Grid-based Navigation for Mobile Robots

Tucker Balch

1 Mobile Robots, Navigate!

Navigation is a special task for mobile robots; after all, isn't getting somewhere what being mobile is all about? And getting somewhere in a complex environment means *navigation*. You may not have given much thought to how you manage to walk across a crowded parking lot, but programming a robot to do the same thing is challenging. Coordinating sensor and motor skills are not the least of the problems, but we'll leave those issues for others to solve for now. This article will examine path planning: how a robot can select a path to a goal.

There's a section on equipment your robot will need to navigate and a step-by-step explanation of how to implement an efficient cost-based algorithm. Tested C source code is included; it's also available by ftp (see the end of the article for ftp information).

2 What A Robot Needs to Navigate

There are a few capabilities a robot must have to navigate. Most importantly it must have some way of sensing where it is supposed to go. This information may be provided by an infrared beacon at the goal, it might be an (x, y) coordinate, or if the robot is equipped with a positional sensor like GPS, a point on the earth's surface defined by latitude and longitude. It is also important for the robot to know where it is in relation to the goal. I'll assume this information is available and that it has been converted into cartesian coordinates. The robot's location is given by `(robot_x, robot_y)` and the goal is `(goal_x, goal_y)`.

You might be wondering how accurate these values need to be. It depends. If your robot directly senses the goal as it moves about, homing on an IR beacon for instance, the values can be somewhat coarse. The accuracy of this type of sensory information does not degrade over time. On the other hand, if the robot depends on internal sensors for position, using timing or shaft encoders for instance, the data must be more precise. As the robot moves about its estimate of position will get worse and worse with this type of sensor. In the end you'll have to experiment with your particular robot to see if its position sensors are good enough.

Next, the robot must be able to detect obstacles. Sonar range sensors, infrared proximity sensors and laser ranging devices are all excellent, but a good old bump sensor works just fine.

Last is the issue of computing power and memory. The code listings here were implemented and tested on a Unix system with lots of memory. I realize many robots are running around with a 6811 and only 512 bytes of RAM. There's no reason the code shouldn't work on a 6811 (provided you have a compiler), but you'll probably need at least 10K of RAM. The code is in C; but translation to assembly, BASIC, or other languages should be straight forward. Just remember: grid-based path planners are traditionally compact in code, but fat in RAM.

Now on to path planning.

3 Representation for Navigation

Lots of approaches to robot path planning have been proposed and implemented. An important distinction between them is in how they *represent* the world. "Representation" refers to how the data is stored in a computer's memory and how that corresponds to objects in the outside world. In this article, we'll look at one type of grid-based representation. A good reference for information on other approaches is the book *Robot Motion Planning*, by Jean-Claude Latombe. He covers this approach, and others, in great detail.

With respect to navigation, the world consists of open areas, where a robot may travel freely and closed areas (obstacles) where the robot cannot travel. We'll represent these two types of space in a two-dimensional occupancy grid. Each cell in the grid corresponds to a small section of the real world. The occupancy grid is filled in so that a cell is marked "empty" if the corresponding part of the world is free space, and "full" if it contains an obstacle. Figure 1 shows how an example scene is represented in an occupancy grid. Occupancy grids are convenient for a robot to update when sensors indicate changes in the world. They are also easy to use for planning. The primary disadvantage is memory consumption; a high resolution map might require several megabytes.

The occupancy grid is usually read in from a file at start-up time. But such a map may not always be available, or even worse, it might be available but wrong. Luckily, occupancy grids are easy to update if the robot discovers a discrepancy. Suppose, for instance, that the robot discovers a new obstacle just to the north of itself. Since north is in the $+Y$ direction, the occupancy grid is corrected like this:

```
occupancy[robot_x][robot_y + 1] = FULL;
```

