16-782 Homework 1 (Two late-days used)

Changsheng Shen (Bobby)
changshs@andrew.cmu.edu

09/29/2019

How to compile:

Navigate to the code folder, then:

```
mex planner.cpp dijkstra.cpp
```

Note that I used static variables, so the above command need to be executed every time. Namely, always re-compile before starting to plan.

Explanation:

I implemented a Dijkstra planner that expands the entire map. (See `dijkstra.h` and `dijkstra.cpp`)

It is abstracted as a C++ class, and should be able to easily applied to other grid-based 2D planning problems with minimum modification.

The idea is:

1. Expand the entire traversable of the map.

2. For each coordinate on the target trajectory, back-trace a path from it all the way to the start position of the robot.

3. For each path, evaluate the waiting time of the robot. For a given coordinate (i, j) on the target trajectory,

$$t_{wait}^{ij} = t_{target}^{ij} - path^{ij}.length$$

(i, j) where t_{target}^{ij} is the time when target will arrive at (i, j) (essentially, its index in the target trajectory).

4. Only if $t_{wait}^{ij} > 0$, the robot will be able to catch the target on time at (i, j). We denote these paths as valid paths.

5. For each valid path, we need to find a coordinate (i_{min}, j_{min}) with the smallest cost, comparing to all other coordinates on this path, to wait there. Only in this way we could minimized the accumulated cost of waiting. This coordinate (i_{min}, j_{min}) can be tracked during the back-tracing in step 2.

6. Interpolate t_{wait} points into the path, at (i_{min}, j_{min}) found in the previous step. Basically, just stop and wait at (i_{min}, j_{min}) for t_{wait} steps as soon as the robot reaches there.

7. Evaluate the cost of each valid path:

$$cost = cost_{path} + cost_{(i_{min}, j_{min})} * t_{wait}$$

where $cost_{path}$ is already computed during the initial expansion.

8. Store all the evaluated valid paths in a sorted `std::map`:, with key value as its actual cost.

9. Calculate how many time steps spent on the initial planning (step 1 - 8) denoted as t_{init}.

10. Check if the current least-cost path is valid with t_{init} steps delayed. (i.e. if its $t_{wait} >= t_{init}$).

11(a). If not, then keep searching forward in the sorted map to find the first path that is still valid, given the delay. Extract it as the best path to execute.

11(b). If yes, then extract this path as the best path to execute.

12. Trim the best path by removing t_{init} points at the waiting coordinate, (i.e. $t_{wait} = t_{wait} - t_{init}$). Intuitively, we spent too much time on the initial planning and the target already start moving. In order to catch it on time, we should not wait less on the waiting point.

13. Execute the trimmed path until catching the target.

Data Structures:

2D `std::vector`:

The original map, the updated cost map, the map to store parent coordinates, visited (open-list flag) and explored (closed-list flag). All the 2D vectors have dimension same as the original map.

`std::priority_queue`:

The open list. This happens to be significantly faster than using `std::multimap` when the number of elements becomes large (e.g. on map1 and map2). Given the nature of a queue structure, if we need to update the information (a lower cost, parent) of a cell that already exists in the open list, we do not search for it in the queue. Instead, we update the corresponding 2D vector, and then push a new element with the updated information into the open list. The out-dated element will be automatically ignored when it is popped out from the open list later, since the cost of it is higher.

`std::unordered_map`:

The trajectory of the target, where each coordinate on the target trajectory is mapped to its index (i.e. the timestamp when the target reaches it).

`std::map`:

All the valid candidate paths of the robot found by the planner, sorted based on the total path cost.

Self-defined `struct Cell { ... }` and `struct Path { ... }`:

Mainly for book-keeping, abstraction and readability improvement.

Efficiency tricks:

The planner is declared as a static class object, and it will initialize, expand the entire map and generate all candidate paths for the robot in the very beginning.

After that, the robot just pick the best candidate path and follows it, and the planner does not re-plan. The memory usage also won't increase as time increases.

