CSCI 3202, Spring 2022

Homework 2

Due: Wednesday, February 15 at 9:00 pm

Your name: Julia Troni

In [1]:
```
import random
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from collections import deque
import heapq

# Sets random seeds for reproducibility
SEED=42
random.seed(SEED)
np.random.seed(SEED)
```

Problem 1

Discrete Robot Path Planning

A robotics group at CU needs some help designing a path planning algorithm that can navigate around the engineering center. To get started they have designed two test environments for you to implement breadth-first, depth-first, and uniform-cost search.

- In the first environment, every movement to an adjacent cell has a cost of 1. The first environment will be represented by the `edge_weights_1` dictionary.
- In the second enviornment, traveling to adjacent cells has a random int cost between 1 and 100. The second environment will be represented by the `edge_weights_2` dictionary.
- Both setups have the same obstacles, free space, and goal. The code below creates and gives a visual representation of the robot's environment.
- The first figure shows the environment itself with free spaces, barriers, robot start, and goal point.

- The second image shows an example of what a path might look like as the robot moves through the environment.

Color representation

yellow = obstacle

teal = free space

pink = goal

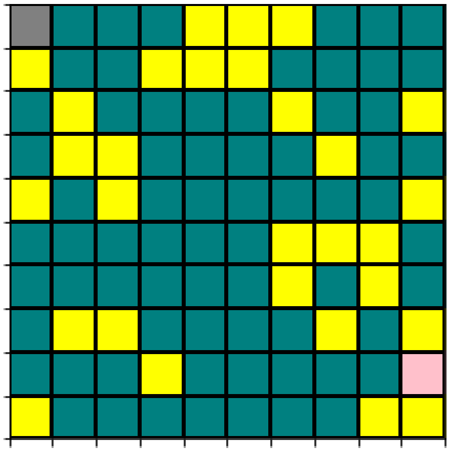
grey = robot start

red = part of the path

In [2]:

```
# Given code
def show_path(env, path):
    """Generates a matplotlib visual of thethe path in the env grid."""
    env_copy = np.copy(env)
    for cell in path:
        if(env_copy[cell[0]][cell[1]] >= 5 and env_copy[cell[0]][cell[1]] < 20):
            env_copy[cell[0]][cell[1]] = 41
    show_env(env_copy)


def show_env(env):
    """Generates a matplotlib visual of the env grid, where colors represent
    the start node (grey), end node (pink), an obstacle (yellow),
    an open node (teal), or part of the path found by a search algorithm (red)."""
    cmap = colors.ListedColormap(['yellow', 'teal', 'pink', 'grey', 'red'])
    bounds = [0,5,20,30,40,41]
    norm = colors.BoundaryNorm(bounds, cmap.N)

    fig, ax = plt.subplots()
    ax.imshow(env, cmap=cmap, norm=norm)
    ax.grid(which='major', axis='both', linestyle='-', color='k', linewidth=2)
    ax.set_xticks(np.arange(-.5, 10, 1));
    ax.set_yticks(np.arange(-.5, 10, 1));
    ax.xaxis.set_ticklabels([])
    ax.yaxis.set_ticklabels([])

    plt.show()


env = np.random.rand(10, 10) * 20
#set robot start position, grey color spot
start = (0, 0)
env[start[0]][start[1]] = 31
#set goal position pink square
goal = (8, 9)
env[goal[0]][goal[1]] = 21
# show the original graph
show_env(env)

# show an example path in the original graph, not a valid path
```

```
# example_path = [(1,1), (2,2), (3,3), (4,4), (5,4), (6,4), (7,4), (8,4), (8,5),(8,6),(
# show_path(env, example_path)
```



Graph Representation

We represent the graph above as an adjacency dict in the following code. You can see what edges any cell has by indexing into the two dicts:

- `edge_weights_1`
- `edge_weights_2`

For indexing, the top left of the graph is (row=0, col=0). Row values increase downward and column values increase to the right.

