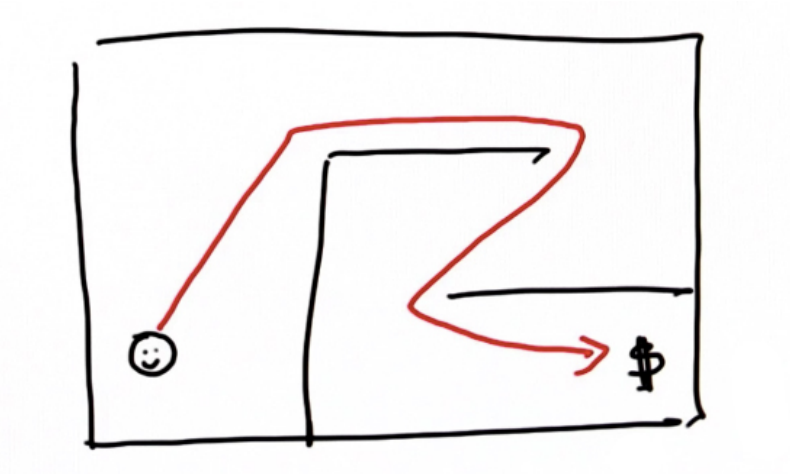
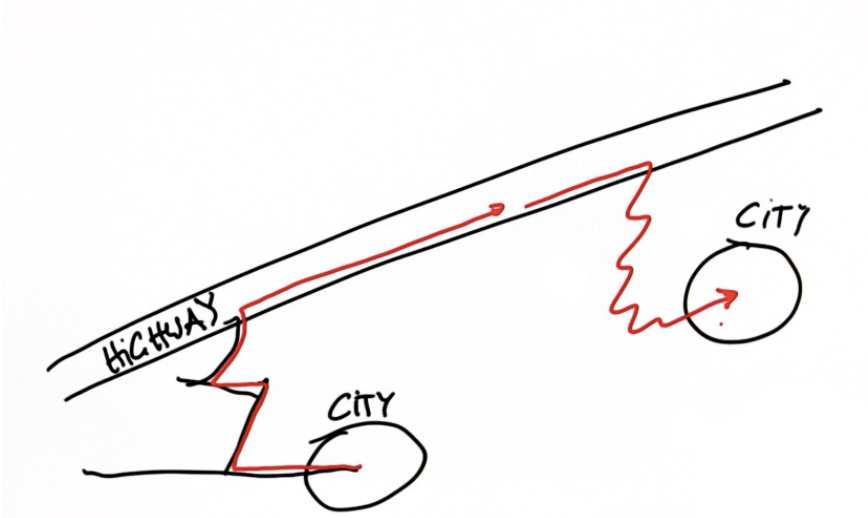
Udacity CS373: Programming a Robotic Car
Unit 4: Motion

Motion Planning (Motion Planning)

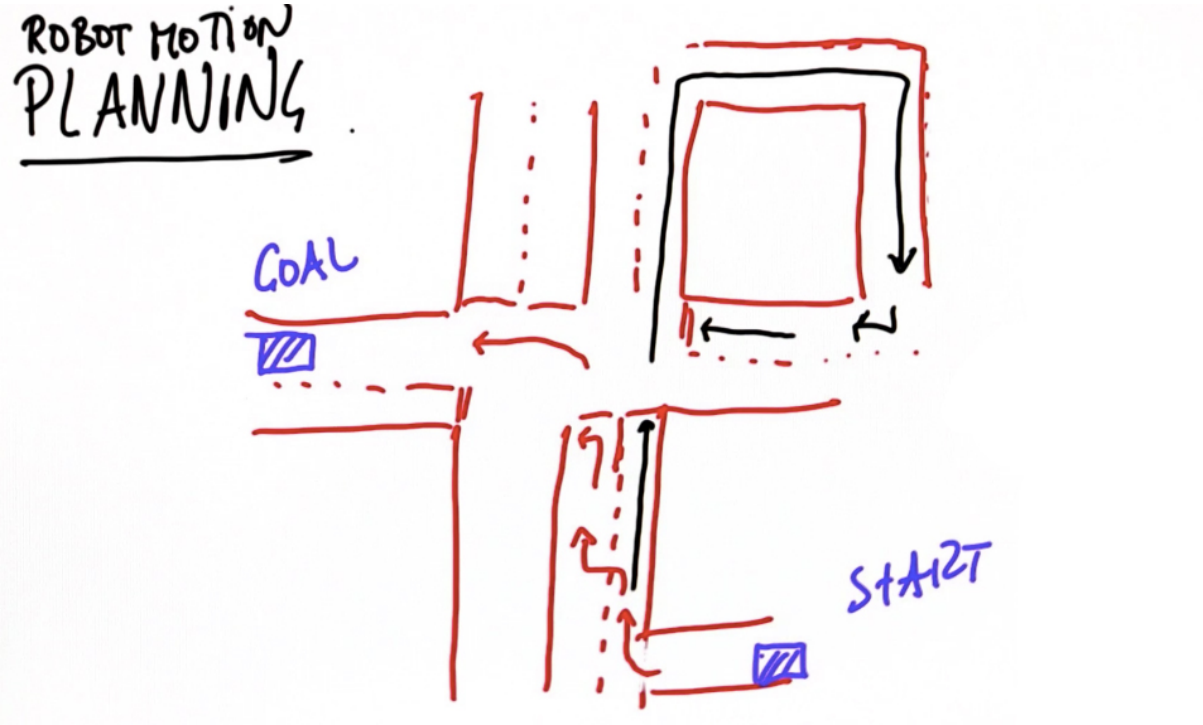
The fundamental problem in motion planning is that a robot might live in a world that looks like the one pictured below and will want to find a goal like $. The robot has to have a plan to get to its goal, as indicated by the red line route.



This same problem occurs for a self-driving car that lives in a city and has to find its way to its target location by navigating through a network of streets and along the highway.



For the self-driving car, other obstacles besides winding streets, may become a problem. For example, in this bird's-eye view of the car's route (red arrows), the car would have to turn right, shift lanes and then make a left-hand turn, crossing opposing traffic to reach its goal. Alternatively, the car could opt to take a detour (black arrows) if there is some obstacle in the lane it was trying to change to. The process of getting a robot from a start location to a goal location is referred to as **robot motion planning**, or just planning.



In this unit you will learn about discrete methods for planning in which the world is chopped into small bins and then in the next unit you will learn about continuous motion using those plans.

The planning problem is:

Given:

Map
Starting Location
Goal Location
Cost (time it takes to drive a certain route)

Goal:

Find the minimum cost path.

Computing Cost (Compute Cost)

Suppose you live in a discrete world that is split up into grid cells and your initial location is facing forward, while your goal location is facing to the left.

Assume that for each time step you can make one of two action -- either move forward or turn the vehicle. Each move costs exactly one unit.

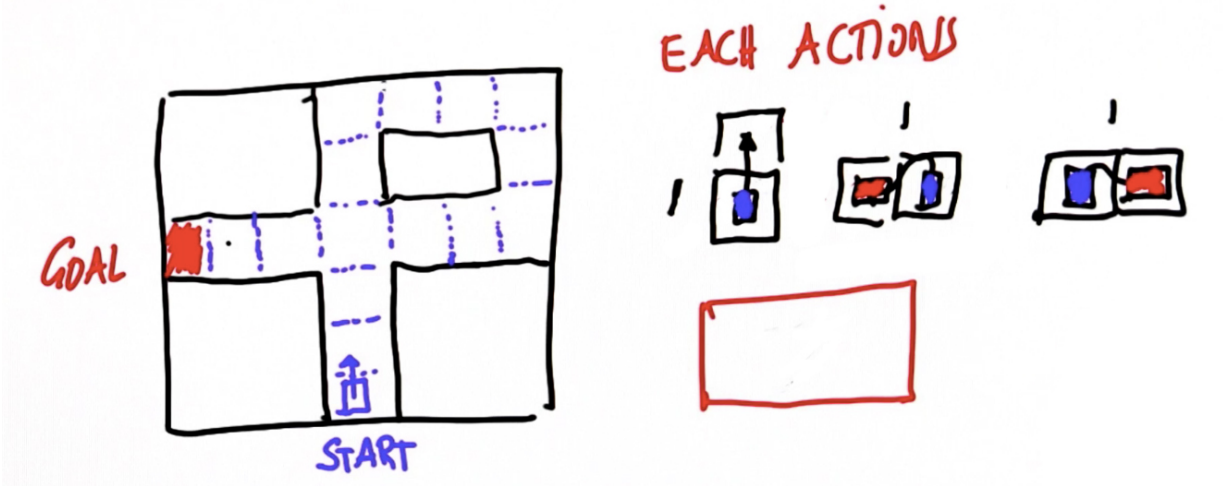
Q-1: Quiz (Compute Cost)

What is the total cost (number unit of cost) to move from start to goal?

Answer to Q-1

Q-2: Quiz (Compute Cost 2)

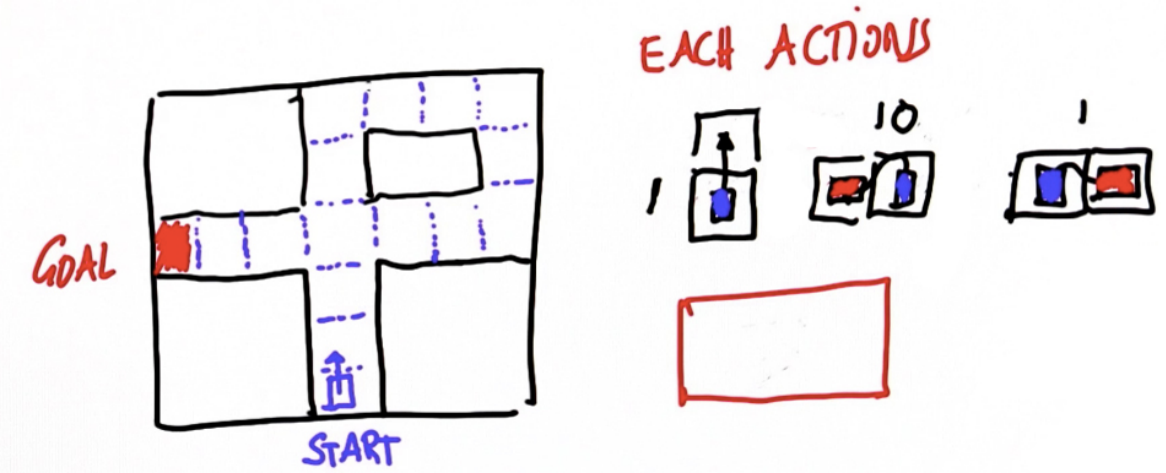
What if the action model changes so that you can take one of three actions: forward, turn left and then go forward, or turn right and then go forward. Each of these actions is again one unit of cost. What is the total cost of the optimum path to get from the start to the goal.



Answer to Q-2

Q-3: Quiz (Optimal Path)

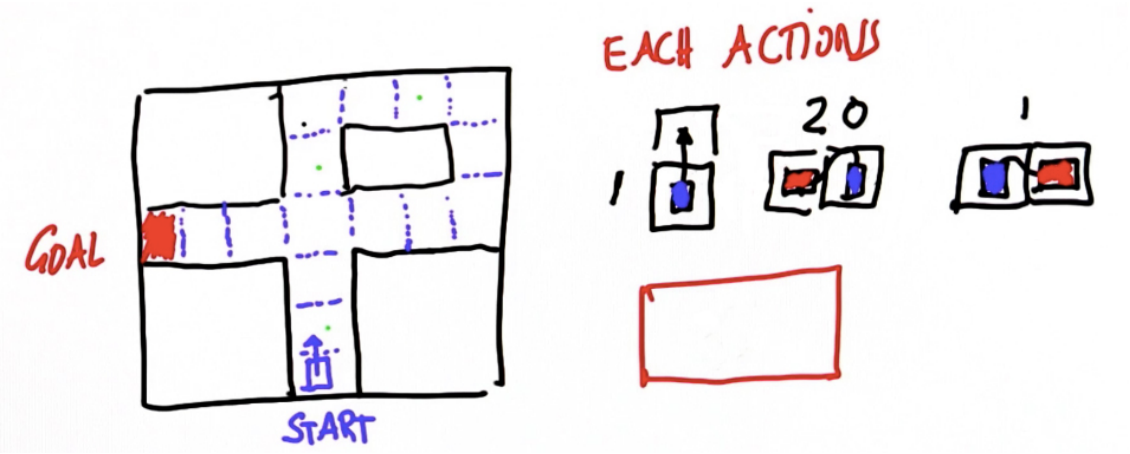
For this quiz, suppose left turns are more expensive (delivery driver services, such as UPS or FedEx, do try and avoid routes that require a left turn during rush hour). If moving forward is one unit of cost, taking a right turn is one unit of cost and taking a left turn is ten units of cost, what is the cost of the optimum path.



Answer to Q-3

Q-4: Quiz (Optimal Path 2)

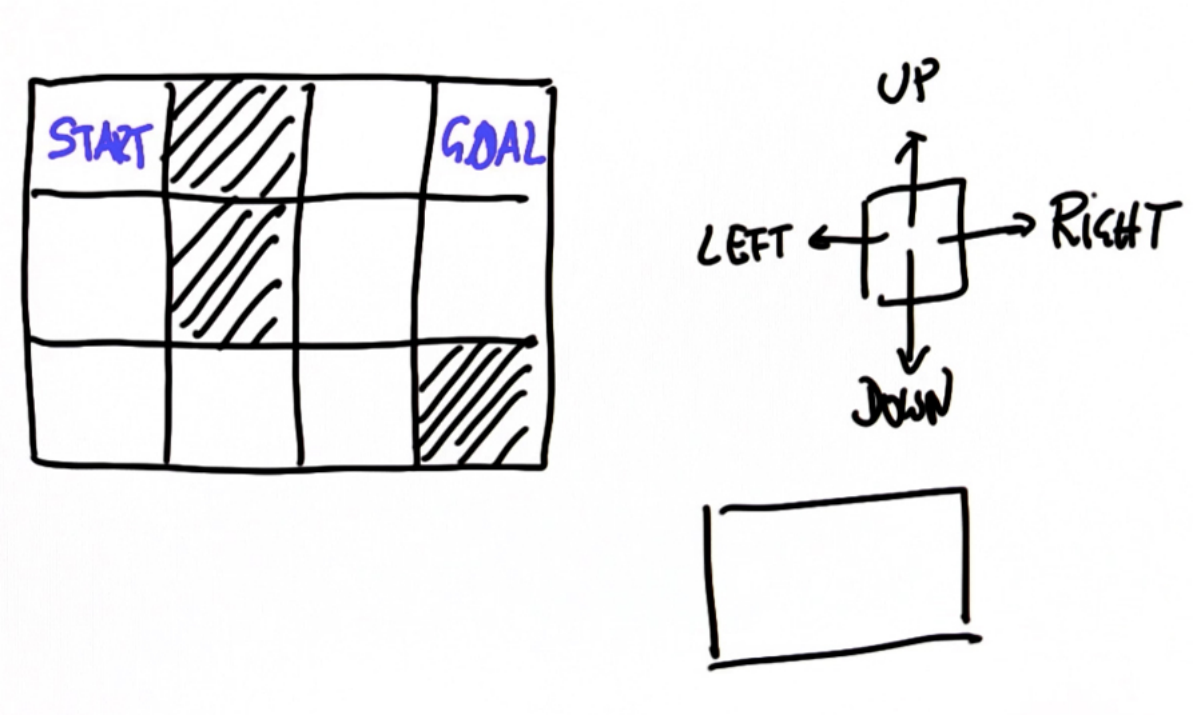
Now, suppose it is a 20-unit cost to take a left turn. What is the cost of the optimum path?



