Dijkstra's Algorithm - Pseudo code

Functions:

Distance(wpt1, wpt2) → calc. distance from one node/wpt to another

Collision Check(obstacle list, grid bounds, current location)
- xy locations
- grid bounds (could be time, energy, etc) in xy
- → Return True/False depending on if current location is a valid location

Calculate Node Index(current node, grid information)
- → Returns node number from previously defined numbering/counting scheme
- grid bounds, step size

Dijkstra(start location, goal location, grid info, obstacle list)
- → Runs the Dijkstra algorithm, returns the desired path/waypoints

*Optional Node Neighbor Cost(current node, grid info, obstacle list)
- → Calculates cost of moving to all neighbor nodes
- → updates unvisited node dictionary as appropriate

New Python Items:

Classes - many use, but in our case its an easy way to create a custom variable structure for storing our nodes

Example:

```
class data:

    def __init__(self, input1, input2, cheese):
        self.x1 = input1
        self.y1 = input2
        self.quality = cheese
```

- whatever variable name you would like
- variables you want to pass into class
- Name you use to reference

then usage of class:

```
temp = data(1.0, 3.0, 4)
Print temp.quality → 4
```

Dictionaries:

Lists/arrays you are used to are indexed by numbers.
And ~~your~~ if you want to store dummy[3] = 5, then dummy[0] - [2] must be created in memory (even if 0 value).

A dictionary uses keys for indexing. Keys can be strings, numbers, ...
And importantly, because the indexing is with keys, its not dependent on having a monotonically increasing index.

We will use a dict. to store our visited and unvisited nodes. Then we can search through the dict. as needed.

Usage

```
unvisited_nodes = dict()          → initialize the dictionary
current_node = node(0, 0, 0, -1)  → a node class to store
node_index = calc_index(current_node, ...)   x, y coords, cost, parent index
unvisited_nodes[node_index] = current_node
                                  ↳ get node index of current node
                                  ↳ store current node at key index node_index
```

Minimum: Want to search through unvisited_nodes to find next node with lowest cost to visit

```
min(example, key = lambda X: X['key'])
```

→ set search of keys

(dictionary to search

For our use:

```
index of node to visit = min(unvisited_nodes, key = lambda X: unvisited_nodes[X].cost)
```

what to return

what I'm trying to find min of.

Dijkstra Function "Pseudo code"

- Initialize visited and unvisited nodes dictionaries
- Current Node = starting point, no travel cost (=0), Parent node index = -1
- Calculate location index of current node
  (based upon where node exists in grid, need to create numbering scheme) → calc_index function
  When finding final path, -1 is easy way to know we have found start point
- Put current node in unvisited nodes dictionary

While current node not equal to goal location

  Curr_node index = node number of minimum cost in unvisited nodes

  current node = unvisited nodes [curr_node index]

  put/store current node in visited nodes

  delete current node from unvisited nodes → Don't want to visit node again

  Check if current node is goal location, if so break out of loop.

* Can make this a function call

  With current node, need to calculate costs to travel to all neighbors

  From current node, & connected nodes. One approach is to cycle through all of them. Will have to check if any neighbor is an obstacle, out of bounds, or the node itself

  for i in range (-grid size to grid size) → pseudo code, watch how this steps/starts (include end?)
    for j in range (-grid size to grid size)
      - Calculate cost → distance for us to go from current node to current +(i, j)
      - temp node = node (x, y, cost, current node index) ← this is the index of temp node parent

- Calculate temp node location index → used for storing/looking up node
- Check if temp node is valid
  - → Not same location as current node
  - → Not in obstacle
  - → Not out of grid boundary
- if check is good, see if temp node exists in unvisited nodes dictionary → by seeing if node index exists in dict
  - → If temp node <u>does</u> exist in unvisited nodes <u>and</u> cost from temp node is less than node in unvisited nodes
    - → then update cost and parent node index in unvisited nodes dictionary
  - → If temp node <u>does not</u> exist in unvisited nodes
    - → then node is new. Stick/store temp node in unvisited nodes at index => unvisited_nodes[temp Node loc index] = temp node

Loop Around → After this the goal has been found

Time to find the path from <u>goal</u> to <u>start</u>

- Initialize path x ; y as a list or array with goal coords
- Get node location index of goal location
- While Parent Node Index ≠ -1 → Remember this is the start
{
- Add visited nodes [Parent Node Index] to path x ; y
- New parent node index = parent of visited_node[parent node index] ↳ = visited_node [parent node index] . parent or what ever you named it

Loop around
}

Plot path (use Mat Plot Lib)