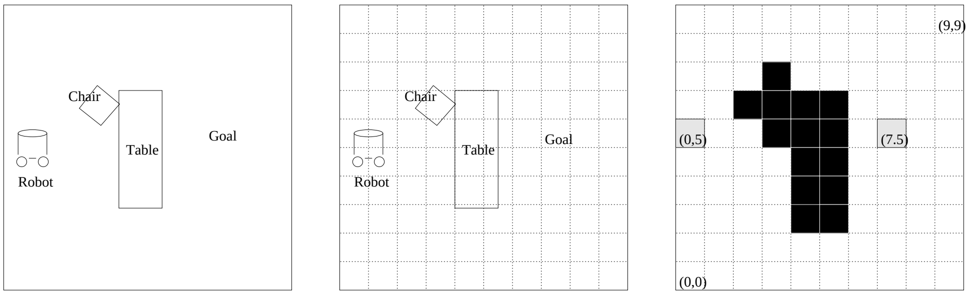

Figure 1: How an example scene (left) is represented in an occupancy grid (right). The black cells are "full" while the white ones are "empty". Locations of the goal and the robot are not usually stored in the occupancy grid, but here they are colored gray for visualization purposes.

Similarly, if space to the east was previously thought to be occupied but turns out to be free, the correction is made by:

```
occupancy[robot_x + 1][robot_y] = EMPTY;
```

Note that changes in the occupancy grid will require a replanning step since a new obstacle might obstruct the planned route, or newly discovered open space might offer a short cut.

One final issue regarding the occupancy grid is *resolution*. The grid's resolution refers to how large an area in the real world is represented by one cell in the grid. The example uses a 1 foot resolution on 10 by 10 foot grid (Figure 1). For most robot applications this resolution is too low. Low resolution may lead to a less optimal path and jerky robot motion. The selection of a resolution should depend on the accuracy of the robot's position sensors, how fast it will move and how much memory is available. In general, it is best to use as high a resolution as possible since this will result in the most accurate representation of obstacles and the "smoothest" plan. But there are reasons to avoid too high a resolution. It doesn't make sense, for instance, to use a higher resolution than the precision of the robot's position sensor. Also, if the robot moves so quickly that it skips over several cells between computation cycles, the resolution is probably too high. Finally, higher resolution maps will take longer to plan over and use more space. A good starting point is to set the resolution to the distance your robot will travel in one computation cycle.

4 Representation of the Plan

The plan is a cost grid. Each cell in the grid is an estimate of the shortest travel distance from that point to the goal. Usually the cost grid is the same

8.66	7.66	6.66	5.66	5.24	4.83	4.41	4.00	4.41	4.83
8.24	7.24	6.24	5.24	4.24	3.83	3.41	3.00	3.41	3.83
8.66	7.66	6.66	BIG	3.83	2.83	2.41	2.00	2.41	2.83
9.07	8.07	BIG	BIG	BIG	BIG	1.41	1.00	1.41	2.41
9.49	9.07	9.49	BIG	BIG	BIG	1.00	0.00	1.00	2.00
10.49	10.07	9.66	9.24	BIG	BIG	1.41	1.00	1.41	2.41
10.66	9.66	8.66	8.24	BIG	BIG	2.41	2.00	2.41	2.83
10.24	9.24	8.24	7.24	BIG	BIG	3.41	3.00	3.41	3.83
9.83	8.83	7.83	6.83	5.83	4.83	4.41	4.00	4.41	4.83
10.24	9.24	8.24	7.24	6.24	5.83	5.41	5.00	5.41	5.83

Figure 2: The cost grid for the example navigation problem.

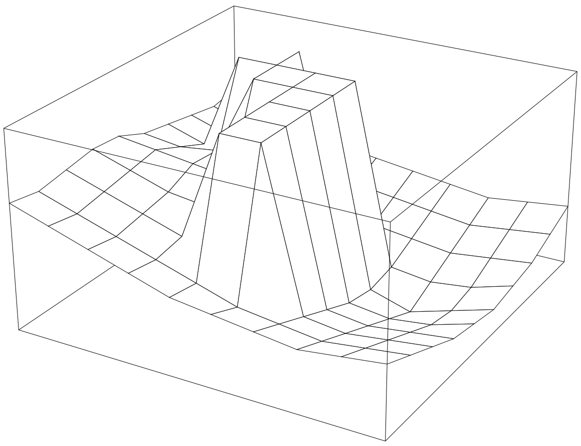

Figure 3: The cost grid viewed as a three dimensional surface.

resolution as the occupancy grid. This makes referencing one while using the other more convenient.