Answer to Q-4

Q-5: Quiz (Maze)

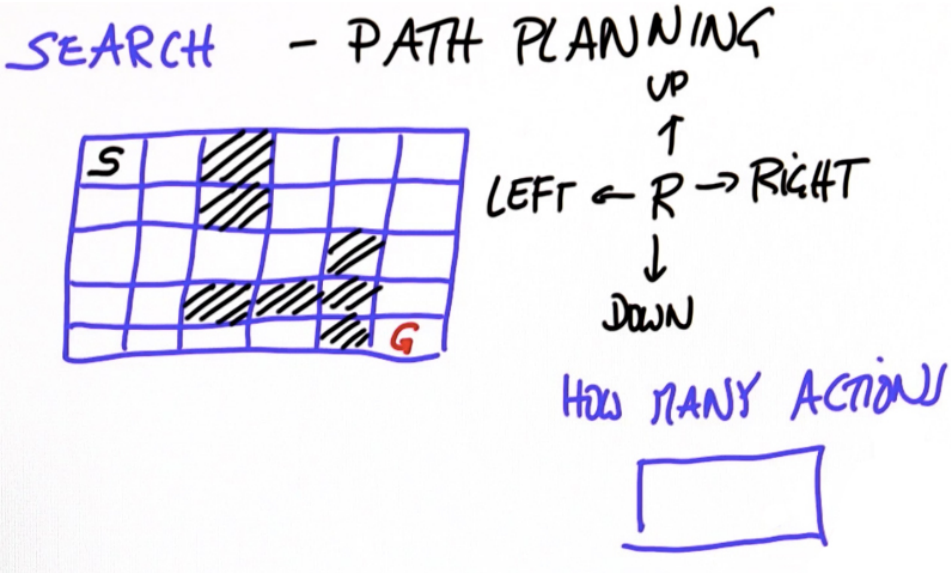
Given the maze below, say you have a robot that can move up, down, left and right. How many steps will it take the robot to move from the start to the goal position.



Answer to Q-5

Q-6: Quiz (Maze 2)

You can look at the path planning problem as a search problem. For example, say you have the grid world below, and a robot that can move up, down, left, or right. Assume that the robot carries out every action with absolute certainty. For the path planning problem, you want to find the shortest sequence of actions that leads the robot from the start position to the goal position. How many actions do you think the robot needs to take?

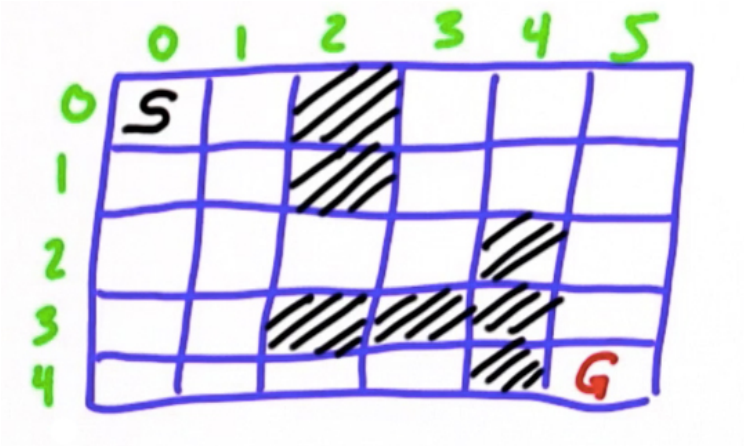


Answer Q-6

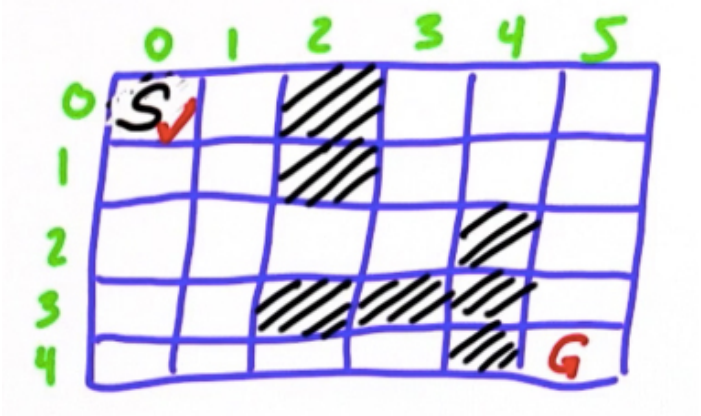
Writing a Search Program (First Search Program)

Now, the question is, can you write a program that computes the shortest path from the start to the goal?

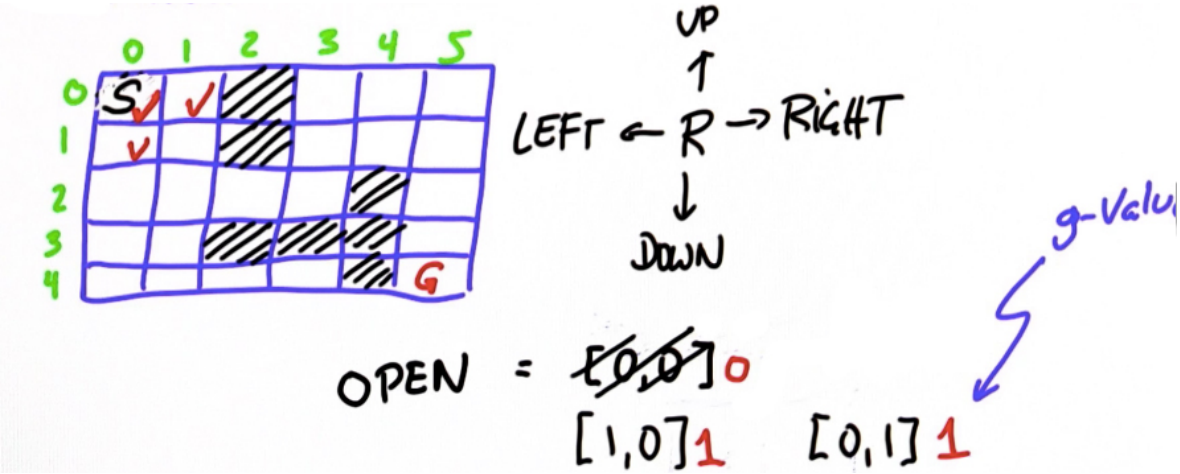
The first step towards writing this program is to name the grid cells. Across the top, name the columns zero to five, and along the side you can name the rows zero to four. Think about each grid cell as a data point, called a **node** and the goal point is called the **goal node**.



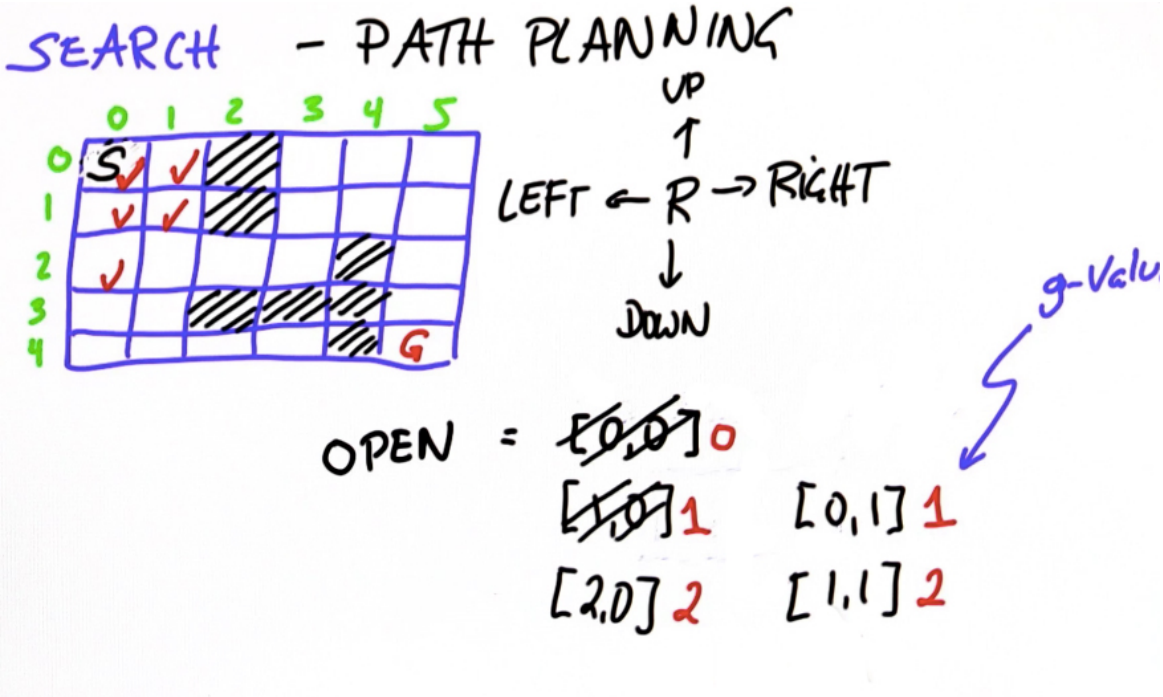
From here you will make a list of nodes that you will be investigating further -- that you will be expanding. Call this list **open**. The initial state of this list is **`[0,0]`**. You may want to check off the nodes that you have already picked, as shown here with the red check mark.



Now, you can test whether this node is your final goal node -- which, just by looking at the grid you can tell that it obviously is not. So, what you want to do next is to expand this node by taking it off the **open** list and looking at its successors, of which there are two: **`[1,0]`** and **`[0,1]`**. Then, you can check those cells off on your grid. Most importantly, remember to note how many expansions it takes to get to the goal -- this is called the **g-value**. By the end of your planning, the g-value will be the length of the path.



Continue to expand further. Always expand the cell with the smallest g-value, but since the two above are equivalent, it doesn't make a difference which one you expand first. In the image below, the first cell is expanded to include the unchecked cells, **`[2,0]`** and **`[1,1]`**, each with a g-value of two.



The next lowest g-value is the cell **[0,1]**, but since the surrounding cells, **[0,0]** and **[1,1]**, are already checked it cannot be expanded any further. The process of expanding the cells, starting with the one with the lowest g-value, should yield the same result you found as the number of moves it takes to get from the start to the goal -- 11.

Q-7: Quiz (First Search Program)

Write a piece of code that implements the process just described. Here is some code to get your started:

```
# ----------
# User Instructions:
#
# Define a function, search() that takes no input
# and returns a list
# in the form of [optimal path length, x, y]. For
# the grid shown below, your function should output
# [11, 4, 5].
#
# If there is no valid path from the start point
# to the goal, your function should return the string
# 'fail'
# ----------

# Grid format:
#   0 = Navigable space
```

```
#   1 = Occupied space

grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 1, 0],
        [0, 0, 1, 1, 1, 0],
        [0, 0, 0, 0, 1, 0]] # this is the same as image grid

init = [0, 0] # starting position
goal = [len(grid)-1, len(grid[0])-1] # goal position

delta = [[-1, 0], # go up
         [0, -1], # go left
         [1, 0], # go down
         [0, 1]] # do right

delta_name = ['^', '<', 'v', '>'] # ignore for now

cost = 1 # each move costs 1
def search():
    # ----------------------------------------
    # insert code here and make sure it returns the appropriate result
    # ----------------------------------------
```

The code you write should output triplets, where the first value is the g-value and the next two are the x and y coordinates. Then, the program should retrieve the smallest g-value from the list, expand it and select again the list with the smallest g-value. Your program should continue through this process until it has reached the last position, where it should only output the triplet **`[11, 4, 5]`**. If there is no way to reach the goal point the output should be **`fail`**.

Answer to Q-7

Q-8: Quiz (Expansion grid)

In this programming quiz you have to print out a table, called **`expand[]`**.

Expand is a table, the same size as the grid, that maintains information about at which step the node was expanded. The first node that was expanded was at the start position, at a time **`0`**. The next node that was expanded was the node to the right, at time 1. This is an example output:

```
[ 0,  1,  -1,  11,  15,  18]
[ 2,  3,   5,   8,  12,  16]
[ 4,  6,  -1,  13,  -1,  19]
[ 7,  9,  -1,  17,  -1,  21]
[10, 14,  -1,  20,  -1,  22]
```

In this table every node that has never been expanded (including obstacle nodes) should have a value of *-1*. When a node is expanded, it should get a unique number that is incremented from expansion to expansion, and counts from **0** all the way to **22** (in this case) for reaching the goal state.

For a second example of how the code should work, modify the grid to look like this (the road to goal is blocked):

```
grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0]
```

In this case the search fails and in the expansion list you find that all the nodes on the right side have never been expanded, and the expand list will look similar to this:

```
[0, 1, -1, -1, -1, -1]
[2, 3, -1, -1, -1, -1]
[4, 5, -1, -1, -1, -1]
[6, 7, -1, -1, -1, -1]
[8, 9, -1, -1, -1, -1]
```

Warning! Depending on how you break ties when choosing which node to expand next, your table might look a little bit different than the one displayed here. For example, the position of 1 and 2 might be swapped. However, if there is a full blockage of road, the right side should never expand.

Answer to Q-8

Q-9: Quiz (Print Path)

Try this completely different and challenging programming assignment. This has nothing to do with expand. You will have to print out the final solution for the path. Please implement a new data structure for this. Below is the output that your code should produce:

```
['>', 'v', ' ', ' ', ' ', ' ']
[' ', '>', '>', '>', '>', 'v']
[' ', ' ', ' ', ' ', ' ', 'v']
[' ', ' ', ' ', ' ', ' ', 'v']
[' ', ' ', ' ', ' ', ' ', '*']
```

Again, this is a little ambiguous, because your code might chose to take a different first step.
For your program indicate directions by using the variable **delta_name**, which is provided within the code and that corresponds to the four actions that are defined in **delta**. At the end of the path, a star (*) indicates the final goal.

If we change the grid according to these specifications:

```
grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 1, 0]]
```

the program should output a different path:

```
[['>', 'v', ' ', '>', '>', 'v']
[' ', 'v', ' ', '^', ' ', 'v']
[' ', 'v', ' ', '^', ' ', 'v']
[' ', 'v', ' ', '^', ' ', 'v']
[' ', '>', '>', '^', ' ', '*']]
```

This is a really challenging task! While there is not a lot of code to write, you have to think about how to store actions and how to assign them to the end results. Don't worry if you don't get this right. Follow along and you will still understand how everything works and you will see an explanation in the answer part of these notes.

