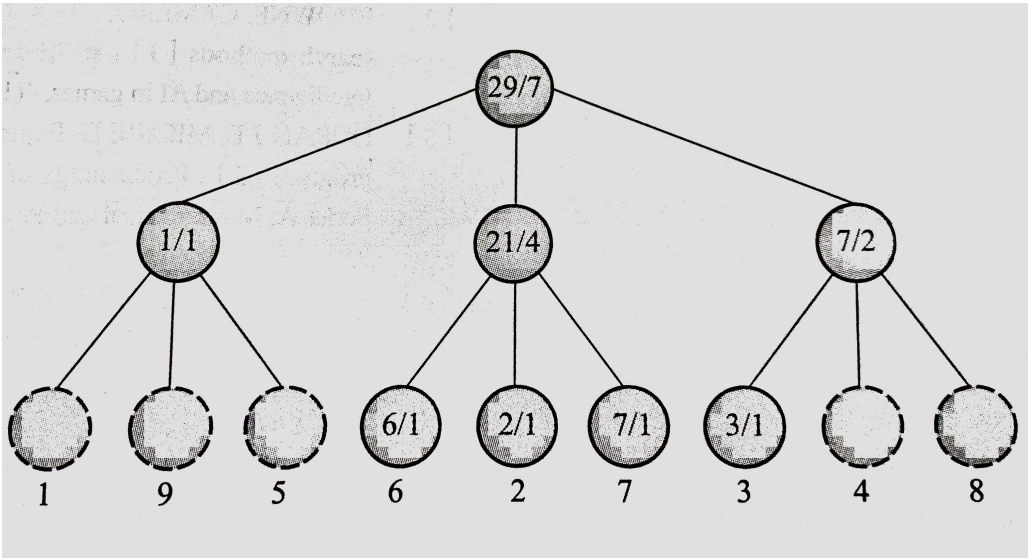
Question Answering (20 points)

The following figure shows an example of Monte Carlo Tree Search, where each leaf node is annotated with the utility value. To maximize the utility, here Monte Carlo Tree Search is applied to return a planning path. Suppose that after several steps, the current state of the MCTS search tree is also shown in the following figure, where the numbers within a node represent "estimated value / the number of visits" and those nodes encircled by dashed lines are not expanded yet.

- (5 points) Provided that the hyperparameter C=1 in the UCB bound, please show the path selected by MCTS.
- (5 points) Please continue the procedures of selection, expansion, roll out, and backpropagation, and visualize the state of the search tree afterwards. You can set the value of C for exploration.
- (10 points) Try more iterations of the four procedures of MCTS, and see if the leaf node with the largest utility (i.e., 9) can be returned. If not, please give your suggestions on how to modify the algorithm for higher efficiency.



Programming (80 points)

In this project, your drone agent will find paths through an urban environment, both to reach a particular location and to cover a set of specified locations along the path. You will build general search algorithms and apply them to this path planning problem.

Instructions before coding:

1. Clone the project repository from https://github.com/udacity/FCND-Motion-Planning
2. Download the simulator from https://github.com/udacity/FCND-Simulator-Releases/releases (selecting the simulator according to your operating system)
3. Set up your Python Environment using Anaconda. If you are not familiar with Anaconda, please kindly refer to this site https://github.com/udacity/FCND-Term1-Starter-Kit.
   3.1 If you install the package **udacidrone** by Anaconda, you can ignore the next step in 3.2.

3.2 If you have problems in installing the package **udacidrone** by Anaconda, you can 1) download it from https://github.com/udacity/udacidrone, and 2) put it into the file **FCND-Motion-Planning3** making sure that this package can be called by **motion_planning.py.** 3) When you run the project **motion_planning**, you may be reminded some packages are missing. The solution is to install these required packages following this instruction: **pip --upgrade package's name** (you can refer to https://linuxize.com/post/how-to-install-pip-on-ubuntu-18.04/ to learn how to use **pip**, if you do not know about this.)

4. After finishing the setup of your Python Environment, you can test if this project runs well in your computer. First, open the simulator and choose MOTION PLANNING shown in Fig. 1. Second, run the project motion_planning, and you will see this drone flies automatically as shown in Fig. 2.