So for example, if you wanted to look at the pink cell's (the goal location) connections you can call `print(edge_weights_2[(8,9)])`

In [3]:

```
# create dictionary
edge_weights_1 = {}
edge_weights_2 = {}

# Builds both graphs.
for row, row_vals in enumerate(env):
    for col, val in enumerate(env[row]):
        # create dictionary
        edge_weights_1[(row, col)] = {}
        edge_weights_2[(row, col)] = {}
        #set all 6 direction options in edge_wights
        # 1) up
        if(row > 0):
            edge_weights_1[(row, col)][(row-1, col)] = 1
            edge_weights_2[(row, col)][(row-1, col)] = np.random.randint(101)
            if(col > 0):
                edge_weights_1[(row, col)][(row-1, col-1)] = 1
                edge_weights_2[(row, col)][(row-1, col-1)] = np.random.randint(101)
            if(col < 9):
                edge_weights_1[(row, col)][(row-1, col+1)] = 1
                edge_weights_2[(row, col)][(row-1, col+1)] = np.random.randint(101)
        #2) left
        if(col > 0):
```

```
            edge_weights_1[(row, col)][(row, col-1)] = 1
            edge_weights_2[(row, col)][(row, col-1)] = np.random.randint(101)
            if(row < 9):
                edge_weights_1[(row, col)][(row+1, col-1)] = 1
                edge_weights_2[(row, col)][(row+1, col-1)] = np.random.randint(101)
        #3) down
        if(row < 9):
                edge_weights_1[(row, col)][(row+1, col)] = 1
                edge_weights_2[(row, col)][(row+1, col)] = np.random.randint(101)
                if(col < 9):
                    edge_weights_1[(row, col)][(row+1, col+1)] = 1
                    edge_weights_2[(row, col)][(row+1, col+1)] = np.random.randint(101)
        #) right
        if(col < 9):
                edge_weights_1[(row, col)][(row, col+1)] = 1
                edge_weights_2[(row, col)][(row, col+1)] = np.random.randint(101)

for first_node in list(edge_weights_2.keys()):
    for second_node in list(edge_weights_2[first_node].keys()):

        # if first_node is yellow (an obstacle) the connection going both ways should b
        if(env[first_node[0]][[first_node[1]]] < 5):
            edge_weights_2[first_node].pop(second_node)
            edge_weights_2[second_node].pop(first_node)
            edge_weights_1[first_node].pop(second_node)
            edge_weights_1[second_node].pop(first_node)
        # if there is a connection, make sure both edges are the same
        else:
            w1 = edge_weights_2[first_node][second_node]
            w2 = edge_weights_2[second_node][first_node]
            if(w1 != w2):
                edge_weights_2[first_node][second_node] = edge_weights_2[second_node][f


# These represent the same location
print('goal:    ', edge_weights_2[goal])
print('(8, 9): ', edge_weights_2[(8,9)])
```

```
goal:    {(7, 8): 46, (8, 8): 24}
(8, 9):  {(7, 8): 46, (8, 8): 24}
```

Useful helper routines for searching

In [4]:

```
def path(previous, s):
    """
    previous (Dict): Dictionary chaining together the predecessor state that led to eac
        `s` will be None for the initial state.
        otherwise, starts from the last state `s` and recursively traces `previous` bac
        constructing a list of states visited as we go.
    """
    if s is None:
        return []
    else:
        return path(previous, previous[s])+[s]


def pathcost(path, step_costs):
    """
    Adds up the step costs along a path, which is assumed to be a list output
```

```
    from the `path` function above.
    """
    cost = 0
    for s in range(len(path)-1):
        cost += step_costs[path[s]][path[s+1]]
    return cost
```

In [5]:

```
def check_map(step_costs):
    """Checks if all the path costs are at least symmetric."""
    check_states = []
    for state1 in step_costs.keys():
        for state2 in step_costs[state1].keys():
            uh_oh = step_costs[state2][state1]!=step_costs[state1][state2]
            if uh_oh:
                print('Check the costs between states {} and {}'.format(state1,state2))
                check_states.append([state1,state2])
    if len(check_states)==0:
        print('all okay! (symmetric at least)')
    return check_states
```

(1a)