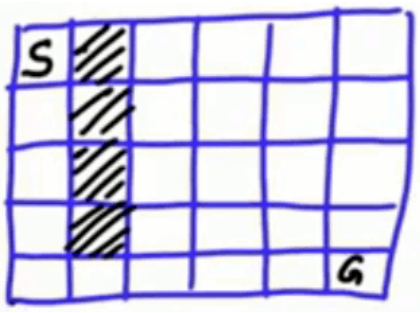
Answer to Q-9

A* search algorithm (A Star)

The A* (A-star) algorithm can be used for our path finding problem. It is a variant of the search algorithm that we implemented, that is more efficient because you don't need to expand every node. The A* algorithm was first described by Peter Hart, Nils Nilsson and Bertram Raphael in 1968.

If you understand the mechanism for searching by gradually expanding the nodes in the open list, A* is *almost* the same thing, but not quite.

To illustrate the difference, consider the same world as before, but with a different obstacle configuration:



This is one example where A* performs really well. Obviously, from the start node we have to go straight down. But from the corner we still have to search for the best path to the goal.
Here is the same world configured in code:

```
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0]]
```

When you run the code you wrote before and print out the expansion list, you will see that you have to travel down to node **4**, and then expand into the open space, until finally you hit the goal node at **15**:

```
[0, -1, -1, -1, -1, -1]
[1, -1, 12, -1, -1, -1]
[2, -1,  9, 13, -1, -1]
[3, -1,  7, 10, 14, -1]
[4,  5,  6,  8, 11, 15]
```

This took **16** expansions (including the initial zero-expansion) to get us to the goal node.

If you use the A* algorithm, you'll get this output:

```
[0, -1, -1, -1, -1, -1]
[1, -1, -1, -1, -1, -1]
[2, -1, -1, -1, -1, -1]
[3, -1, -1, -1, -1, -1]
[4,  5,  6,  7,  8,  9]
```

Notice how that took only **10** expansions to get to the goal. It expands down to **4** and then goes straight to goal, never expanding to the area above. The A* algorithm somehow magically knows that any other path to the goal will be longer than going straight.

Change the world to something more challenging. Put an obstacle right next to the goal:

```
grid = [[0, 1, 0, 0, 0, 0],
```

```
[0, 1, 0, 0, 0, 0],
[0, 1, 0, 0, 0, 0],
[0, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0]]
```

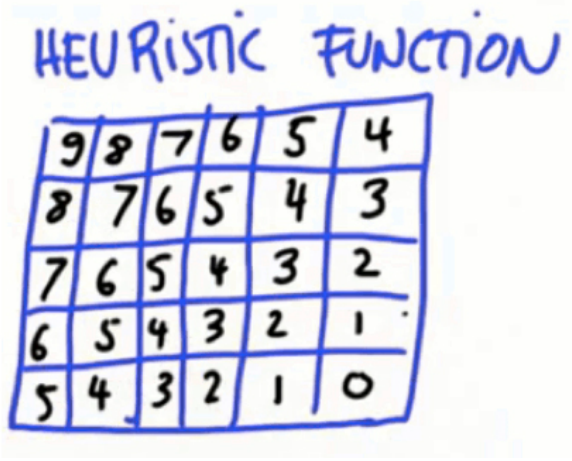
Run the A* search again to return:

```
[0, -1, -1, -1, -1, -1]
[1, -1, -1, -1, -1, -1]
[2, -1, -1, -1, -1, -1]
[3, -1,  8,  9, 10, 11]
[4,  5,  6,  7, -1, 12]
```

When A* encounters the obstacle, it expands up, but then goes right and then down to the goal again. So, it somehow does the minimum amount of work necessary to make the maximum progress towards the goal. This is the basic principle of A*.

A* uses a **heuristic function**, which is a function that has to be set up. If it's all zeroes, A* resorts back to the search algorithm already implemented. If we call the heuristic function ***h***, then each cell results in a value.

For example, what is the number of steps it takes to get to a goal if there are no obstacles:



Clearly these numbers are not really reflective of the actual distance to the goal, because they do not take into consideration the obstacles. In a world without obstacles, the heuristic function would give you the distance to the goal. So, the heuristic function has to be an optimistic guess of how far you are from the goal. In other words, for any cell (x, y) the heuristic function has to be an optimistic guess that is less than or equal than the actual distance to goal from the current x and y. This is written as:

h(x,y) ≤ actual goal distance from x,y

Although this sounds ad hoc, very often you can find good heuristic functions quite easily. In our

example, knowing that the agent can move up, down, left, or right, it is easy to see how many steps it would take for the agent to get to the goal state if there were no obstacles -- the table above, which can be easily generated automatically. In reality this is an underestimate. If there are more obstacles, it will take more steps to get to the goal.

The beauty of the heuristic function is that it doesn't have to be accurate to work. If it were accurate, you probably would have already solved the planning problem.

The heuristic function has to be a function that helps you to find out where to search next in the case of ties, and it has to be just so that it underestimates -- or at best equals the true distance from the goal, as written in the formula above.

The self-driving car uses a function like this to solve free-form navigation problems. It boils it down to the distance to a target location, not to the number of grid cell steps.

If you want to find out more about how different search algorithms work, check out the videos about this topic at Unit 2. Problem Solving at AI-Class website or at Unit 2 at this student developed website where you can watch the videos alongside subtitle transcript.

You should have an idea of what an heuristic function might look like. There are many, many valid heuristic functions, for example setting everything to zero, which would not really help us.

Examples:

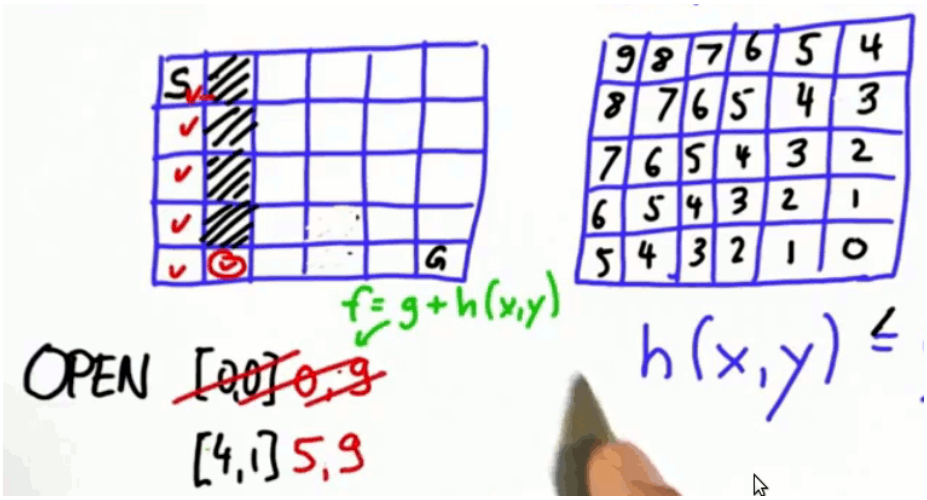
```
[9, 8, 7, 6, 5, 4]
[8, 7, 6, 5, 4, 3]
[7, 6, 5, 4, 3, 2]
[6, 5, 4, 3, 2, 1]
[5, 4, 3, 2, 1, 0]
```

With this, the key modification to our search algorithm is really simple. You still use the **open** list to add the state and write the **g**-value. In this list you also include the cumulative **g**-value plus the heuristic value **h(x,y)**, which you can call the **f** value:

```
f = g + h(x,y)
```

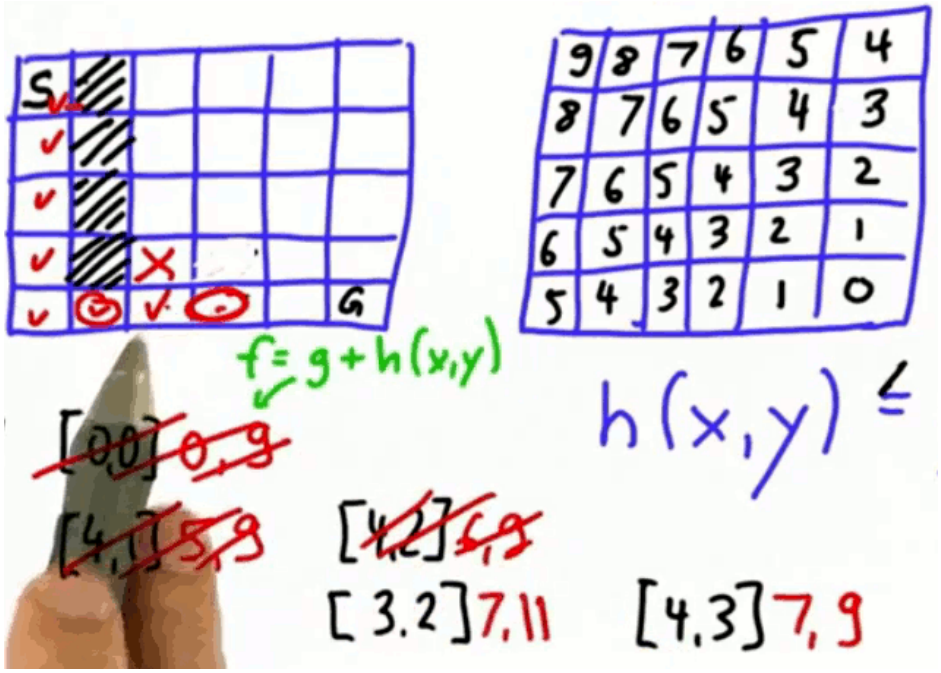
Now, you expand and remove the element with the lowest **f** value, not the lowest **g** value. This is the core concept of A*.

Example:



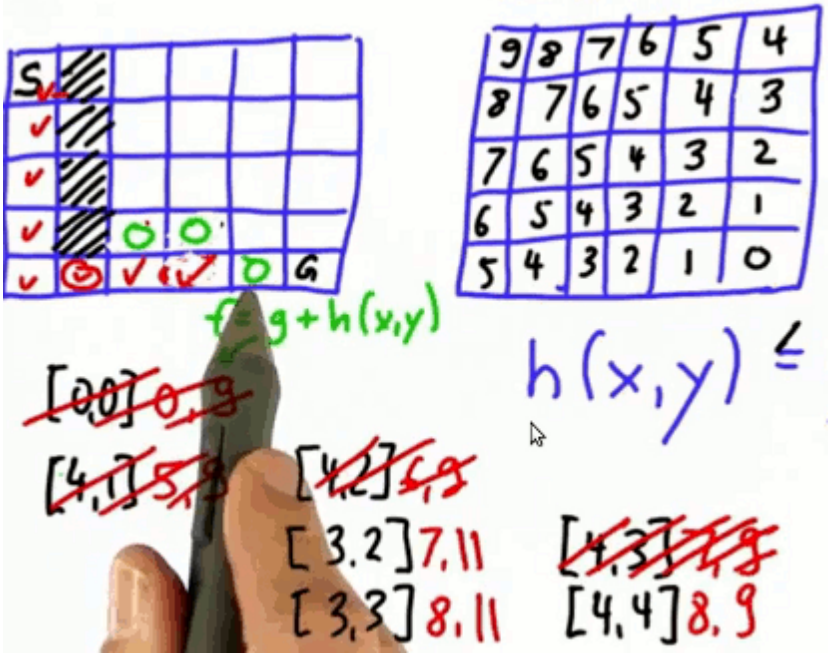
Look at the **g** and **h** values for the node that is circled, at coordinates **[4,1]**, and **g** is **5**, because it is the 6th node we are expanding. You can obtain the **f** value of **9**, by adding **h(4,1) = 4** to the **g**. In the next step expand the node at **[4,2]** and you will see that the **g** value is increased by one and is now **6**, but the **f** value is still **9** because **h(4,2) = 3.**

This next step is the interesting one, because you finally have two nodes to choose from.



The node at **[3,2]** is marked with an **X**, and the second node at **[4,3]**, marked with **0**. Both of them have the **g** value of **7**. But now comes the interesting part - when you add the **h(x,y)** to **g**, the resulting **f** values are different -- **11** for the first and **9** for the second node because the heuristic function values are different, as you can see in the heuristics table. What this reflects is that according to heuristics, the second node is two steps closer to the goal than the first node. And A* will expand the node at **[4,3]** that has the smallest **f** value.

When you expand the node, you will find that it has two valid neighbours; one up, with coordinates **`[3,3]`** and one to the right with coordinates **`[4,4]`**. Increment **`g`** again by **`1`**, and they both have a value of **`8`** there, but the **`f`** values are different again because **`h(3,3)`** is **`3`**, but **`h(4,4)`** is **`1`** and you have a preference in expanding the next node again, because of the heuristic function.



Now there are three nodes in the **open** list, marked with green circles in the image. But the one on the right will be expanded because it has the lowest **`f`** value. In in the next step you will reach the goal node and the search will be done. Nothing above the actual path will be expanded. That feels like magic, but the key thing here is that by providing additional information via the heuristic function, you can guide the search, and when you have a deadlock, you can pick a node that looks closer to the goal state and as a result you will likely make more progress towards the goal.

Q-10: Quiz (Implement A*)

Now, you can write an algorithm that implements A* by maintaining not only the **`g`**-values, but also the **`f`**-values, which is the **`g`** value added to the heuristic. Your output should be an **expand** table that looks exactly like this:

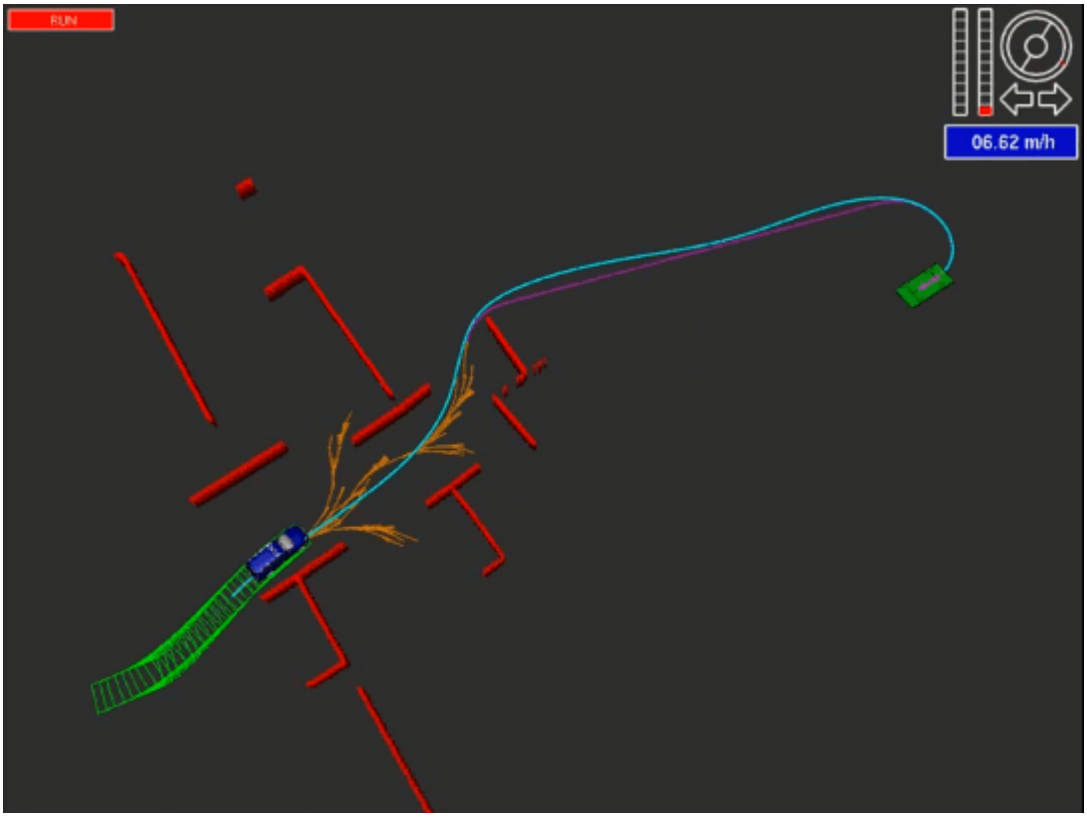
```
[1, -1, -1, -1, -1, -1]
[2, -1, -1, -1, -1, -1]
[3, -1,  8,  9, 10, 11]
[4,  5,  6,  7, -1, 12]
```

In your table **`-1`** should appear not only where there are obstacles, but everywhere it has not been expanded according to heuristics. So, for this quiz, please write a program that generates the same output.