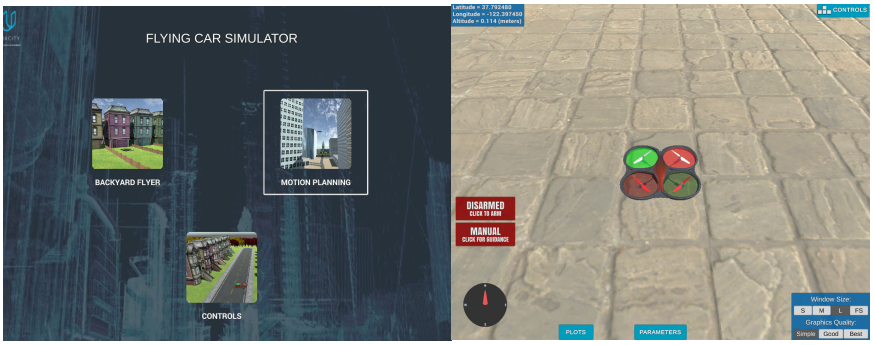


Figure 1 Click MOTION PLANNING and you will see a drone.

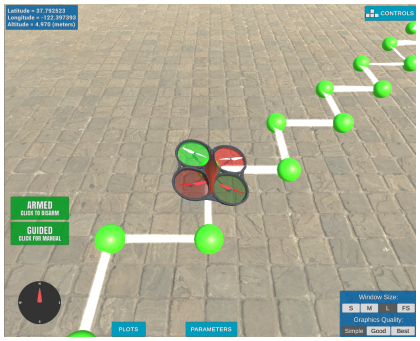


Figure 2 The drone flies automatically.

5. Write your planner with A* search algorithm based on **motion_planning.py**. If it is difficult for you, you can refer to the file **Untitled.ipynb** which you can open via github or via the link https://github.com/udacity/FCND-Motion-Planning/blob/master/Untitled.ipynb.
6. After you finish your planner, you can test it. Through adjusting the values of parameters grid_start and grid_goal, you can set up the start point and end point for your drone. **Reminder**: **if these points is in the obstacle map recorded by the file colliders.csv, the search algorithm may take a very long time and fail to output the path. In this situation, you should better reset these parameters and run the updated program again**.

Files you'll edit:

1. **motion_planning.py**: this function leverages some extra functions in planning_utils.py to plan the path for the done.
2. **planning_utils.py**: where all of your search algorithms will reside.

Files to edit and submit:

You are expected to submit the two files above with your code and comments. Please do not change the other files in this distribution or submit any of our original files other than these files.

Question 1 (20 points): Implementing the iterative deepening A* search algorithm

You are expected to write an iterative deepening A* search algorithm in planning_utils.py named iterative_astar(grid, h, start, goal) to help the drone plan routes. Procedure for the iterative deepening A* search algorithm can be found in the lecture slides.

Question 2 (20 points): Implementing the uniform cost search algorithm

You are expected to write a uniform cost search algorithm in planning_utils.py named ucs(grid, h, start, goal) to help the drone plan routes. Procedure for the uniform cost search algorithm can be found in the lecture slides. Note that you should first of all design and implement the cost function.

Question 3 (20 points): Implementing different heuristics for A* and the A* search for traversing 3 fixed points

In the current A* version in planning_utils.py, the Manhatten distance is used as the heuristic. You are encouraged to propose one valid heuristic, implement that, and see how the planned routes change.

You are also required to set three fixed points in motion_planning.py which the drone has to traverse before reaching to the destination. Building on this, you re-implement the A* search algorithm in order to go through the 3 points.

