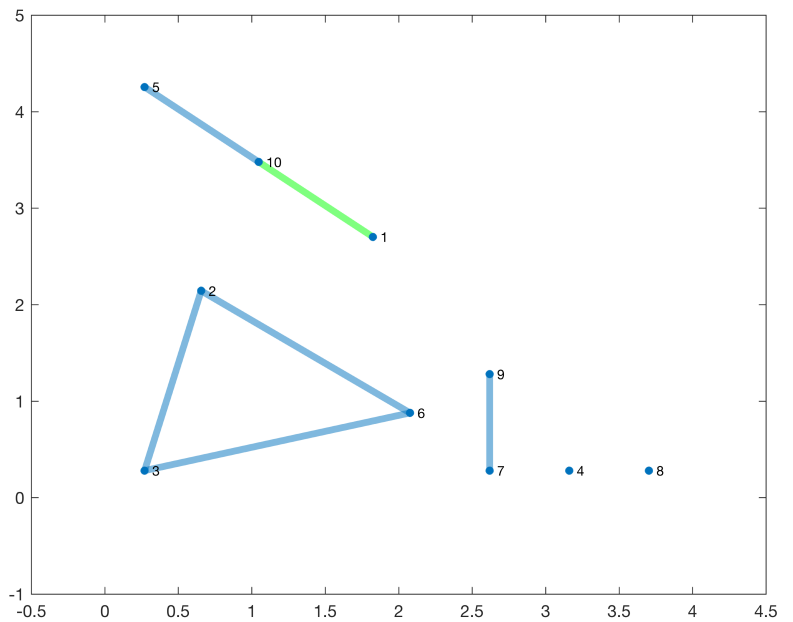
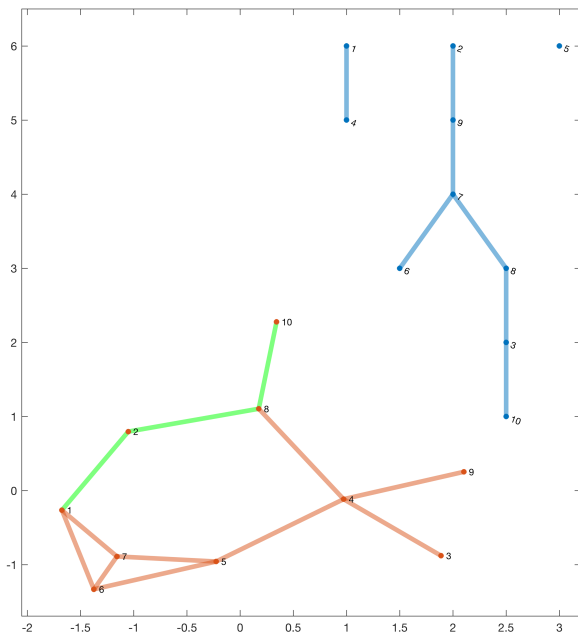
Generated 09-Dec-2016 13:20:53 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
main	1	6.275 s	0.003 s	
Dijkstra	1	6.264 s	0.046 s	
graph.subsref	1015	5.145 s	0.365 s	 ←
graph.graph>graph.get.Edges	726	4.185 s	0.116 s	
relaxationRoutine	289	4.105 s	0.012 s	 ←
distance	289	4.093 s	0.080 s	
tabular.horzcat	726	2.152 s	0.229 s	
makeUniqueStrings	2905	1.415 s	0.214 s	
table.table>table.table	730	1.374 s	0.249 s	 ←
makeUniqueStrings>makeUnique	2905	1.054 s	0.306 s	 ←
tableDimension>tableDimension.setLabels	726	1.015 s	0.019 s	
...leVarNamesDim.validateAndAssignLabels	726	0.996 s	0.075 s	
...gt:tableMetaDim.checkAgainstVarLabels	1453	0.762 s	0.049 s	
cell.ismember	2180	0.752 s	0.199 s	

The computations that significantly increase overhead have been marked in red. It can be noted that comparison, insertion, and deletion from the graph are expensive processes. On the contrary, relaxation (update of a matrix containing distances, shortest paths to all vertices, and parents of all vertices) grows by less than 6 times (marked in green). There are few key takeaways from this -

- When using high level languages like MATLAB, especially after its shift to just in time compilation, there are many behind the scenes overheads. Therefore, upon increasing the number of edges by 10 times, we had an increased overhead of over 6 times. This implementation of Dijkstra's is not very optimized.
- There are some parts of the algorithm, such as graph operations, that have the most contribution to overheads (the sections marked red went up by over 15 times). On the other hand, insertion and deletion from a Matrix is heavily optimized in Matlab, therefore the component marked green doesn't show the same amount of degradation (even though we are resizing it each iteration).
- The graph sub referencing (i.e, the part of the program that inserts, deletes and searches on the graph - the most expensive component of this algorithm) shot up to 0.365 seconds (over 21 times). Additionally, the time to reference tables

To conclude, the serial Dijkstra's was used on disconnected graphs. As before, the shortest paths from source to end have been marked in green.