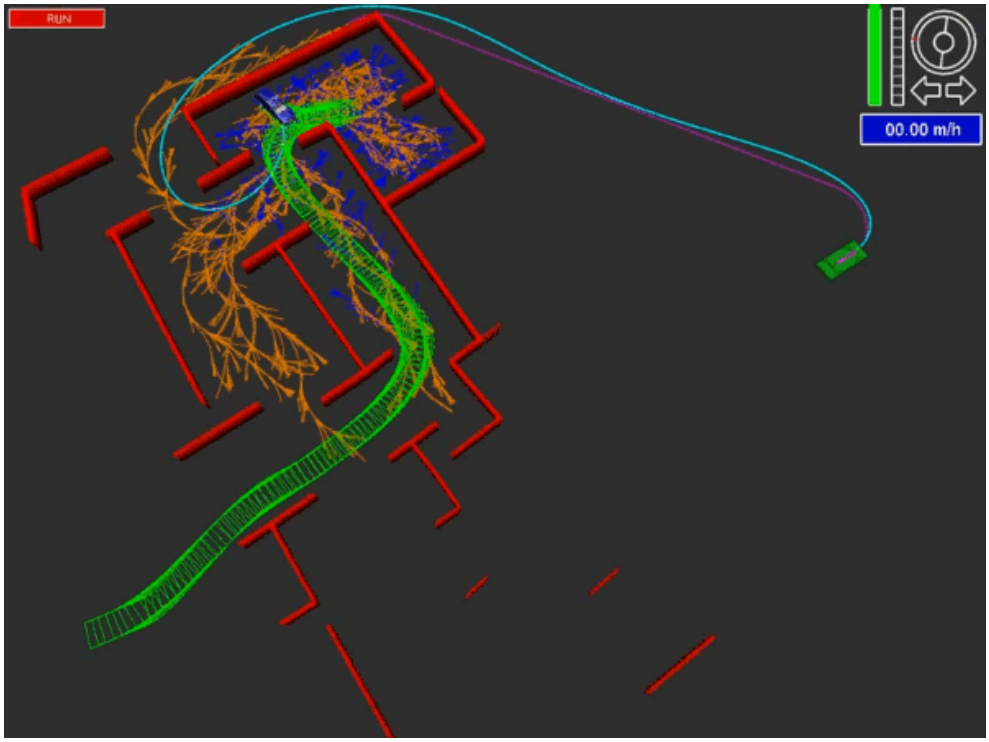
Answer to Q-10

Examples of A* in the real world (A Star In Action)

Here you can see an actual implementation from the DARPA Urban Challenge. A Stanford car trying to find a way through a maze:

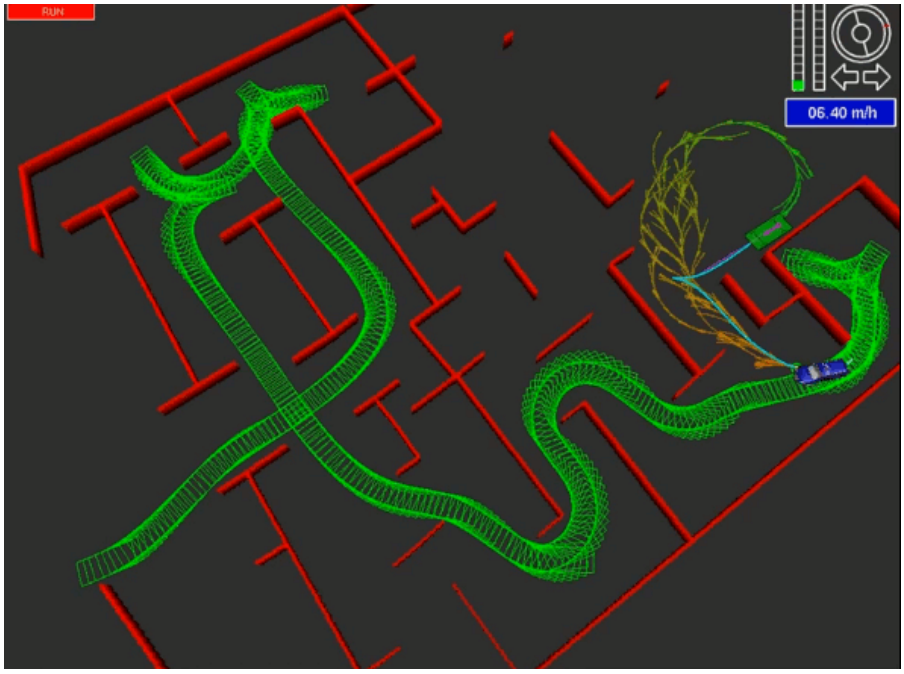


In the video you can see the maze constantly changing as the car moves. This reflects the fact that the car uses sensors to see obstacles, and obstacles are sometimes occluded, that means the car can only see them when they are nearby. What's really remarkable here is that the car is able to plan highly complex maneuvers towards the goal. At any point in time you can see it's best guess towards an open path to the goal. The orange trees are the A* search trees. They are not exactly grid trees because the Stanford car moves differently from a grid-based robot. The car can turn at different angles; each step is a different turning angle combined with a different forward motion. However, leaving this aside you have amazing trees that find a path all the way to the goal using A*.



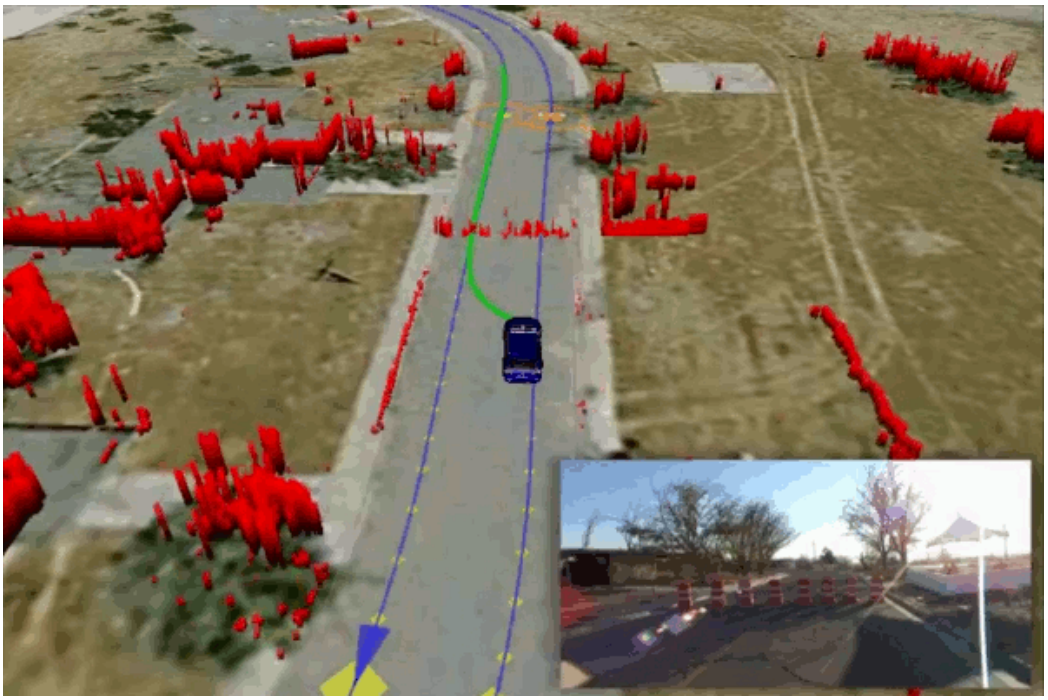
This implementation is so fast that it can plan these paths in less than 10 milliseconds for any location in this maze. It was faster than any other team at the DARPA Grand Challenge or the DARPA Urban Challenge.

The planing is repeated every time the robot cancels the previous plan. You can see additional adjustments in certain places and times. But as you go through the video, or look at these pictures, you can see that A* is planning with a simple Euclidean distance heuristic providing the car the ability to find a path to the goal.



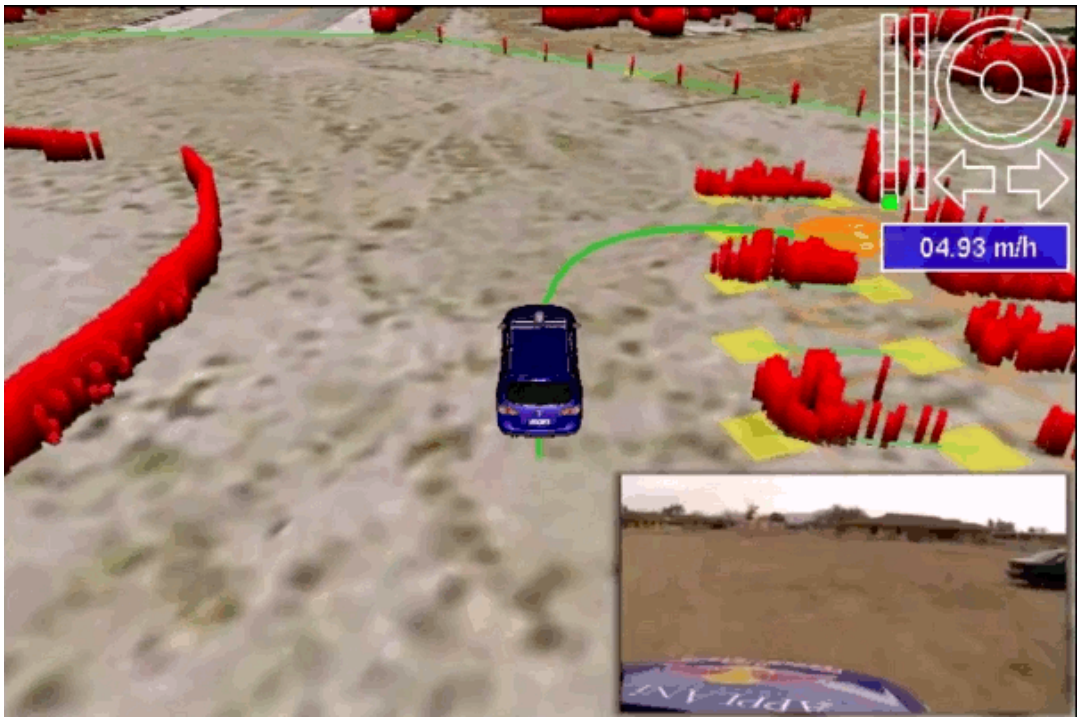
The big difference between implementing this yourself and implementing it on a grid is the motion model. You have to implement a robot that is able to turn and you have to write all the math of a robot that is able to turn and go forward. This robot can also revert, so going backwards becomes a distinct different action. But other than that it's essentially the same A* algorithm you just implemented. So, if you want to build a self-driving car, you now understand how to make a really complex, search algorithm to find a path to a goal.

This is a scene where DARPA trapped the self-driving car using a barrier that went all the way across the street.



The only way for the car to navigate is to take multiple U-turns, and it had to plan it all by itself by using A* planning. The car pulls up to the barrier, realizes there is no valid path, and invokes it's A* planner to come up with a turn-around maneuver that is not particularly elegant, but is super effective. The car was able to turn around by itself using A*, find the optimal plan to do so and then move on. Otherwise, it would have been stuck forever behind the obstacle.

The final video is an example of a parking situation where the car has to park in a parking space between two other cars. You can see how the obstacles are visible, how these other cars are visible and the vehicle, Junior, navigates an actual parking lot.



Again, the car is using A* to find the optimal path into this parking lot, and then back in and back out again. The planning run for each of these A* runs is less than 10 msec and the car was able to competently do this, even though in advance it had no clue where the obstacles were and where the parking spot was.

That is A* for robot path-planning, and what you have implemented yourself is at the core of it. Again, if you want to turn it into a real robotic motion algorithm, you have to change the motion model. And you have to see the next class where you will learn how to turn this into a continuous path.

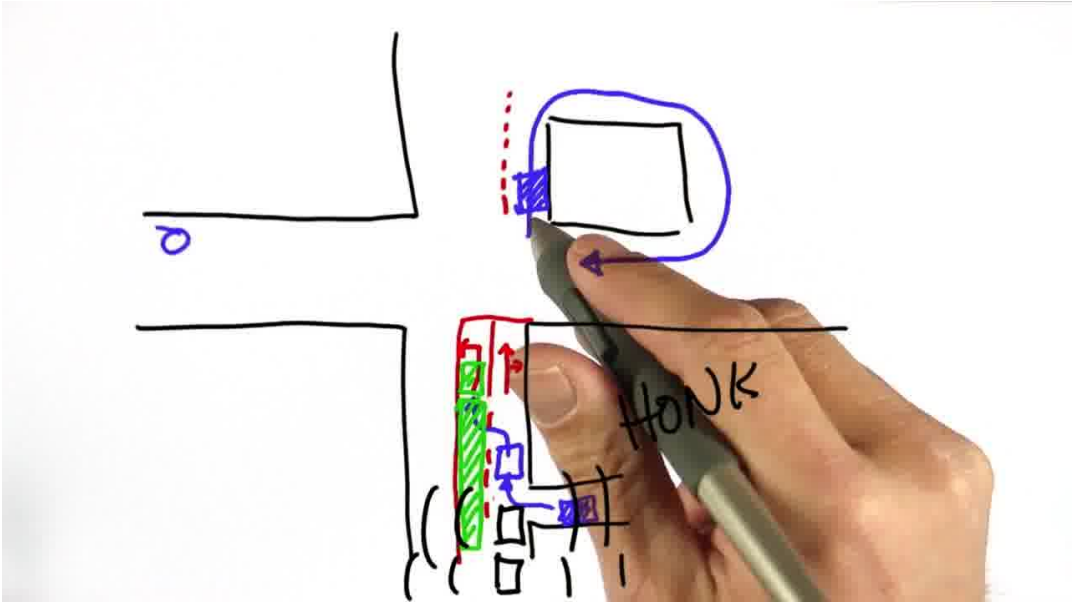
Dynamic Programming (Dynamic Programming)

Dynamic programming is an alternative method for planning. It has a number of advantages and disadvantages. Much like A* it will find the shortest path. Given a map and one or more goal positions, it will output the best path from any possible starting location.