Breadth-first search

Implement a function **breadth_first(start, goal, state_graph, return_cost)** to search the state space defined by the **state_graph** using breadth-first search:

- **start** (Tuple[int, int]): initial state (e.g., '(0,0)' or `start`)
- **end** (Tuple[int, int]): goal state (e.g., '(8,9)' or `goal`)
- **state_graph** (Dict[Tuple[int, int], Dict[Tuple[int, int], float]]): the dictionary defining the edge costs (e.g., `edge_weights_1` or `edge_weights_2`)
- **return_cost** (bool): logical input representing whether or not to return the solution path cost
  - If **True**, then the output should be a tuple where the first value is the list representing the solution path as tuples (row, col) and the second value is the path cost
  - If **False**, then the only output is the solution path list object

Note that in the helper functions above, two useful routines for obtaining your solution path are provided (and can be used for all the search algorithms):

- **path(previous, s)**: returns a list representing a path to state **s**, where **previous** is a dictionary that maps predecessors (values) to successors (keys)
- **pathcost(path, step_costs)**: adds up the step costs defined by the **step_costs** graph (e.g., `edge_weights_2`) along the list of states **path**

In [6]:

```
def breadth_first(start, goal, state_graph, return_cost=False):
    """Finds a shortest sequence of states from start to the goal using BFS.

    Args:
        start (Tuple): The coordinates for the start node
```

```
        goal (Tuple): The coordinates for the goal node
        state_graph (Dict): The graph to search.
        return_cst (bool): Whether or not the cost of the path should also be returned.
            Default: False

    Returns:
        The List of nodes in the shortest path, and potentially the cost (List, Optiona
    """

    frontier= [] #fifo queue
    explored= set() #list to hold explored nodes

    path_dict={} #dictionary to hold the path we have traversed
    path_dict[start]=None #initialize with just the start node an no path


    frontier.append(start)
    explored.add(start)

    while frontier:
        current=frontier.pop(0) #get front of queue

        #if it is the goal return the path
        if current == goal and return_cost:
            shortest_path= path(path_dict,goal)
            cost= pathcost(shortest_path,state_graph)
            return shortest_path,cost

        if current == goal and not return_cost:
            shortest_path= path(path_dict,goal)
            return shortest_path

        #for all neighbors of current
        for e in state_graph.get(current):
            if e not in explored: #if not visited
                frontier.append(e) #add to queue
                explored.add(e) #mark visited

                path_dict[e]=current #add to path
```

(1b)

Uniform-cost search

First, let's create our own `Frontier_PQ` class to represent the frontier (priority queue) for uniform-cost search. Note that the `heapq` package is imported in the helpers at the bottom of this notebook; you may find that package useful. You could also use the `Queue` package. Your implementation of the uniform-cost search frontier should adhere to these specifications:

- Instantiation arguments:
    - **Frontier_PQ(start, cost)**
    - **start** (Tuple[int, int]): is the initial state (e.g., **start**=(0,0) or `start`)

- **cost** (float): is the initial path cost (what should it be for the initial state?)
- Instantiation attributes/methods:
  - **states** (Dict[Tuple[int, int], float]): maintains a dictionary of states on the frontier, along with the *minimum* path cost to arrive at them
  - **q** (List[Tuple[float, Tuple[int, int]]]): a list of (cost, state) tuples, representing the elements on the frontier; should be treated as a priority queue (in contrast to the **states** dictionary, which is meant to keep track of the lowest-cost to each state)
  - appropriately initialize the starting state and cost
- Methods to implement:
  - **add(state, cost)**: add the (cost, state) tuple to the frontier
  - **pop()**: return the lowest-cost (cost, state) tuple, and pop it off the frontier
  - **replace(state, cost)**: if you find a lower-cost path to a state that's already on the frontier, it should be replaced using this method.

Note that there is some redundancy between the information stored in **states** and **q**. We only suggest to code it in this way because we think it's the most straightforward way to get something working. You could reduce the storage requirements by eliminating the redundancy, but it increases the time complexity because of the function calls needed to manipulate your priority queue to check for states (since that isn't how the frontier queue is ordered).