For the moment don't worry about how cost cells are filled in. You'll see how to do that in the next section. Look at Figure 2. This is a cost grid for the example in Figure 1. Note that the cost at the goal cell is 0.0, and the cost at other cells increases the further they are from the goal. To get a better idea of what the cost grid looks like we can view a three dimensional surface generated by plotting the cost at each point as a height. Figure 3 shows the plot for this cost grid. The high spots on the plot are obstacles. The goal is at the low point at the front right. You can see that following the plan is just like rolling a ball down hill.

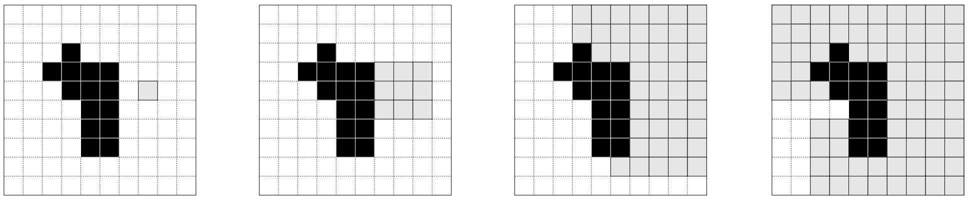

Figure 4: This sequence shows how cost computation expands outwardly from the goal. Initially (left), only the cost at the goal is known. The other images, from left to right, show the cells that have been evaluated after 1, 4 and 7 iterations.

5 How the Cost Grid is Computed

As you may realize by now, the hard part is computing the cost grid. Recall that the number in each cell is an estimate of the cost of traveling from that cell to the goal.

First the program sets all the cells in the grid to initial values. Goal and obstacle cells have constant values throughout the computation. Since the cost of traveling from the goal to the goal is 0 the goal cell is always set to 0.0. Since traveling through obstacles is not normally desired (!) we discourage this by setting the cost at obstacle cells to an arbitrarily large value called `BIG_COST`. "Empty" cells are also initially set to `BIG_COST`, but lower values for them will be computed later.

After setting the cells to initial values, the program repeatedly scans through the grid looking for cells it can reset to lower values. The lower cost is computed by looking at neighboring cells and using an estimate of travel cost from the adjacent cells to the current cell. For laterally adjacent cells the estimate is 1.0, and for diagonally adjacent cells it's $\sqrt{2}$. This is just the distance from the center of one cell to the center of the next, assuming each cell measures one unit on a side. Note that if the cost estimates were multiplied by the resolution of the grid the values at each cell would reflect the true distance to the goal. At each sweep through the grid one more layer of cells is re-evaluated until, eventually, the entire grid is minimized. Figure 4 shows how the evaluation spreads outwardly from the goal.

At heart of the planner is the procedure `cell_cost()` (listing at end of article). `cell_cost()` evaluates the cost at one cell by inspecting the cost of each cell adjacent to it. For the neighboring cells, it adds appropriate lateral or diagonal travel costs and notes the lowest value. That lowest value is recorded in the current cell. `cell_cost()` returns a 0 if there was no change in the cost, 1 otherwise.

We now have all the pieces needed to build a cost-based grid path planner. Here it is:

```
int count = 1;
while (count != 0)
        {
        count = 0;
        for (i = 1; i < GRID_SIZE-1; i++)
                {
                for (j = 1; j < GRID_SIZE-1; j++)
                        {
                        count = count + cell_cost(i,j);
                        }
                }
        }
```

Yes, that *really* is the entire planner! It repeatedly cycles through the grid to recompute each cell's cost until it makes a full pass with no changes. Now for the bad news: as it stands, this planner is *extremely* inefficient. If the grid has N cells along each side, and we plan over a complicated map, it might cycle through the grid N^2 times and make N^4 cell evaluations. This will use up a lot of CPU cycles for a large grid. In a later section we'll see how to make it faster.