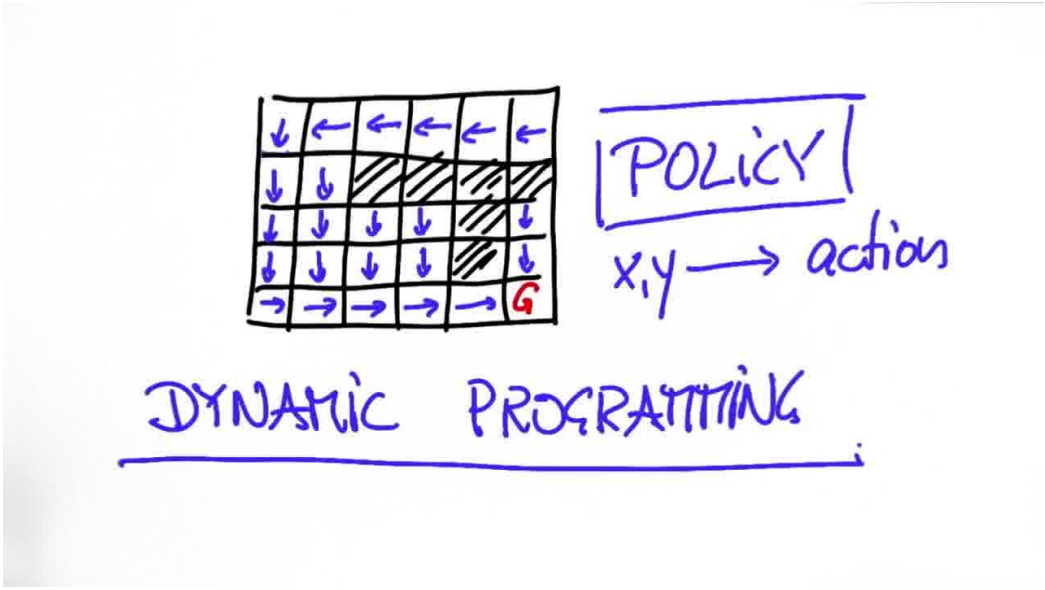
This planning technique is not just limited to one starting location, but from any starting location.

Consider this scenario: Our car, as indicated by the blue square at the bottom of the map, has a goal of the blue circle on the left side of the map. If you are able to perform a lane shift into the left lane, you will quickly reach your goal by turning left. However, there may be an obstacle such as a large truck in the left lane that does not allow us to make that turn. So we will instead have to go through the intersection and make three right turns in order to get into the lane that will lead to our goal.

This scenario shows the stochastic (probabilistic) nature of the world and that there is a need to plan not only for the most likely position, but for other positions as well. Dynamic programming is helpful because it does give us a solution for every location.



Redraw this environment as a grid with a goal location and a number of obstacles. Dynamic programming will give us an optimum action (called the **policy**) to perform for every navigable grid cell.



So how do we implement Dynamic Programming? Here is code which initializes our grid world, the initial state, and a goal state:

```
# grid format:
#   0 = navigable space
#   1 = occupied space
```

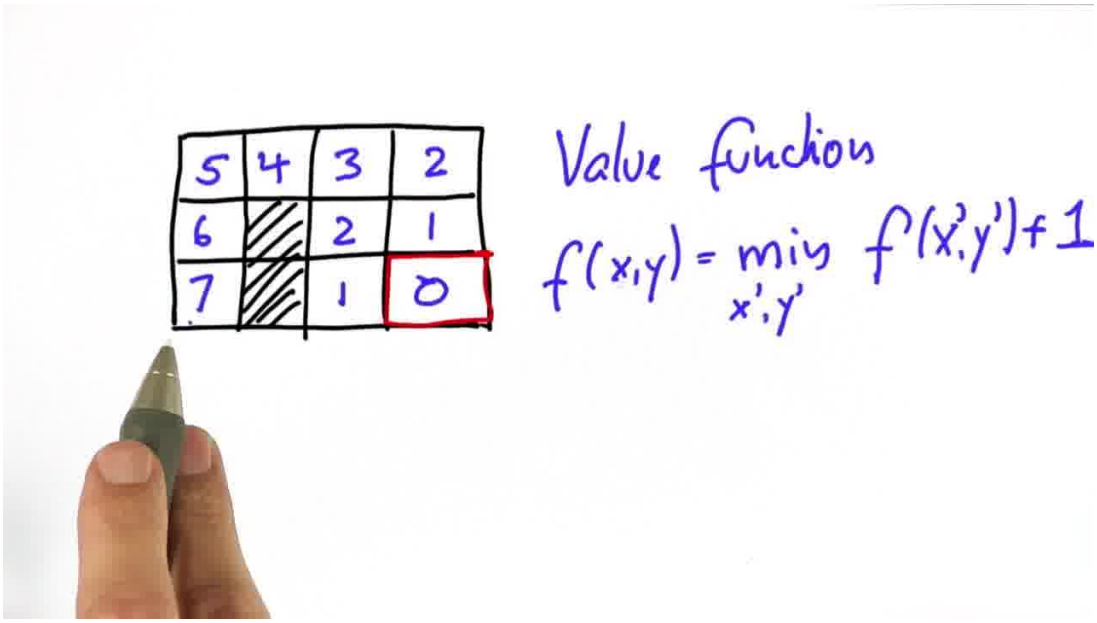
```
grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 1, 0]]

init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1]
```

When you run it, your output shows the grid overlaid with directional arrows representing the optimum policy for each location . Note that we compute the policy even for states that we are not likely to reach.

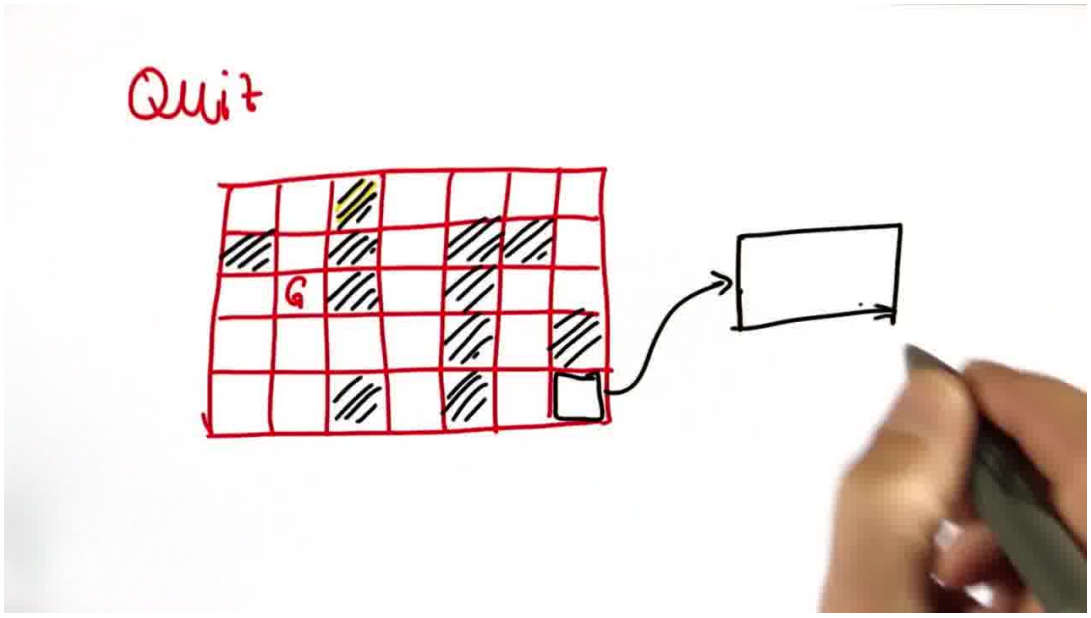
```
['v', 'v', ' ', 'v', 'v', 'v']
['v', 'v', ' ', '>', '>', 'v']
['v', 'v', ' ', '^', ' ', 'v']
['v', 'v', ' ', '^', ' ', 'v']
['>', '>', '>', '^', ' ', ' ']
```

Before looking at the implementation, consider a simpler example. In this example, the value function $f(x,y)$ associates to each cell the length of the shortest path to the goal. You can recursively calculate the value function for each cell by summing the optimum neighbor value and a movement cost of one.



Q-11: Quiz (Computing Value)

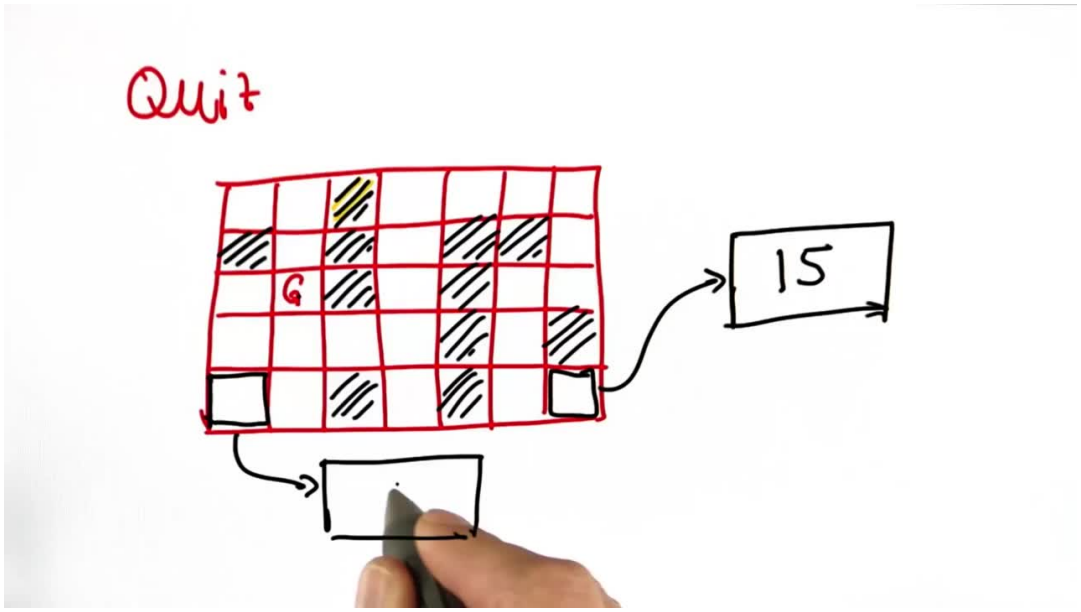
What is the value of the cell in the bottom right corner?



Answer to Q-11

Q-12: Quiz (Computing Value 2)

What is the value of the cell in the bottom left corner?



Answer Q-12

Q-13: Programming Quiz (Value Program)

Lets now implement something that calculates the value function. You have the familiar grid again, with a vertical wall up in the middle and a T-shaped obstacle on the bottom, and our goal location is in the bottom right corner:

```
grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 1, 0],
        [0, 0, 1, 1, 1, 0],
        [0, 0, 0, 0, 1, 0]]
```

When you run your value program you should get an output exactly like this:

```
[12, 11, 99,  7,  6,  5]
[11, 10, 99,  6,  5,  4]
[10,  9, 99,  5,  4,  3]
[ 9,  8,  7,  6, 99,  2]
[10,  9, 99, 99, 99,  1]
[11, 10, 11, 12, 99,  0]
```

Set the value of each obstacle to **99**. The value of the **goal** location is **0**, and then it counts up as you move to nodes farther away from the goal node. Your program should calculate the values for each node and return a table of values. For this grid example the output is completely unambiguous, and you should be able to get output exactly like in this example.

If you open up a node in the grid like this:

```
grid = [[0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 1, 0],
        [0, 0, 1, 1, 1, 0],
        [0, 0, 0, 0, 1, 0]]
```

You should get a completely different output:

```
[10,  9,  8,  7,  6,  5]
[11, 10, 99,  6,  5,  4]
[10,  9, 99,  5,  4,  3]
[ 9,  8,  7,  6, 99,  2]
[10,  9, 99, 99, 99,  1]
[11, 10, 11, 12, 99,  0]
```

To implement this, you are given some set-up code. You have the **delta** table with possible

movements. You are given the **cost_step** - the cost associated with moving from a cell to an adjacent one.

```
# ----------
# User Instructions:
#
# Create a function compute_value() which returns
# a grid of values. Value is defined as the minimum
# number of moves required to get from a cell to the
# goal.
#
# If it is impossible to reach the goal from a cell
# you should assign that cell a value of 99.

# ----------

grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0]]

init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1]

delta = [[-1, 0 ], # go up
         [ 0, -1], # go left
         [ 1, 0 ], # go down
         [ 0, 1 ]] # go right

delta_name = ['^', '<', 'v', '>']

cost_step = 1 # the cost associated with moving from a cell to an
adjacent one.

# ----------------------------------------
# insert code below
# ----------------------------------------

def compute_value():
```

Answer Q-13

Q-14: Quiz (Optimal Policy)

Given this grid with the goal in the bottom right corner

```
# grid format:
#   0 = navigable space
#   1 = occupied space

grid = [[0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 1, 0],
        [0, 0, 1, 1, 1, 0],
        [0, 0, 0, 0, 1, 0]]

init = [0, 0]
goal = [len(grid)-1, len(grid[0])-1]
```

The optimal policy is ambiguous. In some cells, there are multiple options that are equally optimal. For this quiz, write code that outputs the following policy:

```
['v', 'v', ' ', 'v', 'v', 'v']
['v', 'v', ' ', 'v', 'v', 'v']
['v', 'v', ' ', '>', '>', 'v']
['>', '>', '>', '^', ' ', 'v']
['^', '^', ' ', ' ', ' ', 'v']
['^', '^', '>', '<', ' ', ' ']
```

Answer to Q-14

Q-15: Quiz (Left Turn Policy)

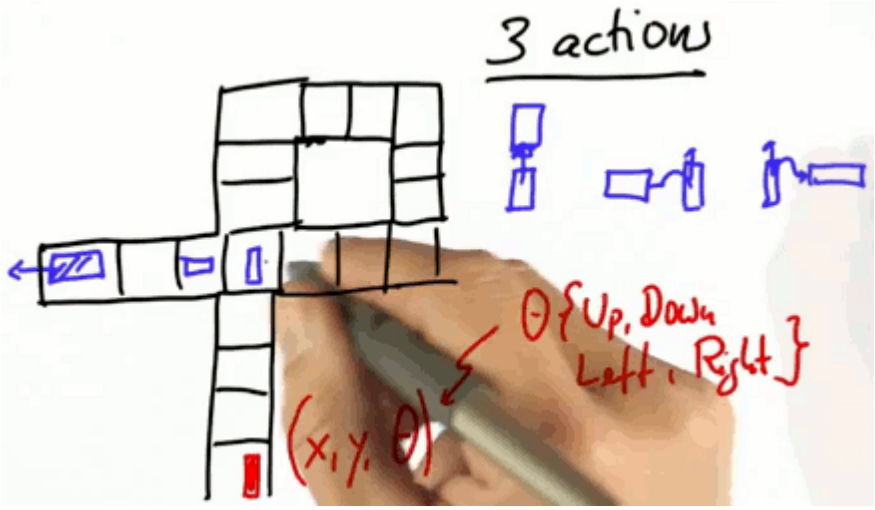
You know what's fun? Applying this to an actual car problem! The one you will solve will be a little bit simplified, as always, but it does relate to the real world path planning as it is done, for example, by Google Maps.