In [7]:

```
class Frontier_PQ:
    """Frontier class for uniform search, ordered by path cost."""

    def __init__(self, start, cost):
        """Initializes the attributes `q` and `states`
        q is a List, and states is a Dict. Each should be intialized with the start nod

        states is a dictionary, which is meant to keep track of the lowest-cost to each
        q is a list representing the elements on the frontier; should be treated as a p
        self.start= start
        self.cost= cost

        self.q=[(cost,start)]
        self.states={start:cost}

    def add(self, state, cost):
        heapq.heappush(self.q, (cost,state)) #Push new state and cost on heap
        self.states[state]=cost #add to frontier

    def pop(self):
        """Pops the lowest cost state from the `q`, and the `states` Dict."""
        return heapq.heappop(self.q)


    def replace(self, state, cost):
        """Replaces old `cost` with new `cost` iff old `cost` > new `cost`.
        This method maintains the heap invariant of `q`.

        if lower-cost path to a state that's already on the frontier, change the cost""
        self.states[state]=cost

        for i,j in self.q:
```

```
            if j[1]==state:
                self.q[i][0]=cost #updating cost it it is the one we are replacing
```

Now, actually implement a function to search using `uniform_cost` search, called as **uniform_cost(start, goal, state_graph, return_cost)**:

- **start**: initial state
- **goal**: goal state
- **state_graph**: graph representing the connectivity and step costs of the state space (e.g., **edge_weights_1** or **edge_weights_2**)
- **return_cost**: logical input representing whether or not to return the solution path cost
  - If **True**, then the output should be a tuple where the first value is the list representing the solution path and the second value is the path cost
  - If **False**, then the only output is the solution path list object

In [8]:

```
def uniform_cost(start, goal, state_graph, return_cost=False):
    """Finds a shortest sequence of states from the start to the goal with the Uniform
    """

    frontier=Frontier_PQ(start,0) #priority queue for frontier
    reached= set() #list to hold explored nodes

    path_dict={start:None}



    while frontier.q:
        parentcost, parentnode=frontier.pop() #get lowest-cost node in frontier

        #if found the goal return the path to get there
        if parentnode== goal:
            shortest_path= path(path_dict,parentnode)
            if return_cost:
                cost= pathcost(shortest_path,state_graph)
                return shortest_path,cost
            else:
                return shortest_path

        reached.add(parentnode) #mark explored

        #for all children of current node
        for childnode in state_graph[parentnode]:
            childcost=state_graph[parentnode][childnode] #get child cost

            if childnode not in reached:  #if not visited
                if childnode not in frontier.states:
                    path_dict[childnode]=parentnode #add to path
                    frontier.add(childnode, childcost+parentcost) #add to frontier

                elif frontier.states[childnode] > parentcost+childcost: # if child is i
                    path_dict[childnode]=parentnode #add to path
                    frontier.replace(childnode,childcost+parentcost)        #replace c
```

(1c)

In the code cell below, for each of the two search algorithms defined above (in **1a** and **1b**), display the following information to the screen:

First, use `edge_weights_1` : 1) Print the path from start to goal 2) Print the total cost of that path 3) Use the `show_path(env, path)` function to showcase the output your search algorithm.

Second, use `edge_weights_2` : 1) Print the path from start to goal 2) Print the total cost of that path 3) Use the `show_path(env, path)` function to showcase the output your search algorithm.

Then, in a markdown cell below your code cell, write a few sentences:

- Which algorithm yields the shortest path for both edge weights?
- Does this surprise you? Or is this your expected result?