Results:

```
Map1:
RESULT:
 target caught = 1
 time taken (s) = 2642
 moves made = 2638
 path cost = 2642

Map2:
RESULT:
 target caught = 1
 time taken (s) = 4673
 moves made = 1235
 path cost = 1603276

Map3:
RESULT:
 target caught = 1
 time taken (s) = 247
 moves made = 243
 path cost = 247

Map4:
RESULT:
 target caught = 1
 time taken (s) = 380
 moves made = 266
 path cost = 380
```

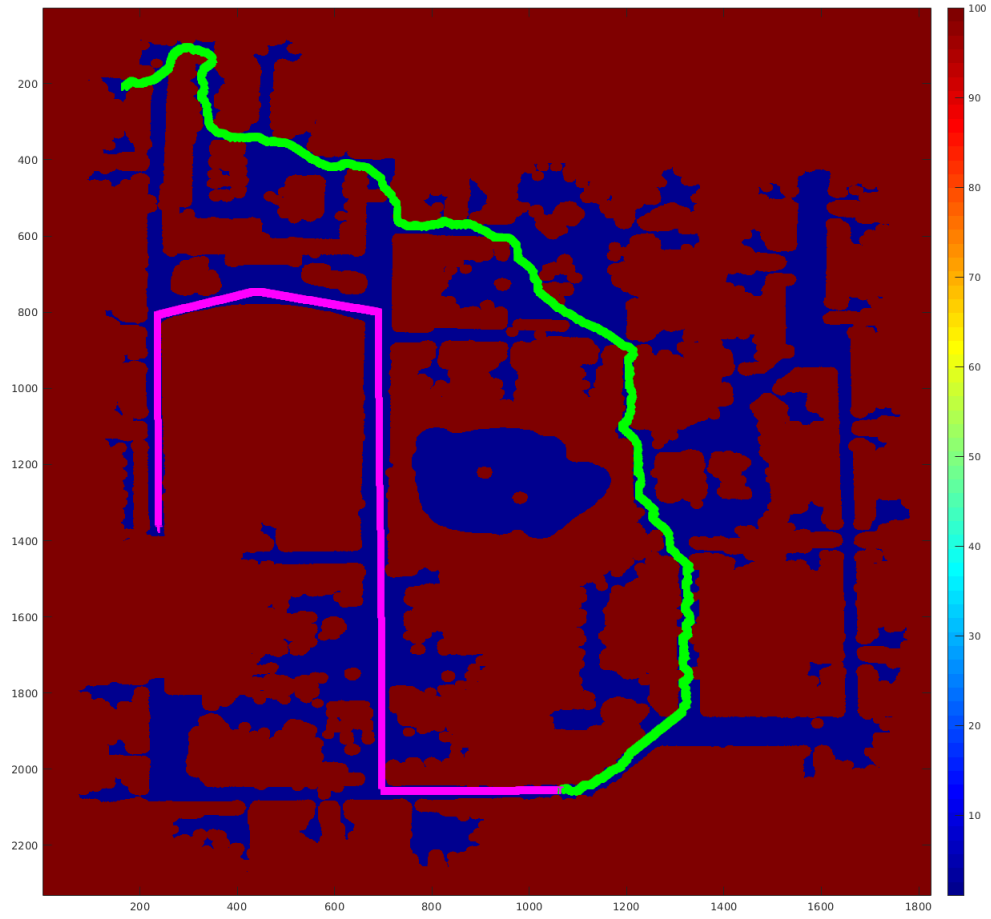