Suppose we have a car that has as its state an x, a y and an orientation (theta). Orientation -- for simplicity -- is chosen from four different directions: up, down, left and right. And, as in the quiz at the beginning, you need to get to the position to the left where the blue car is. Realize that now our state space is 3-dimensional, just like in our localization example.
Now you will have to implement a dynamic programming planner that gives the optimal path for going from the red start position to the blue end position and that lets you play with cost functions.

There are 3 possible actions :

1. Go straight.

2. Turn left then move
3. Turn right then move



The world looks a bit different now with representation of the streets that are navigable and with a loop on the right side. Remember that the state-space now is 3-dimensional, since each cell in the 2D grid now has an additional dimension associated with the robot's orientation.

```
grid = [[1, 1, 1, 0, 0, 0],
        [1, 1, 1, 0, 1, 0],
        [0, 0, 0, 0, 0, 0],
        [1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1]]
goal = [2, 0] # final position
init = [4, 3, 0] # first 2 elements are coordinates, third is direction
cost = [2, 1, 20] #the cost field has 3 values: right turn, no turn,
left turn
```

Your goal is to move to the cell **[3,0]** which is the same as the blue location on the drawing. The initial state that has coordinates of **[4,3]** also has an orientation of **0**.

```
# there are four motion directions: up/left/down/right
# increasing the index in this array corresponds to
# a left turn. Decreasing is is a right turn.

forward = [[-1,  0], # go up
           [ 0, -1], # go left
           [ 1,  0], # go down
           [ 0,  1]] # do right
forward_name = ['up', 'left', 'down', 'right']
```

This is where it gets interesting. The robot can have three actions, which you can add to the index orientation **-1, 0** or **1**. If we add **-1** we jump *up* in the cyclic array named **forward**, which is the same as making a right turn, while adding **+1**, is the same as turning left. If the orientation is left unchanged, the robot goes straight, which is indicated by this hash **#** symbol in the **action_name** table. The actions come with different costs.

```
# the cost field has 3 values: right turn, no turn, left turn
action = [-1, 0, 1]
action_name = ['R', '#', 'L']
```

Depending on the cost of the left turn, the car must choose the best path, and this is what you have to implement. Your output for a **cost = [2, 1, 20]** should like this. Notice that the goal state should be marked with a star **'*'**

```
[' ', ' ', ' ', 'R', '#', 'R']
[' ', ' ', ' ', '#', ' ', '#']
['*', '#', '#', '#', '#', 'R']
[' ', ' ', ' ', '#', ' ', ' ']
[' ', ' ', ' ', '#', ' ', ' ']
```

One more hint: The value function itself is 3-dimensional, and this is what was used in the example:

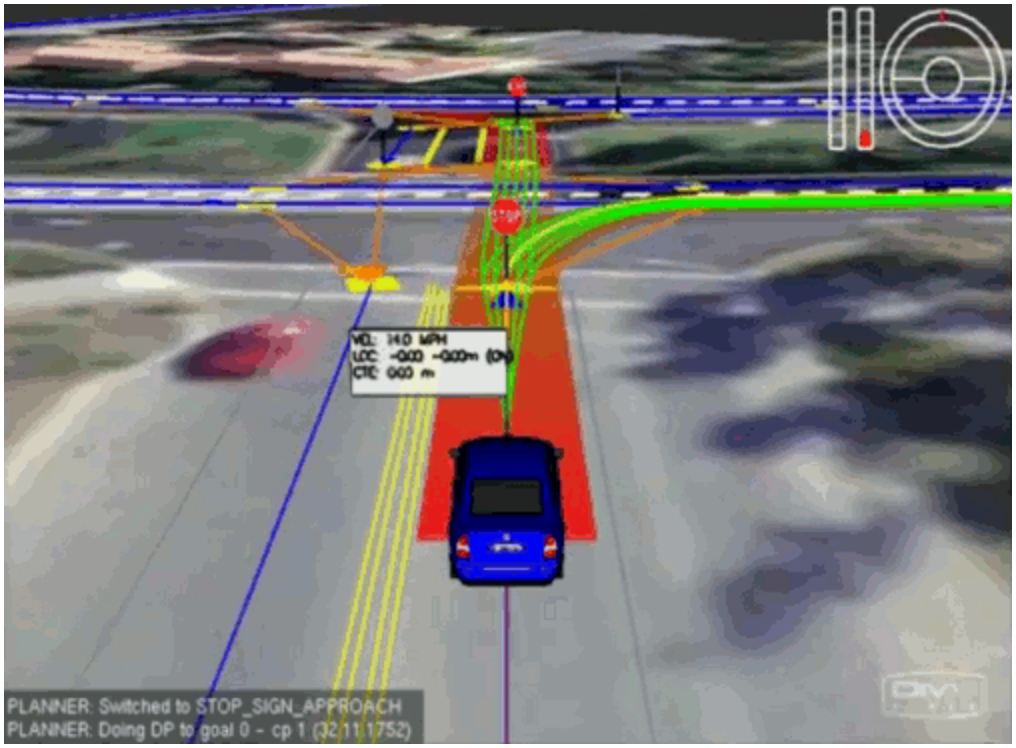
```
value = [[[999 for row in range(len(grid[0]))] for col in
range(len(grid))]],
[[999 for row in range(len(grid[0]))] for col in range(len(grid))]],
[[999 for row in range(len(grid[0]))] for col in range(len(grid))]],
[[999 for row in range(len(grid[0]))] for col in range(len(grid))]]]
```

While this is not necessarily the most efficient, it does have four identical arrays the size of the grid concatenated into a megagrid and initialized by a very large value of 999 in this case. You need structures just like these but it turns out that this makes it more difficult to write the code.
This is the last quiz and programming assignment in this unit, and it might take you some time to write this code.

Answer to Q-15

Conclusion

Here is another visualization of the Stanford racing car Junior in action, applying the same algorithm for actual driving.



If you watch the video, you can see how the car changes it's path to the goal according to rule changes. it works the same way than the program you wrote. Your program could effectively drive this car and by changing the cost functions, as you can see in the video, the car would be able to obtain its goal in the optimal way relative to the costs that you have given him.

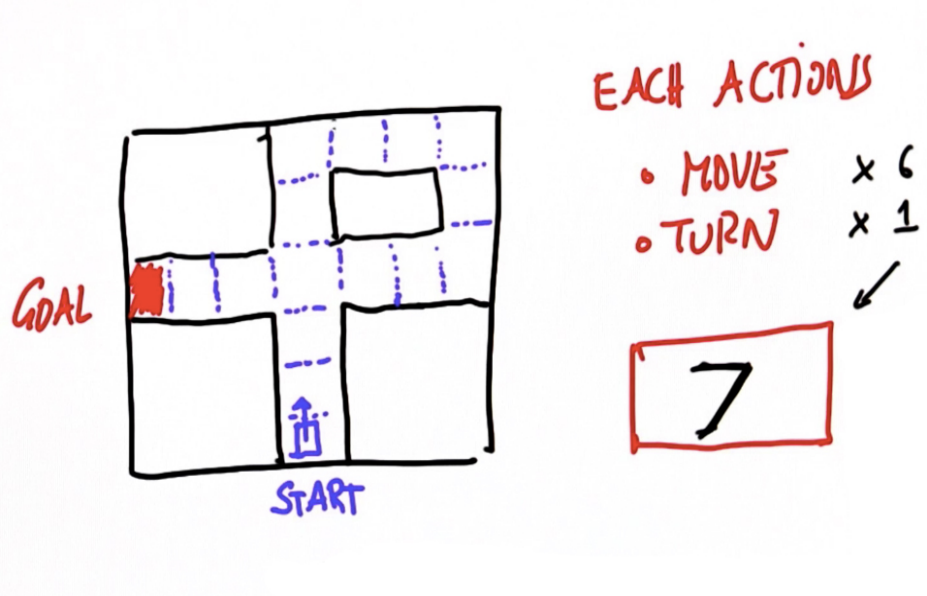
Congratulations! You have made it through the first motion class!

In this class the world was discrete and we looked at 2 planning algorithms - A*, which uses a heuristic to find a path and dynamic programming, which finds an entire policy that has a plan for every location. You implemented both of them. You have even done this in 3D, which is quite an achievement. Congratulations on getting so far! You really now understand two of the main paradigms by which robots make motion decisions, and they are also some very major paradigms in artificial intelligence in general.
In the next class we will talk about how to turn this into an actual robot motion. We will talk about continuous state spaces and we will talk about what is called "control", in which we make the robot move.

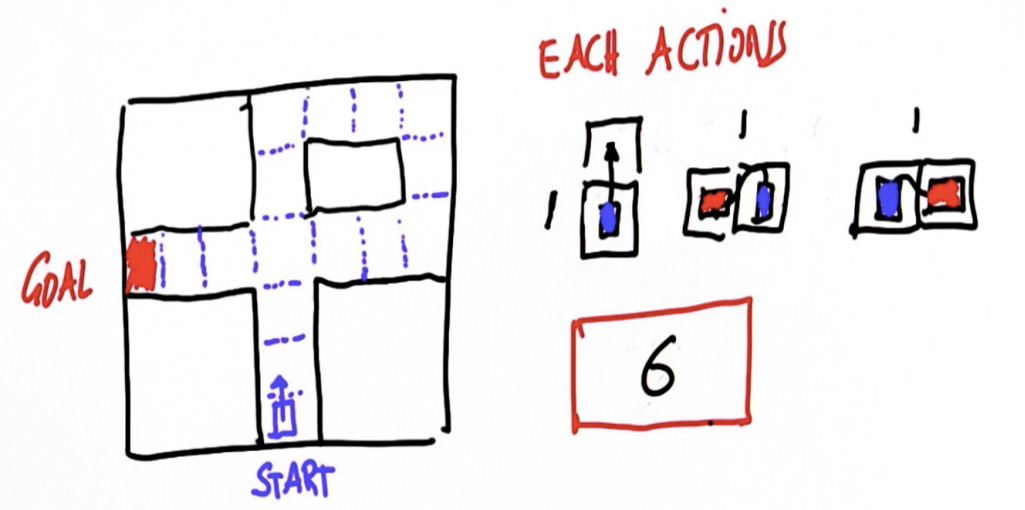
See you next week!

Answer Key

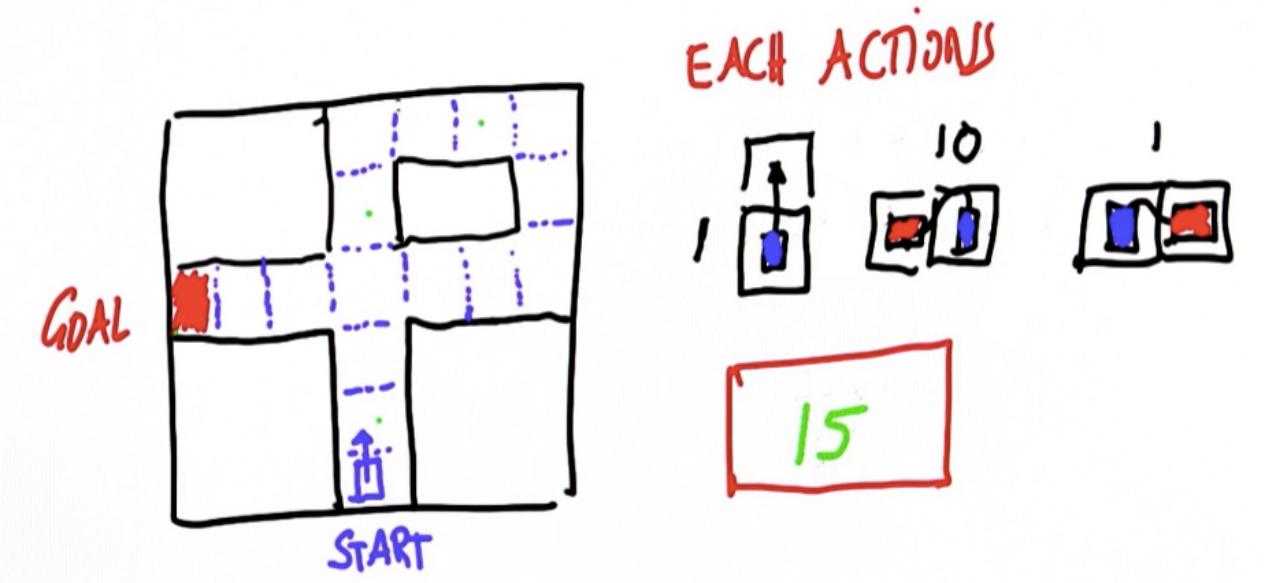
A-1: Answer (Compute Cost)



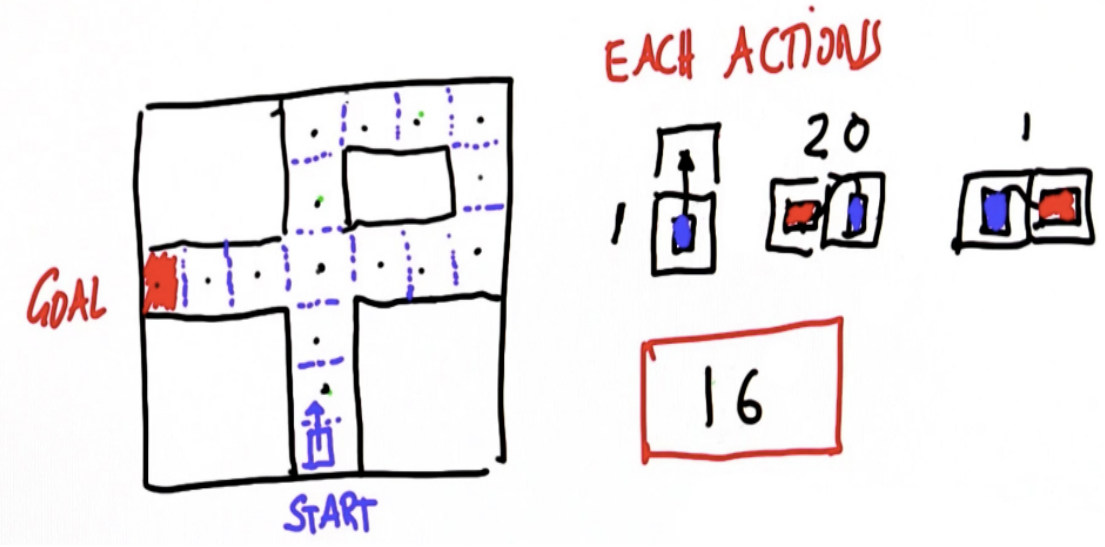
A-2: Answer (Compute Cost 2)