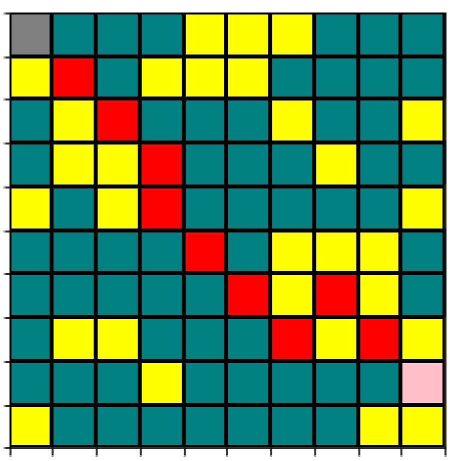
In [9]:

```
start = (0,0)
goal = (8,9)
graph1 = edge_weights_1
```

In [10]:

```
# BFS
bfs1=breadth_first(start,goal, graph1, return_cost=True)
print("The path for BFS on edge_weights_1 is ", bfs1[0])
print("The total cost for that is ", bfs1[1])
show_path(env,bfs1[0])
```

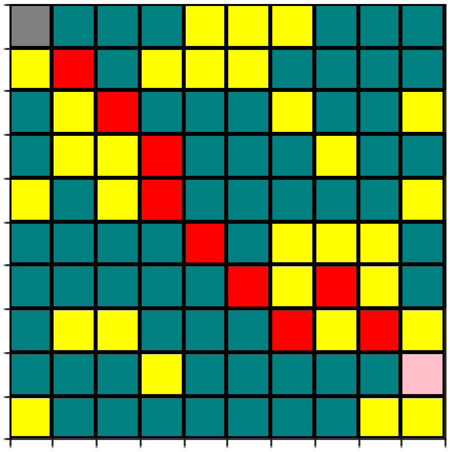
```
The path for BFS on edge_weights_1 is  [(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4),
(6, 5), (7, 6), (6, 7), (7, 8), (8, 9)]
The total cost for that is  10
```



In [11]:

```
# UCS
ucs1=uniform_cost(start,goal, graph1, return_cost=True)
print("The path for UCS on edge_weights_1 is ", ucs1[0])
print("The total cost for that is ", ucs1[1])
show_path(env,ucs1[0])
```

```
The path for UCS on edge_weights_1 is  [(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4),
(6, 5), (7, 6), (6, 7), (7, 8), (8, 9)]
The total cost for that is  10
```



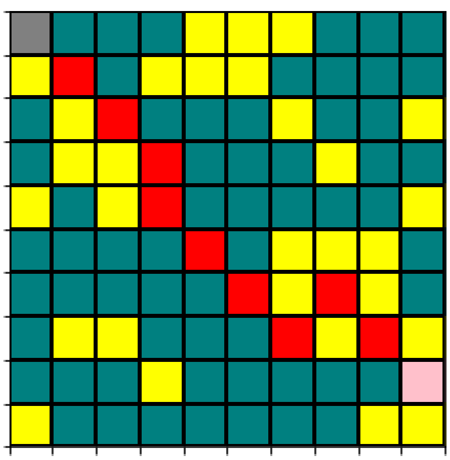
In [12]:

```
# Next we show the solution paths for edge_weights_2
start = (0,0)
goal = (8,9)
graph2 = edge_weights_2
```

In [13]:

```
# BFS
bfs2=breadth_first(start,goal, graph2, return_cost=True)
print("The path for BFS on edge_weights_2 is ", bfs2[0])
print("The total cost for that is ", bfs2[1])
show_path(env,bfs2[0])
```

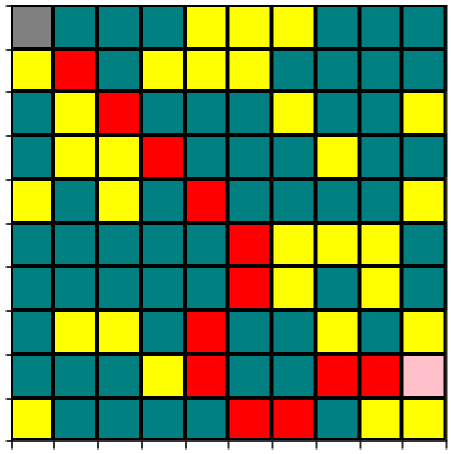
```
The path for BFS on edge_weights_2 is  [(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4),
(6, 5), (7, 6), (6, 7), (7, 8), (8, 9)]
The total cost for that is  390
```



In [14]:

```
# UCS
ucs2=uniform_cost(start,goal, graph2, return_cost=True)
print("The path for UCS on edge_weights_2 is ", ucs2[0])
print("The total cost for that is ", ucs2[1])
show_path(env,ucs2[0])
```

```
The path for UCS on edge_weights_2 is  [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5),
(6, 5), (7, 4), (8, 4), (9, 5), (9, 6), (8, 7), (8, 8), (8, 9)]
The total cost for that is  255
```



Which algorithm yields the shortest path for both edge weights?