Figure 1: Result of Map 1

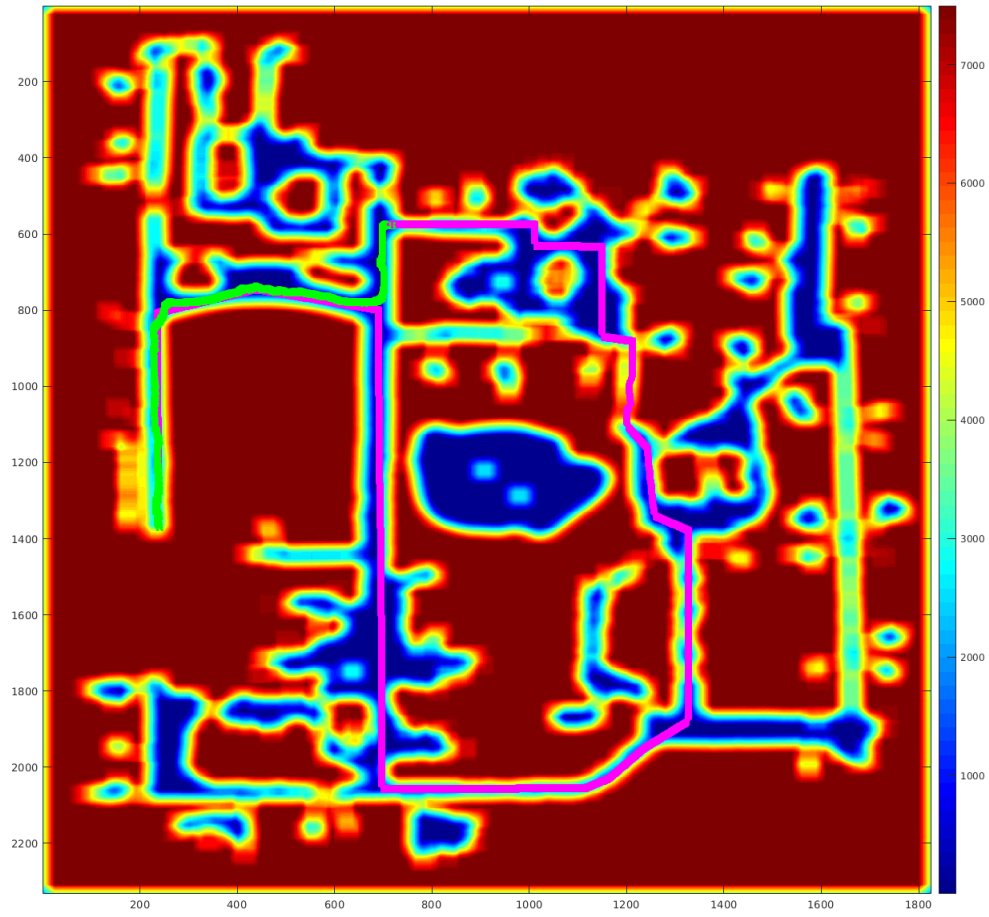

Figure 2: Result of Map 2

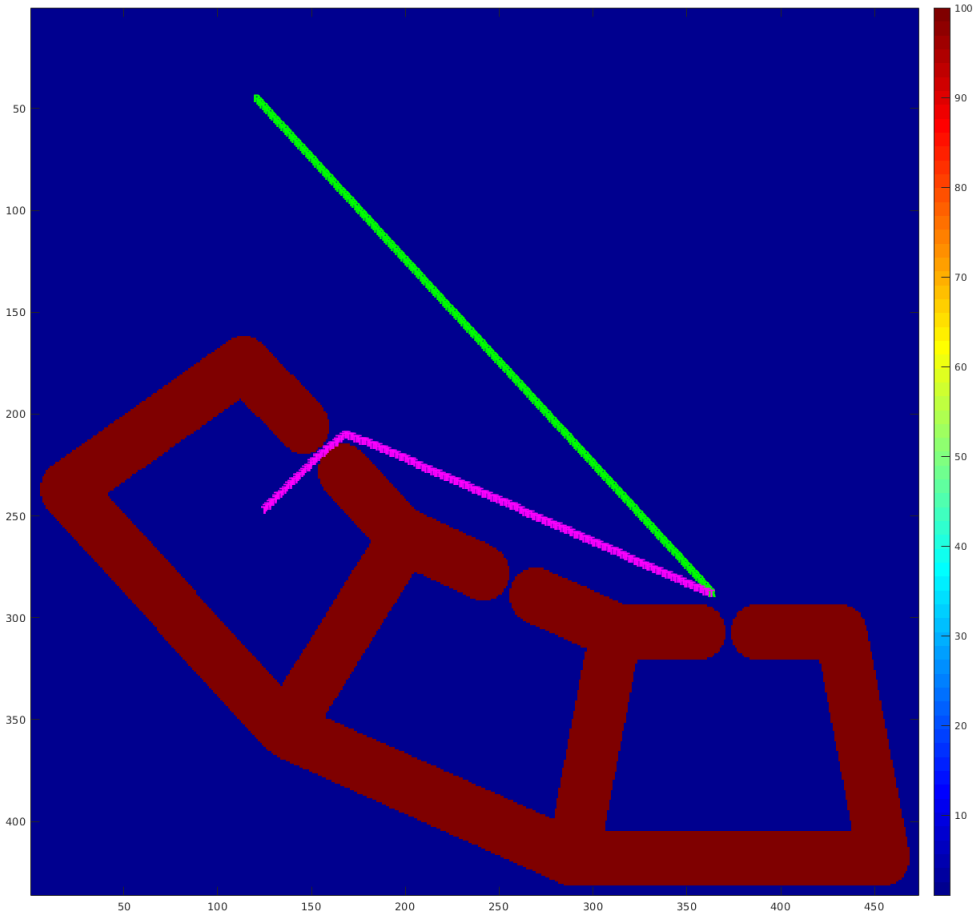

Figure 3: Result of Map 3

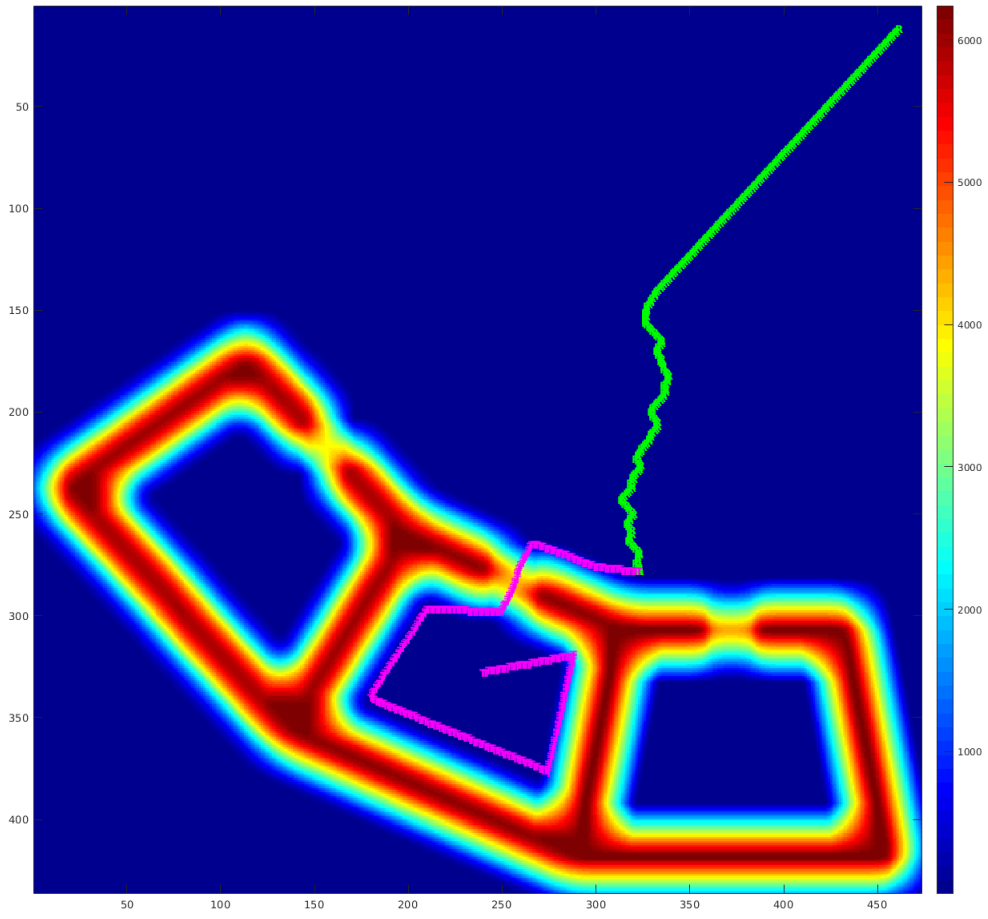

Figure 4: Result of Map 4