A-3: Answer (Optimal Path)

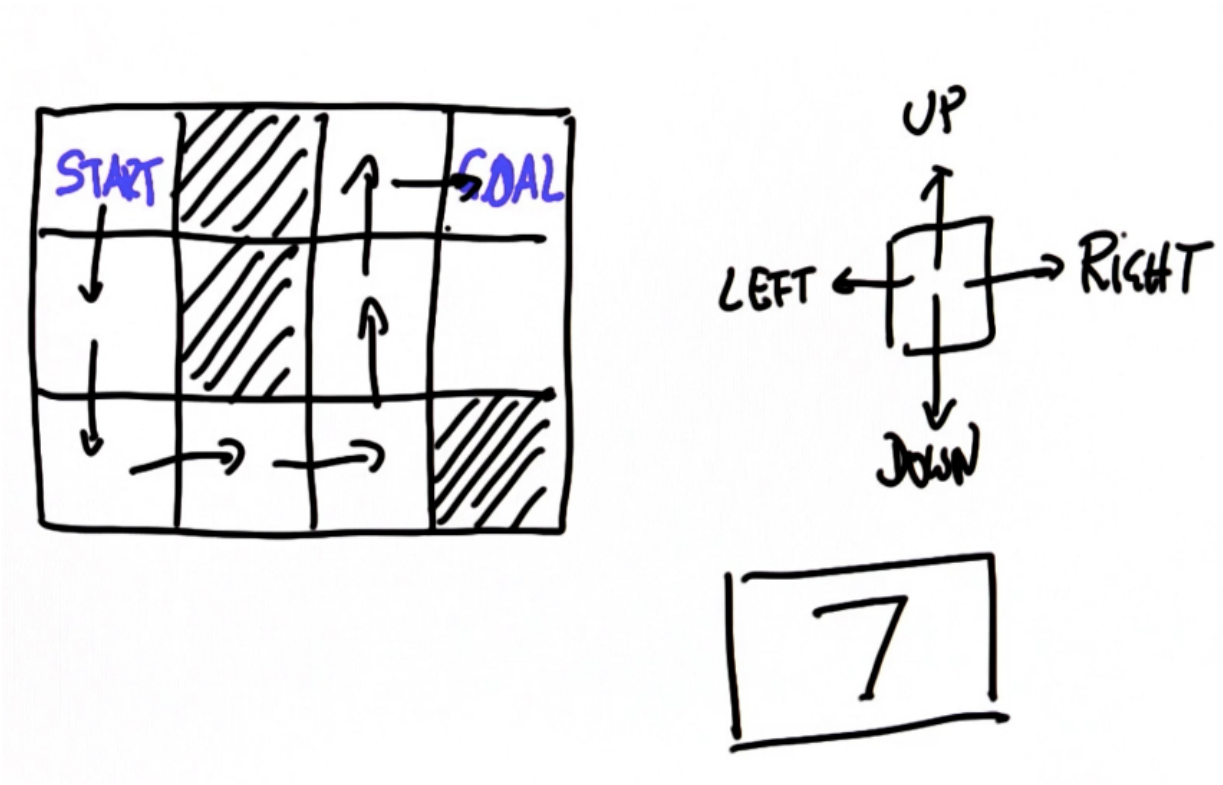


A-4:

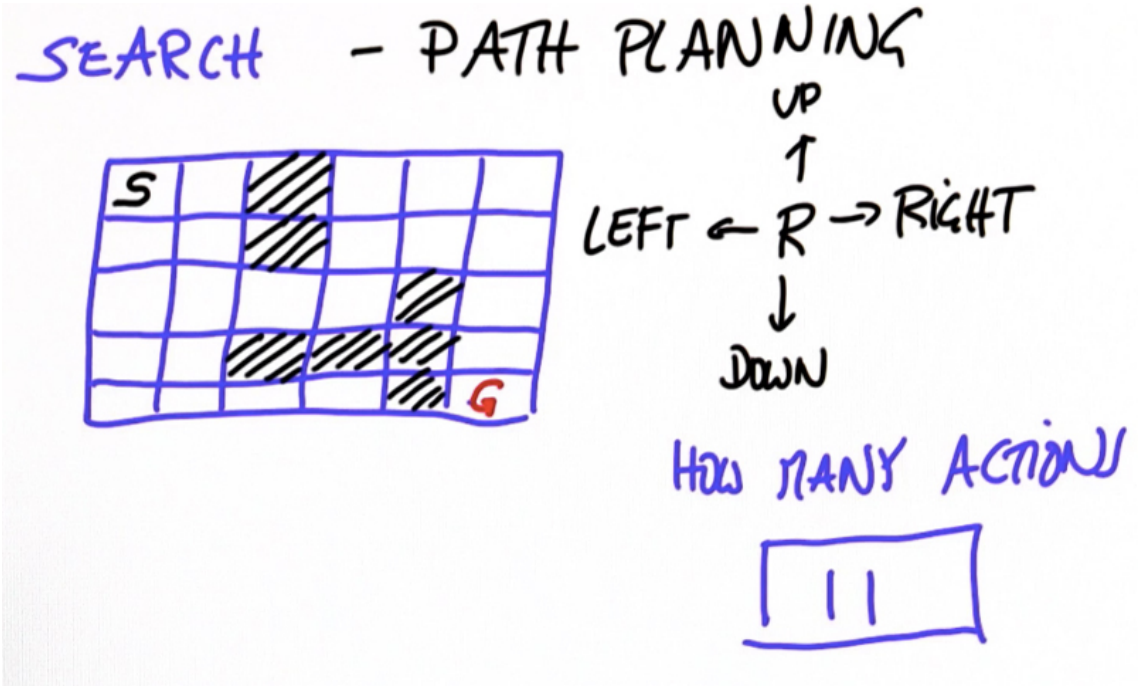


Answer

A-5: Answer (Maze)



A-6: Answer (Maze 2)



A-7: Answer (First search program)

To check the cells once they are expanded, and to not expand them again, initialize a two-dimensional list, called **closed**, the same size as **grid**. Each cell can take one of two values **0** or **1**. You could also use booleans: **True** and **False**. We initialize it with a value of **0**, all cells being open.

```
def search():
closed = [[0 for row in range(len(grid[0]))] for col in range(len(grid))
]
```

Alternatively, it could be initialised like this:

```
closed = [[0] * len(grid[0]) for i in grid]
```

Then, set the value of the starting cell to **1**, since you are expanding it.

```
closed[init[0]][init[1]] = 1
```

Set the initial values for x, y and g and initialize open like this:

```
x = init[0]
y = init[1]
g = 0
open = [[g, x, y]]
```

Next, set two flag values:

```
found = False  # flag that is set when search is complete
resign = False # flag set if we can't find expand
```

The second flag is really important for the case when you can not find a path to a goal after searching all possible locations.

Now repeat the following code while both the **found** and **resign** are **False**, that is until you have either found the goal, or found out that the goal can not be reached. If the open list is empty, it means that you have not found the goal, and have nowhere else to search, so return "fail", else we find the smallest value of **open.**

In the video this was done by sorting the array (**sort()** will sort the list in ascending order), then reversing the array with **reverse()** to make it descending (smallest value at end), and then removing the last value from the array by **pop()** and assigning it to **next**. This gives us the smallest value of **g**, and that is why it's important that the **g** value was first in the triplet values in the **open** list.

There are other ways of doing this more efficiently. For example, you don't need to reverse the array, you can use **open.pop(0)** and that will pop the first item (with index 0) in the list, as opposed to the default last. **pop(0)** in this case of array sorted in ascending order will be the smallest value. You can just do:

```
            open.sort()
            next = open.pop(0)
```

Another, completely different alternative in Python, is to use the **min()** function that returns the minimum value of an iterable (list is an iterable), and then remove the value from the list with **remove()**. So, you can just do:

```
            next = min(open)
            open.remove(next)
```

The interesting thing happens in the case where you have not reached the goal state. For each possible move **delta** first check if you are still within the borders of **grid**. Then check if the node has not yet been expanded and is passable, and if all of these things are true increment **g** by **cost**, and add the cost and coordinates of the node to our **open** list, to be expanded later, and at the end set the value of the current node in **closed** to be **1**, so that it is not expanded again.

```
    while not found and not resign:
        if len(open) == 0:
            resign = True
            return "fail"
        else:
            open.sort()
            open.reverse()
            next = open.pop()
            x = next[1]
            y = next[2]
            g = next[0]

            if x == goal[0] and y == goal[1]:
                found = True
            else:
                for i in range(len(delta)):
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
                    if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                            g2 = g + cost
                            open.append([g2, x2, y2])
                            closed[x2][y2] = 1
    return next
```

A-8: Answer (Expansion Grid)

The modifications you have to make in this case are quite small. You need to initialize a new list, **expand** to same size as the **grid**, but with initial values of **-1**. You can do it in a similar way as with the **open** list. You also need a variable to keep count of expansion steps, called **count** and

initialize it with **0**. At the end of the main procedure set the value of **expand[x][y]** to the value of **count**, and increment **count** by **1**. This way all the grid cells that were explored will have a value of expansion step count, while cells that were not expanded, because the could not be reached, will have a value of **-1**.

```
def search():
    closed = [[0] * len(grid[0]) for i in grid]
    closed[init[0]][init[1]] = 1
    expand = [[-1] * len(grid[0]) for i in grid]
    x = init[0]
    y = init[1]
    g = 0

    open = [[g, x, y]]
    expand[x][y] = g
    count = 0
    found = False  # flag that is set when search is complete
    resign = False # flag set if we can't find expand

    while not found and not resign:
        if len(open) == 0:
            resign = True
        else:
            open.sort()
            open.reverse()
            next = open.pop()
            x = next[1]
            y = next[2]
            g = next[0]
            expand[x][y] = expand_counter
            if x == goal[0] and y == goal[1]:
                found = True
            else:
                for i in range(len(delta)):
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
                    if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                            g2 = g + cost
                            open.append([g2, x2, y2])
                            closed[x2][y2] = 1
                            count += 1 #equivavelent to: count = count +
1
    return expand #Leave this line for grading purposes!
```

A-9: Answer (Print Path)

Here is a possible solution:

Make a list (table) called **action**, the same size as the **grid** and initialized the same way as **expand** to memorize what action it took us to get there. For example, for the goal cell it takes an action to go down to get to it, then the action table for the goal cell will have the index for going down.

It is really easy to program, just add this line after the line **closed[x2][y2] = 1**:

```
action[x2][y2] = i
```

So, for the successor state, **[x2][y2],** add the action **i** that took us there. The tricky part to understand here is why we are using **[x2][y2]** and not **[x][y].** The reason for this is, that **[x][y]** represents the "from" state, and you do not know the correct direction for that cell because you are trying out all the directions (**for i in range(len(delta))**) and expanding them. But in the **[x2][y2]** state you already have expanded, this is the "next" cell, and you know how you got there. If you change out the **action[x2][y2] = i** to be instead**:**

```
action[x2][y2] = delta_name[i]
```

and print out the action list now, you will see:

```
[' ', '>', ' ', '^', '>', '>']
['v', 'v', '>', '>', '>', '>']
['v', 'v', ' ', 'v', ' ', 'v']
['v', 'v', ' ', 'v', ' ', 'v']
['v', 'v', ' ', 'v', ' ', 'v']
```

As you see, this list has direction indices for all the cells we expanded, so it is not what we really want, yet. But you can look at this as a map and see that, since each cell contains directions about how to get there, it is easy to backtrack the correct path, starting from goal node, step by step.

This action table has all the expanded nodes, and that is not what we want. You will therefore, initialize another table, called **policy**, same size as **grid**, filled with blanks **' '**, and set the start of our reverse search to be goal node, and set the value of goal node to be a star**'*'**:

```
policy =  [[' '] * len(grid[0]) for i in grid]
x = goal[0]
y = goal[1]
policy[x][y] = '*'
```

Then, go from the goal state backwards, until you reach the initial start state. Find the previous state by subtracting from our current state the action that took as here. Remember that the action list has exactly that information.

```
while x != init[0] or y != init[1]:
    x2 = x - delta[action[x][y]][0]
    y2 = y - delta[action[x][y]][1]
```

```
        policy[x2][y2] = delta_name[action[x][y]]
        x = x2
        y = y2
```

Backtrack from the goal node to the start position using the actions from the action list to guide you. You can do this by appling the opposite action (by subtracting action from current state), which for "down" happens to be "up", and end up in the cell above. Now, mark the policy field for this state by using the special symbol for the action that you took from this state to get to the goal state. Then, reiterate by setting **x** to **x2** and **y** to **y2**. By doing this you reverse our path step by step and mark it with the direction symbols. Do this until we have reached your initial start position, and then you can return (or print) the **policy** table.

This is the full code for this assignment:

```
    def search():
        closed = [[0] * len(grid[0]) for i in grid]
        closed[init[0]][init[1]] = 1
        action = [[-1] * len(grid[0]) for i in grid]

        x = init[0]
        y = init[1]
        g = 0

        open = [[g, x, y]]

        found = False  # flag that is set when search is complet
        resign = False # flag set if we can't find expand

        while not found and not resign:
            if len(open) == 0:
                resign = True
                return 'fail'
            else:
                open.sort()
                open.reverse()
                next = open.pop()
                x = next[1]
                y = next[2]
                g = next[0]

                if x == goal[0] and y == goal[1]:
                    found = True
                else:
                    for i in range(len(delta)):
                        x2 = x + delta[i][0]
                        y2 = y + delta[i][1]
                        if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
    len(grid[0]):
```

```
                    if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                        g2 = g + cost
                        open.append([g2, x2, y2])
                        closed[x2][y2] = 1
                        action[x2][y2] = i
    policy =  [[' '] * len(grid[0]) for i in grid]
    x = goal[0]
    y = goal[1]
    policy[x][y] = '*'
    while x != init[0] or y != init[1]:
        x2 = x - delta[action[x][y]][0]
        y2 = y - delta[action[x][y]][1]
        policy[x2][y2] = delta_name[action[x][y]]
        x = x2
        y = y2

    for row in policy:
        print row
    return policy # make sure you return the shortest path.
```

A-10: Answer (Implementing A Star)

It turns out that the actual implementation is really minimal compared to what you have already implemented. And with this modification you have implemented A*, which is one of the most powerful search algorithms that are in use at the present day to drive self-driving cars through unstructured environments.