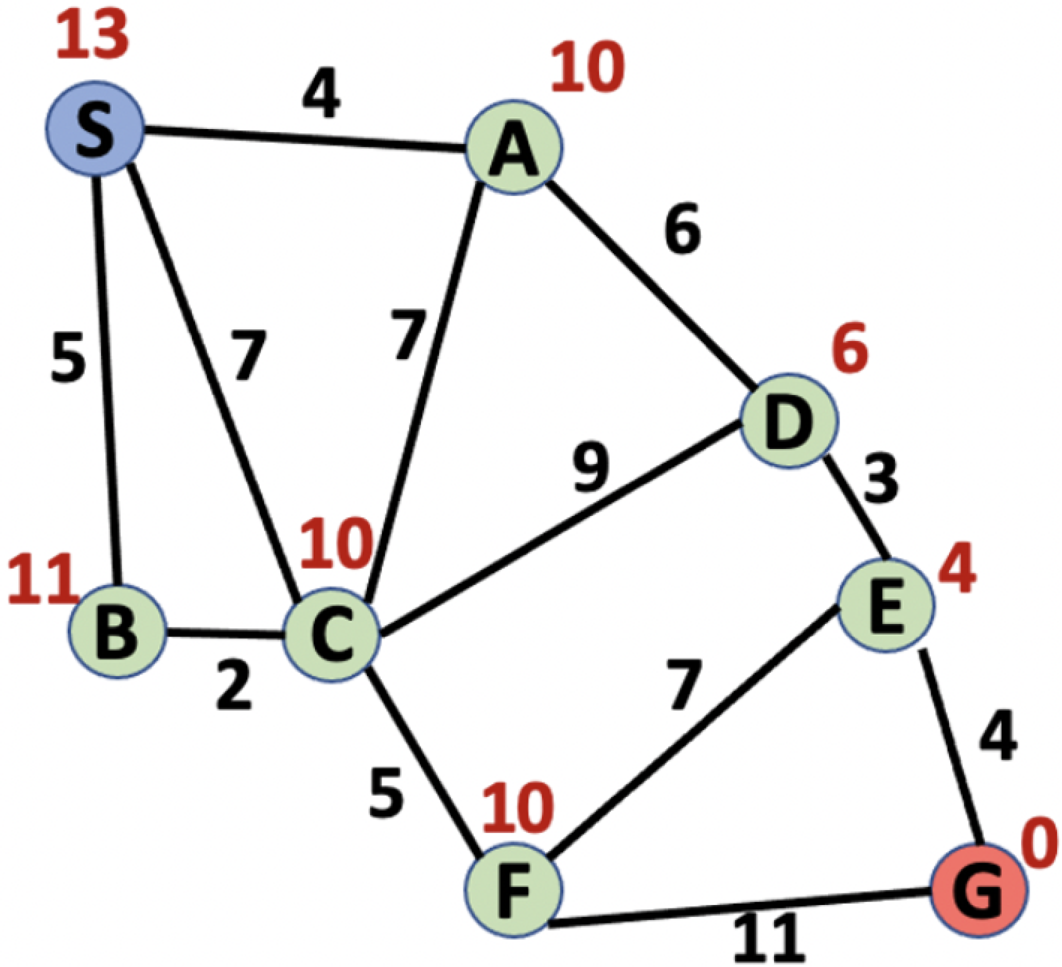
```
-UCS yields a shorter path in terms of cost for edge_weights_2, although
the path costs are the same for edge_weights_1. This is becasue the costs
in edge_weights_1 are uniform (all edged weights have a cost of 1). But,
BFS yields a path with less nodes than UCS.
```

Does this surprise you? Or is this your expected result?

```
- This was my expected result and this does not surprise me. I know that
BFS is optimal for equivalent action costs, creating the path with the
least nodes, whereas UCS is optimal path cost on graphs with either equal
or unequal costs. This is because BFS opperates by expanding shallow
nodes first and does not consider path cost, whereas UCS expands lowest
path cost first and thus always optimizes path cost.
```

Problem 2: A*

Use the graph below to go through the A* algorithm by hand to determine the path that should be taken from S to G. Heuristic values are shown in red above each node. Step costs between nodes are shown near each respective edge in black.