6 Using the Cost Grid as a Plan

Let's look at how to employ the cost grid as a plan. Provided the robot knows its own location, using the plan is as simple as

```
new_direction = check_plan(robot_x, robot_y);
```

The function `check_plan()` knows how to consult the cost grid and return a heading for the robot. It looks at the region in the cost grid corresponding to where the robot is in the world. Then it choses the direction along the shortest path to the goal (i.e. "down hill" on the cost grid).

To do this, `check_plan()` looks at a sample of nearby cells to compute the *gradient* at the position of the robot. The gradient is down hill on the cost grid. X and Y components of the gradient are computed separately, then the `atan2()` function is used to convert them into a direction between 0 and 2π. If one of the nearby cells contains an obstacle, `check_plan()` uses the direction towards the lowest cost neighbor instead of using the gradient. This avoids jittering when the robot is near obstacles.

7 Pulling it All Together

A robot using a grid-based planner should follow a cycle of planning and acting something like this:

1. Initialize variables and read the map.
2. Plan.
3. Check sensors to find robot position, goal and nearby obstacles.
4. Update map if sensors show a discrepancy.
5. If the map has changed, recompute the plan.
6. Check plan and initiate movement along shortest path.
7. Go to 3.

This sequence is used in the C code below. But this program is just a simulation until you replace the sensing and movement "stub" procedures with appropriate subroutines for your hardware.

The program includes a procedure called `readmap()` that will read a map of obstacles into the occupancy grid and initialize the robot and goal locations. If you want to use this capability, you can make a map of the environment in a text file using spaces for open space, the letter 'O' for obstacles, the letter 'G' for the goal, and 'R' for the robot. The example from Figure 1 can be coded in a text file as:

```
␣␣␣␣␣␣␣␣␣␣<CR>
␣␣␣␣␣␣␣␣␣␣<CR>
␣␣␣O␣␣␣␣␣␣<CR>
␣␣OOOO␣␣␣␣<CR>
R␣␣OOO␣G␣␣<CR>
␣␣␣␣OO␣␣␣␣<CR>
␣␣␣␣OO␣␣␣␣<CR>
␣␣␣␣OO␣␣␣␣<CR>
␣␣␣␣␣␣␣␣␣␣<CR>
␣␣␣␣␣␣␣␣␣␣<CR>
```

The $\sqcup$ symbol indicates a space, and $< CR >$ indicates the end of a line. If you run the program using this input file, it will simulate moving the robot from its initial location $(1, 6)$ to the goal, $(8, 6)$. The resulting path is shown in Figure 5. If the computer on your robot cannot read files, you'll obviously have to avoid using `readmap()`.

Also, if you'd like, you can print out the cost grid using the function `printcost()`

8 Making it Faster

Earlier, I pointed out that the present routine for computing the cost grid is inefficient. For the example problem 1000 cell evaluations are made. This means

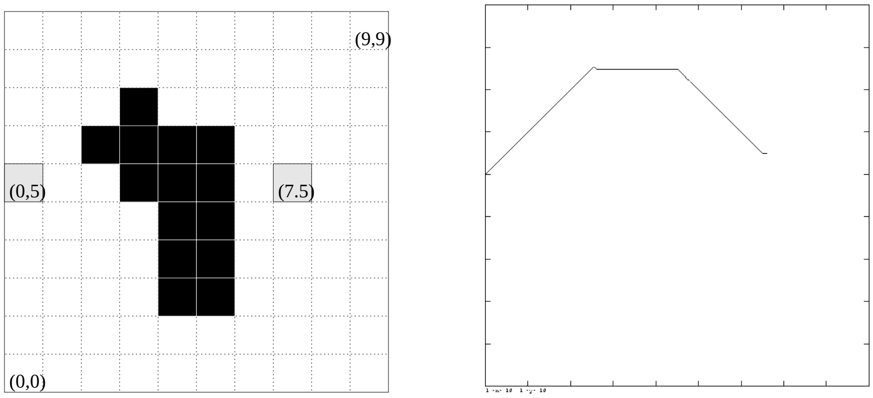

Figure 5: A simulated robot navigates across the scene on the left. The resulting path is shown on the right.

each cell in the 10 by 10 grid is evaluated 10 times. It should really only be necessary to evaluate each cell once. But if efficiency is not important, you might want to stick with the slower version since it is less complicated.