The very first thing you should do is to expand elements of the open list by including not just **g**, but also **f**. In the video also **h** is included, but it isn't necessary.

```
g = 0
h = heuristic[x][y]
f = g + h
open = [[f, g, h, x, y]]
```

So, the **open** list is now two elements larger, because it has two extra elements **f** and **h**, where **h** is the heuristic function and **f** is the sum of **g** and **h**. The reason to put **f** first is for the search trick to work allowing you to sort according to **f** when you sort the list. This will make sure that the element you expand will be the one with the lowest **f** value, not the lowest **g** value.
Now, when you go and expand the node, you need to modify the indexes a little bit, because the open list was changed:

```
        open.sort()
        open.reverse()
        next = open.pop()
        x = next[3]
        y = next[4]
        g = next[1]
```

x is element number **3**, which is technically 4th element, but we start indexing from **0**. **y** is element number **4** and **g** is element number **1**. You don't need to implement **f** and **h** here, because you will calculate them couple of lines of code later.
Down where you expand the node and go trough all possible actions you can take from there, you need to add couple lines:

```
                if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
    len(grid[0]):
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                            g2 = g + cost
                            h2 = heuristic[x2][y2]
                            f2 = g2 + h2
                            open.append([f2, g2, h2, x2, y2])
                            closed[x2][y2] = 1
```

Calculate **h2** from heuristic function, calculate **f2** by suming **g2** and **h2**, and then append the five values to the **open** list. And that is all there is to implementing A* algorithm! All you have done is changing the logic according to which you remove nodes from the stack to pick the one that has the minimum f value.
This is the final code:

```
    def search():
    closed = [[0 for row in range(len(grid[0]))] for col in range(len(grid))
    ]
        closed[init[0]][init[1]] = 1

    expand = [[-1 for row in range(len(grid[0]))] for col in range(len(grid)
    )]
    action = [[-1 for row in range(len(grid[0]))] for col in range(len(grid)
    )]

        x = init[0]
        y = init[1]
        g = 0
        h = heuristic[x][y]
        f = g + h
        open = [[f, g, h, x, y]]

        found = False  # flag that is set when search is complet
        resign = False # flag set if we can't find expand
        count = 0

        while not found and not resign:
            if len(open) == 0:
                resign = True
            else:
                open.sort()
```

```
            open.reverse()
            next = open.pop()

            x = next[3]
            y = next[4]
            g = next[1]
            expand[x][y] = count
            count += 1

            if x == goal[0] and y == goal[1]:
                found = True
            else:
                for i in range(len(delta)):
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
                    if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <
len(grid[0]):
                        if closed[x2][y2] == 0 and grid[x2][y2] == 0:
                            g2 = g + cost
                            h2 = heuristic[x2][y2]
                            f2 = g2 + h2
                            open.append([f2, g2, h2, x2, y2])
                            closed[x2][y2] = 1
    for i in range(len(expand)):
        print expand[i]
    return expand #Leave this line for grading purposes!
```

Check out some examples of this code in action!

Change the grid to be like this, with an opening at top:

```
grid = [[0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 1, 0]]
```

It turns out in this case A* is not so efficient. In the area starting from **[0,2]** it has no preferences to go either way until it finally finds the goal node.

```
[0,   1,  3,  5,  8, 12]
[2,  -1,  6,  9, 13, 16]
[4,  -1, 10, 14, 17, 20]
[7,  -1, 15, 18, 21, 23]
[11, -1, 19, 22, -1, 24]
```

That however changes when you put a big obstacle horizontally like this:

```
grid = [[0, 0, 0, 0, 0, 0],
```

```
    [0, 1, 1, 1, 1, 0],
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 1, 0]]
```

It is very interesting to see that A* can not decide initially if the horizontal path is better or the vertical, and alternatively pops nodes from either of these. But the moment it expands the node at **`[0,5]`** the same trick applies as before. It does not expand anything in the center, but goes straight to the goal (table to the left), as opposed to the original search (table to the right) that expands a bit to the middle:

```
[0,  1,  3,  5,  7,  9]
[2, -1, -1, -1, -1, 10]
[4, -1, -1, -1, -1, 11]
[6, -1, -1, -1, -1, 12]
[8, -1, -1, -1, -1, 13]
```

```
[0,  1,  3,  5,  7,  9]
[2, -1, -1, -1, -1, 10]
[4, -1, -1, 14, 12, 11]
[6, -1, -1, -1, 15, 13]
[8, -1, -1, -1, -1, 16]
```

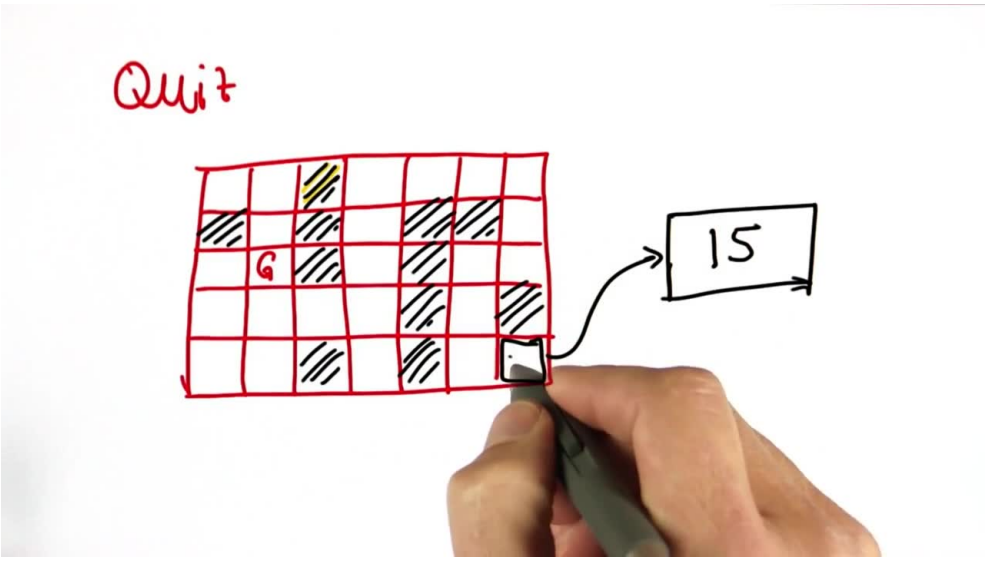
As you can see, the original search expanded more nodes than A* search. This may look very insignificant, but if you get to very large environments, it can make a huge difference, especially if there is huge dead end somewhere that can't reach the goal. Then A* performs much, much more efficiently than the simple search.

Note: you can easily test different grids with your original search by using an empty heuristic function that can be easily set up like this, initializing **`heuristic`** table with all **`0`** values:

```
heuristic = [[0] * len(grid[0]) for row in grid]
```

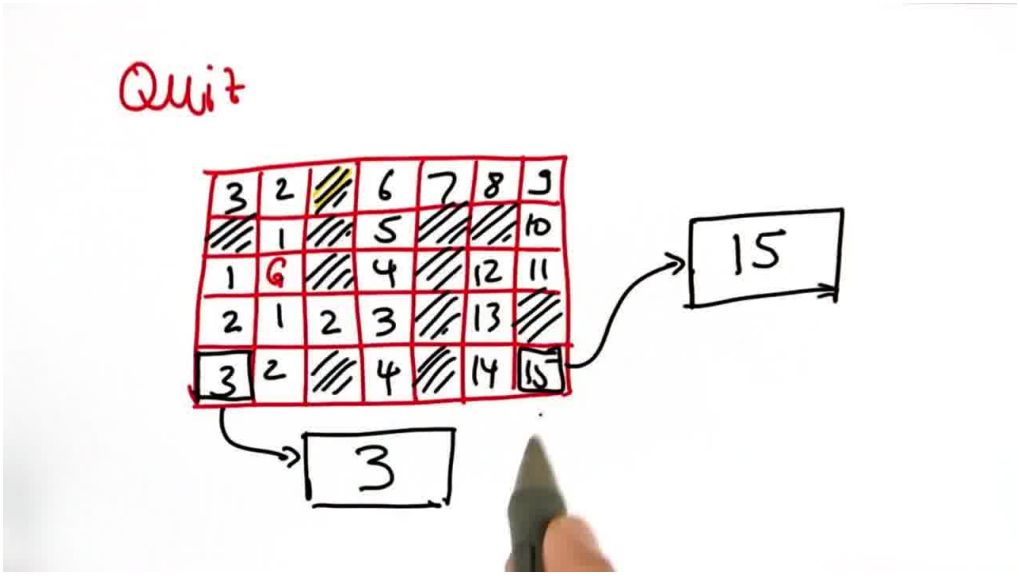
A-11: Answer

It takes 15 steps on the shortest path between the goal state and the initial state.



A-12: Answer

It takes three steps in this case. At this point, you should be able to see how the value function propagates throughout the entire grid.



A-13: Answer (Value Program)

This is a relatively straightforward implementation of the value program. Initialize table **`value`** the same size as the world, with **99** everywhere. This has to be value that is large enough and does not conflict with any actual value. And you should update the value table, we don't know how many times, but as long as there is something to change, you update it. Therefore, there is a flag **`change`** initialized to **`True`**. Inside loop set it to **`False`**, and then set it to **`True`** again only when an actual update in the table **`value`** happens.

Now go trough all the grid cells in a set order. First, check for the goal cell and see if it's larger than 0. If that is the case, set it to 0 and set the flag change to True because you just changed something!

If it's not a goal cell, then you go through the update function. Go through all the actions and project the next state after taking action, then you check if after action you are still in a valid cell inside the grid that is not an obstacle. Then assign the new value **`v2`**, which is the value of the future grid cell plus cost of step, that happens to be one. If that **`v2`** is better than the value you have already, which means it's smaller, assign the new value to the original grid cell and set the change to **`True`**.

This implementation is not very efficient, but it computes the values correctly.

It is interesting to see what happens if you put an obstacle so that part of the grid is not reachable. The cut of parts of grid will retain the value 99.

This is the full code of this solution:

```
def compute_value():
value = [[99 for row in range(len(grid[0]))] for col in range(len(grid))
]
    change = True
    while change:
        change = False
        for x in range(len(grid)):
            for y in range(len(grid[0])):
                if goal[0] == x and goal[1] == y:
                    if value[x][y] > 0:
                        value[x][y] = 0
                        change = True
                elif grid[x][y] == 0:
                    for a in range(len(delta)):
                        x2 = x + delta[a][0]
                        y2 = y + delta[a][1]
                        if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2
< len(grid[0]):
                            v2 = value[x2][y2] + cost_step
                            if v2 < value[x][y]:
                                change = True
                                value[x][y] =  v2
```

A-14: Answer (Optimal Policy)

It turns out that finding the optimal solution through dynamic programming is even easier than A*.

Start by defining a policy matrix the same size as your grid and initialize it with spaces:

```
policy = [[‘ ‘ for row in range(len(grid[0]))] for col in
range(len(grid))]
```

In the loop there are two steps. First, check to see if each cell is a goal state and mark the policy with the * character.

If a cell is not a goal state we look at all possible directions by looping over the available actions. If an action results in an improved value, assign the corresponding directional character that led to the improved value.

```
for x in range(len(grid)):
    for y in range(len(grid[0])):

        if goal[0] == x and goal[1] == y:
            if value[x][y] > 0:
                value[x][y] = 0
                policy[x][y] = '*'
                change = True

        elif grid[x][y] == 0:
            for a in range(len(delta)):
                x2 = x + delta[a][0]
                y2 = y + delta[a][1]

                if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 <
len(grid[0]) and grid[x2][y2] == 0:

                    v2 = value[x2][y2] + cost_step

                    if v2 < value[x][y]:
                        change = True
                        value[x][y] = v2
                        policy[x][y] = delta_name[a]

#for i in range(len(value)):
#print value[i]
for i in range(len(value)):
    print policy[i]
```

A-15: Answer (Left Turn Policy)

Here is the solution. The **value** function is initialized in 3D with a lot of **999**. The policy function is similar, in 3D. And then there is a function called **policy2D** which will be the one that is returned and printed out at the end, and that is in 2D.

The update function is exactly the same as before for dynamic programming. While **change** exists, go through all existing **x,y** and **orientation** of which there are four, so there is now a deeper loop.

If this is the goal location, mark it as the goal location. otherwise check if the cell is navigable, and if it is, here comes the tricky part. Go through the three different actions and when you take the **i-**th action add the corresponding orientation change to your orientation and take modulo four --

a cyclic buffer. By taking modulo 4 (% 4) you are ensuring that the index of the orientation list is always within range [0,3]. Then obtain the corresponding new motion model to obtain new **x2** and **y2**. This is a model of a car that first steers and then moves. If you arrive in a valid grid cell, update the **value** and **policy** tables similarly as before, only taking orientation into consideration.

But you need to return a 2-dimensional table, but you just got a 3-dimensional one!

Set the **x,y** and **orientation** values to the initial state **init**. Now all you do is run the policy.
For the first state (starting position) copy over the policy from the 3D table to 2D. Then, while you haven't reached the goal state yet, as indicated by checking for the star '*', check for the orientation and according to that, apply forward motion to update the new x and y coordinates that correspond after the motion coordinates and the orientation to be o2. Finally, copy the movement symbol from the 3D table directly to the 2D table.

The key insight here is to go from the 3-dimensional full policy to the 2-dimensional, you had to run the policy.

Full solution code:

```
def optimum_policy2D():
    value = [[[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[999 for row in range(len(grid[0]))] for col in
range(len(grid))]]
    policy = [[[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))],
             [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]]
    policy2D = [[' ' for row in range(len(grid[0]))] for col in
range(len(grid))]
    change = True
    while change:
        change = False
        for x in range(len(grid)):
```

```
            for y in range(len(grid[0])):
                for orientation in range(4):
                    if goal[0] == x and goal[1] == y:
                        if value[orientation][x][y] > 0:
                            value[orientation][x][y] = 0
                            policy[orientation][x][y] = '*'
                            change = True
                    elif grid[x][y] == 0:
                        for i in range(3):
                            o2 = (orientation + action[i]) % 4
                            x2 = x + forward[o2][0]
                            y2 = y + forward[o2][1]
                            if x2 >= 0 and x2 < len(grid) and y2 >= 0
and y2 < len(grid[0]) and grid[x2][y2] == 0:

                                v2 = value[o2][x2][y2] + cost[i]
                                if v2 < value[orientation][x][y]:
                                    change = True
                                    value[orientation][x][y] = v2
                                    policy[orientation][x][y] =
action_name[i]
    x = init[0]
    y = init[1]
    orientation = init[2]
    policy2D[x][y] = policy[orientation][x][y]
    while policy[orientation][x][y] != '*':
        if policy[orientation][x][y] == '#':
            o2 =  orientation
        elif policy[orientation][x][y] == 'R':
            o2 = (orientation - 1) % 4
        elif policy[orientation][x][y] == 'L':
            o2 = (orientation + 1) % 4
        x = x + forward[o2][0]
        y = y + forward[o2][1]
        orientation = o2
        policy2D[x][y] = policy[orientation][x][y]
    return policy2D # Make sure your function returns the expected grid.
```