Fill in the table below (that is, add the necessary rows) to show the updated explored set and frontier with each iteration. The first iteration is done for you so that you can see the notation that is expected. If there are any ties, break them in alphabetical order.

Explored Nodes	**Frontier Nodes/Paths & f values**
	$(S, 13)$
S	$(A, 14), (B, 16), (C, 17)$
S, A	$(B, 16), (D, 16), (C, 17)$
S, A, B	$(D, 16), (C, 17)$
S, A, B, D	$(C, 17), (E, 17)$
S, A, B, D, C	$(E, 17), (F, 22)$
S, A, B, D, C, E	$(G, 17), (F, 22)$
S, A, B, D, C, E, G	$(F, 22)$

Resulting path is S -> A -> D -> E -> G for a cost of 17

2(a) - robot path planning

Using the robot environment in Problem 1, implement the A* algorithm using a Euclidean distance heuristic. In this implementation $h(n) = \sqrt{(x_g - x_n)^2 + (y_g - y_n)^2}$. In this equation; x_g and x_n represent the column location of goal and current node, repectively. The variables y_g and y_n are the row values of each node.

Run your code and display the results visually using the `show_path` method.

In [15]:

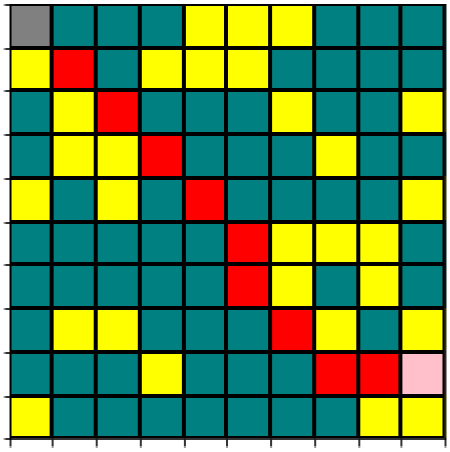
```
#define the heuirstic of Euclidean distance
def h( xg, yg, xn, yn):
    h= ((xg-xn)**2 + (yg-yn)**2)**0.5
    return h

def a_star(start, goal, state_graph, return_cost=False):
    """Runs the a* algorithm to find the shortest path from start to goal.

    returns the optimal path, and optionally the cost to get to that path."""

    frontier=Frontier_PQ(start,0) #priority queue for frontier
    reached= set() #list to hold explored nodes
    path_dict={start:None}

    while frontier.q:
        parentcost, parentnode=frontier.pop() #get lowest-cost node in frontier

        #if found the goal return the path to get there
        if parentnode== goal:
            shortest_path= path(path_dict,parentnode)
            if return_cost:
                cost= pathcost(shortest_path,state_graph)
                return shortest_path,cost
            else:
                return shortest_path

        reached.add(parentnode) #mark explored

        #for all children of current node
        for childnode in state_graph[parentnode]:
            childcost=state_graph[parentnode][childnode] #get child cost
            hn= h(goal[0], goal[1], childnode[0],childnode[1]) #hieuritic at child

            if childnode not in reached:  #if not visited
                if childnode not in frontier.states:
                    path_dict[childnode]=parentnode #add to path
                    frontier.add(childnode, childcost+parentcost+hn) #add to frontier

                elif frontier.states[childnode] > parentcost+childcost+hn: # if child i
                    path_dict[childnode]=parentnode #add to path
                    frontier.replace(childnode,childcost+parentcost)        #replace c
```

In [16]:

```
# Astar
a1=a_star(start,goal, graph1, return_cost=True)
```

```
print("The path for A* on edge_weights_1 is ", a1[0])
print("The total cost for that is ", a1[1])
show_path(env,a1[0])
```