The main problem with the old planner is that cells are not evaluated in an efficient order. Let's look a better way to order the evaluations. Consider the first time a cell is evaluated. Recall that all the cells are initially set to `BIG_COST`. If all the cells adjacent to the current cell contain `BIG_COST` as well, there is no way the cell can be lowered. A cell should not be evaluated until after one of its neighbors has been lowered, otherwise we're wasting time. Initially, the only cells with a low cost neighbor are the cells next to the goal, so they should be evaluated first. Also, if we can be sure the first evaluation is correct, there is no need to evaluate a cell again.

One way to do this is to arrange for cells to trigger the evaluation of neighboring cells after their cost values have been lowered. To do this we keep a list, called the *open list*, of cells whose cost has been lowered. It is automatically sorted from lowest to highest using linked list routines. Cells are "popped" off the top of the list by a routine called `expand()`. After `expand()` pops a cell off the list, it looks at each of the cell's neighbors to determine if they have been evaluated. Each cell that has not been evaluated is evaluated at that point, then pushed onto the open list for later expansion. This ensures that the lowest cost cells are expanded first.

The new planner works by first pushing the goal onto the open list. Next it repeatedly pops cells off the list and expands them until the open list is empty. The computation eventually terminates since each cell is expanded only once and cells outside the boundaries are not pushed onto the list.

This planner makes only 85 cell evaluations on the example problem, so it runs about 10 times as fast as the old planner for that case. In most situations the faster planner will run N times faster where N is the number of cells along one side of the grid. Due to space considerations, listings for fast version are not not included below, but they are available by e-mail or ftp.

9 Wrapping Up

There are ways to make an even faster grid-based planner. The A* (pronounced "A star" with a long "a") algorithm works by only expanding nodes along a direct path between the goal and the robot. D* (pronounced "dee star"), developed by Anthony Stentz at CMU, initially computes the grid as outlined here. But D* keeps additional information so that when errors are found in the map the plan can be corrected without recomputing the entire grid. For dynamic or unknown situations D* is better since it does not require a complete replanning step when errors are found.

If you'd rather not type in the program by hand, you can find it via anonymous ftp at `ftp.cc.gatech.edu` in the directory `people/tucker/gridnav1.0` Another directory, `gridnav2.0` contains the fast planner. Get all the files from one directory or the other. If you are familiar with `tar` and `uncompress` you can grab `gridnav.tar.Z` to get them all at once. If you don't have access to ftp, send me e-mail indicating which version you want and I'll e-mail the files back to you (`tucker@cc.gatech.edu`). After you have the files on your local system, a simple `make gridnav` should compile it for you.

10 About the Author

Tucker Balch was born in Miami, Florida in 1962. He received the B.S. Degree from Georgia Tech in 1984 and the M.S. Degree from U.C. Davis in 1988. He is currently pursuing a Ph.D. in Autonomous Robotics at Georgia Tech. From 1984 to 1988 he supported research at the Lawrence Livermore National Laboratory as a computer scientist. He entered the Air Force as a Pilot Candidate in 1988 and completed fighter training in 1991. He now flies F-15 Eagles at the Georgia Air National Guard.

His research interests include integration of deliberative planning and reactive control, communication in multi-robot societies, and parallel algorithms for robot navigation.

Tucker can be reached by e-mail at `tucker@cc.gatech.edu` . His world-wide-web page is `http://www.cc.gatech.edu:/grads/b/Tucker.Balch` .

References

[1] Latombe, Jean-Claude, *Robot Motion Planning*, Kluwer Academic Publishers, Boston, 1991.

[2] Stentz, A., "Optimal and efficient path planning for partially-known environments", *Proceedings 1994 IEEE International Conference on Robotics and Automation*, p.3310-17 vol.4, 1994.

11 Source Code Listings