Question 4 (20 points): Implementing the breadth-first search algorithm with graph search

You are expected to shift the representation of the state space from grids in the previous questions to a Voronoi graph. Please install the package scipy to access the Voronoi and Voronoi_plot_2d functions. The following code snippet lists how you can build a graph state space of the map given the obstacle data.

```
filename = 'colliders.csv'
data = np.loadtxt(filename, delimiter=',', dtype='Float64', skiprows=2)

# Here you'll modify the `create_grid()` method from a previous exercise
# In this new function you'll record obstacle centres and
# create a Voronoi graph around those points
def create_grid_and_edges(data, drone_altitude, safety_distance):
    """
    Returns a grid representation of a 2D configuration space
```

```
    along with Voronoi graph edges given obstacle data and the
    drone's altitude.
    """

    # minimum and maximum north coordinates
    north_min = np.floor(np.min(data[:, 0] - data[:, 3]))
    north_max = np.ceil(np.max(data[:, 0] + data[:, 3]))

    # minimum and maximum east coordinates
    east_min = np.floor(np.min(data[:, 1] - data[:, 4]))
    east_max = np.ceil(np.max(data[:, 1] + data[:, 4]))

    # given the minimum and maximum coordinates we can
    # calculate the size of the grid.
    north_size = int(np.ceil((north_max - north_min)))
    east_size = int(np.ceil((east_max - east_min)))

    # Initialize an empty grid
    grid = np.zeros((north_size, east_size))
    # Center offset for grid
    north_min_center = np.min(data[:, 0])
    east_min_center = np.min(data[:, 1])

    # Define a list to hold Voronoi points
    points = []
    # Populate the grid with obstacles
    for i in range(data.shape[0]):
        north, east, alt, d_north, d_east, d_alt = data[i, :]

        if alt + d_alt + safety_distance > drone_altitude:
            obstacle = [
                int(north - d_north - safety_distance - north_min_center),
                int(north + d_north + safety_distance - north_min_center),
                int(east - d_east - safety_distance - east_min_center),
                int(east + d_east + safety_distance - east_min_center),
            ]
            grid[obstacle[0]:obstacle[1], obstacle[2]:obstacle[3]] = 1

            # add center of obstacles to points list
            points.append([north - north_min, east - east_min])

    # TODO: create a voronoi graph based on
    # location of obstacle centres
    graph = Voronoi(points)
    # TODO: check each edge from graph.ridge_vertices for collision
    edges = []
    for edge in graph.ridge_vertices:
        point1 = graph.vertices[edge[0]]
        point2 = graph.vertices[edge[1]]

        cells = list(bresenham(int(point1[0]), int(point1[1]), int(point2
[0]), int(point2[1])))
        infeasible = False
```

```
        for cell in cells:
            if np.amin(cell) < 0 or cell[0] >= grid.shape[0] or cell[1] >=
 grid.shape[1]:
                infeasible = True
                break
            if grid[cell[0], cell[1]] == 1:
                infeasible = True
                break
        if infeasible == False:
            point1 = (point1[0], point1[1])
            point2 = (point2[0], point2[1])
            edges.append((point1,point2))

    return grid, edges

# Define a flying altitude (feel free to change this)
drone_altitude = 5
safety_distance = 3
grid, edges = create_grid_and_edges(data, drone_altitude, safety_distance)
print('Found %5d edges' % len(edges))


# equivalent to
# plt.imshow(np.flip(grid, 0))
# Plot it up!
plt.imshow(grid, origin='lower', cmap='Greys')

# Stepping through each edge
for e in edges:
    p1 = e[0]
    p2 = e[1]
    plt.plot([p1[1], p2[1]], [p1[0], p2[0]], 'b-')

plt.xlabel('EAST')
plt.ylabel('NORTH')
plt.show()
```

Now we have a graph with grids and edges. The next step is to use the package of NetworkX to import and manipulate the graph. You are strongly encouraged to read the documentation for Networkx here https://networkx.org/ , while the key steps include the following three.

a) Create a graph by
   G = nx.Graph()
b) Add an edge by
   p1 = (9, 1.2)
   p2 = (20, 30)
   G = nx.add_edge(p1, p2)
c) Add an edge with a weight by
   p1 = (10, 2.2)
   p2 = (50, 40)
   dist = LA.norm(np.array(p2) - np.array(p1))
   G = nx.add_edge(p1, p2, weight=dist)

Based on the graph G, now you are ready to implement the breadth-first search algorithm to search the graph and return the path.