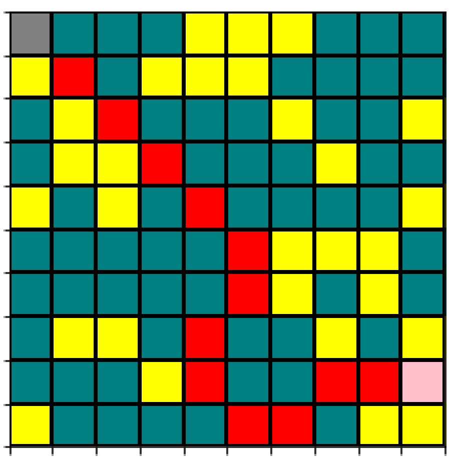
```
The path for A* on edge_weights_1 is  [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 5), (7, 6), (8, 7), (8, 8), (8, 9)]
The total cost for that is  10
```



In [17]:

```
# Astar
a2=a_star(start,goal, graph2, return_cost=True)
print("The path for A* on edge_weights_2 is ", a2[0])
print("The total cost for that is ", a2[1])
show_path(env,a2[0])
```

```
The path for A* on edge_weights_2 is  [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 5), (7, 4), (8, 4), (9, 5), (9, 6), (8, 7), (8, 8), (8, 9)]
The total cost for that is  255
```



2(b)

Explain what benefits this algorithm has over bfs, dfs, and ucs, generally speaking, and how the results compare to your results for bfs and ucs implemented in Question 1.

Your explanation here.

As we see above, A *yields the same optimal cost for edge_weights_1 as both UCS and BFS, however the path cost is different than both UCS and BFS. Then for edge_weights_2, A* yields the same path and cost as UCS. This is expected based on the behavior and optimality of each.

- While BFS is also always complete it is only optimal for equal weights (for example, edge_weights_1). It returns the path with fewest number of steps but does not consider cost like A*.
- Likewise, DFS also does not consider cost and the major downside of DFS is that very rarely optimal and also not complete in infinite state space.
- Lastly, as seen UCS is most similar to A*. It returns the lowest path cost and is complete and optimal for lowest cost of both weighted and unweighted graphs. However, since UCS is an uninformed search, it has no information about goal location. It expands nodes in order of their optimal path and does not care about number of steps a path has, only total cost of the path
- The major advantage is that A *is always optimal if the heuristic is consistent and admissible. A* is informed meaning it has additional information adds a heuristic which estimates the cost of a solution. On the other hand, BFS and DFS and UCS are uninformed searches and do not use additional information in decididing which nodes to explore.

Problem 3 - Heuristics

For questions 3A, 3B, and 3C answer True or False and provide a brief explanation, or a counterexample where applicable.

Part 3A.

Depth-first search always expands at least as many nodes as A* search with an admissible heuristic.

```
- False. In rare cases DFS will find the goal node when expanding the
least number of nodes. Consider if the solution is in depth d. Depending
on the implementation of DFS, it might traverse directly to the goal
without backtracking and thus only expand d nodes
```

Part 3B.

Uniform cost search always expands at least as many nodes as A* search with an admissible heuristic.

```
- True. A* is guaranteed to expand as many or fewer than UCS as long as
it uses an admissible (optimistic) heuristic. This means UCS expands as
many or more than A*. In the worst case the h(n)= 0, in which case A* and
UCS will expand the same number of nodes, but A* will never expand more
than UCS
```

Part 3C.

In the game of chess, in a single move, a rook can move any number of squares on a chessboard in a straight line, either vertically or horizontally, but cannot jump over other pieces. Manhattan distance is an admissible heuristic for the smallest number of moves required to move the rook from square A to square B.

```
- False. Admissible means that h(n) will never overestimate the cost from
A to B. This fails since a rook can move any number of squares in one
move. For instance, on a 6x6 board, a rook can move from the bottom left
corner to the top left corner in 1 move, however Manhattan estimates 6
thus it it not admissible.
```

In [ ]:

```

```