Parallel Implementation

The parallel implementation was done with the below pseudocode

1. Initialize the lists as -

```

Parent P ← Invalid           //Since no vertex has a parent yet

Distance D ← Infinity        //Stating that each vertex is at infinite
                             //distance from the previous one (i.e,
                             //we haven't traversed the graph yet)

```

2. Empty the set of shortest path **S**

3. While (vertices present in VS) //Continue execution until we have

//traversed all vertices

//Each CPU handles V/P vertices now

```

for (v belongs to subgroup, but outside S)
  Calculate the distance from v vertex to s
  Select the vertex with the shortest path as the local
  closest vertex;
end                               //This takes O(V/P)

```

4. Search globally for closest vertex amongst local minimums - $\log(P)$

end

Running time is thus $O(V^2/P + V \cdot \log(O))$, //p = number of processors

Curiously, parallel implementation of this algorithm read to vastly **increased** overheads during MATLAB execution. The causes for this are the following -

- MATLAB parallel execution is faulty on quite a few systems (especially on Mac). The algorithm had reduced runtimes on a much slower Celeron computer - this points to software incompatibility
- Insertions into queues, amongst other things, are much faster when queue is implemented as a Binary heap. This requires advanced level of C++ integration with MATLAB and therefore was beyond the scope of this project.
- Although the algorithm is parallelized without much inter-processor communication, there was significant overhead with iterative recompilation of results - each processor gave its local minima, which stalled processing while calculating global minima. Therefore, unless the value of number of nodes is very large ($>10^3$) - the parallel implementation is, in many ways, inferior to the serial one that is straightforward.

The results from the parallel simulation are as below -

Profile Summary

Generated 09-Dec-2016 13:53:38 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
main	1	22.232 s	0.016 s	
parallelDijkstra	1	22.208 s	0.649 s	
Composite.subsref	595	11.110 s	0.155 s	
...Composite>Composite.getValOrError	793	10.384 s	0.037 s	
KeyHolder>KeyHolder.getFromLab	793	9.825 s	0.034 s	
...ceSet>RemoteResourceSet.getFromLab	793	9.764 s	0.827 s	
spmd_feval	199	9.585 s	0.018 s	
spmd_feval_impl	199	9.562 s	0.089 s	
...>RemoteResourceSet.remoteRetrieval	793	6.150 s	1.353 s	
...box.distcomp.pmode.SpmControllerImpl (Java method)	6163	6.039 s	6.039 s	
...oteSpmExecutor.isComputationComplete	399	5.872 s	0.357 s	

Therefore, the project concludes that parallelism is not always beneficial!

As mentioned in the introductory paragraph, Dijkstra's by itself has been surpassed by a great deal of algorithms. Most common of them being an A*. However, exploring the behavior of this algorithm has proven to be a good way of understanding algorithms in general.

Conclusion (A debrief - What would I have done if I had gotten the time to start over)

In retrospect, the idea of implementing parallel Dijkstra doesn't seem very alluring. Despite time and effort, the result wasn't appealing. Parallel programming by itself poses significant challenges, and unless one has a very clear idea about the behavior of algorithms, it is difficult to bring a project to fruition. It might have been better to explore other aspects, such as pseudorandom distribution based planning, Markov chains, or inverse kinematics of robotic arms.

However, if I had a chance to do this very project again, I would start with implementing a simple scheme in terms of algorithm and exploring how motion planning can be integrated with real-world robots. I was able to parse a simple 2D map in terms of a Matrix and apply Dijkstra's on it - but it would be more interesting to explore how point cloud representations, and probabilistic sensing can be leveraged to implement motion planning on robots in uncertain environments.

However, as with any endeavor - successful or otherwise, I was able to learn a great deal. I got to learn about parallel programming, got to explore internals of MATLAB and how and where it can be leveraged to maximum efficiency. Additionally, a number of attempts did not make it into projects cause I wasn't able to find time to explore them satisfactorily. For instance, I tried integrating the concept of Dijkstra's with ROS. The idea was to run a simulation in ROS, while feeding it real time camera messages via an Android phone. So, although the results weren't to my liking, it was still a fun experience!
