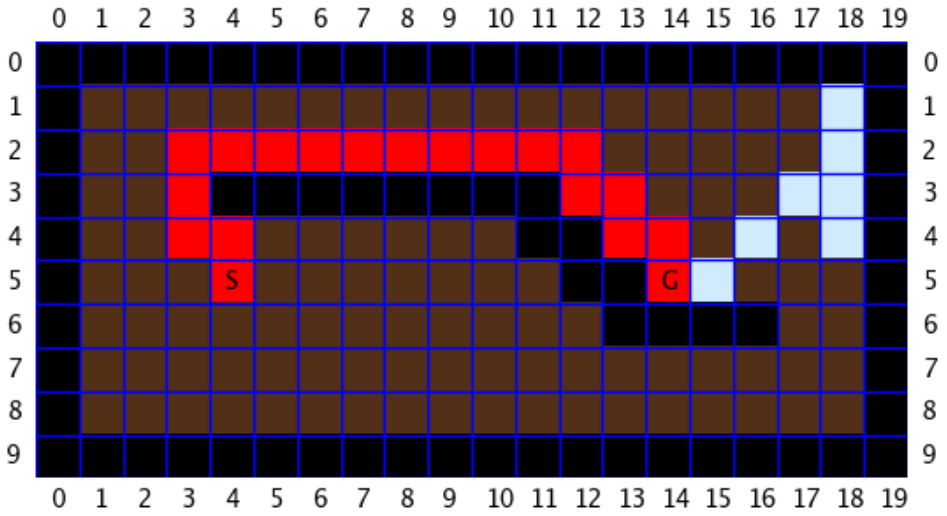
CS 133b **January 9, 2024**

Problem Set 1

Problem 1 (Dijkstra's considering only the number of steps)



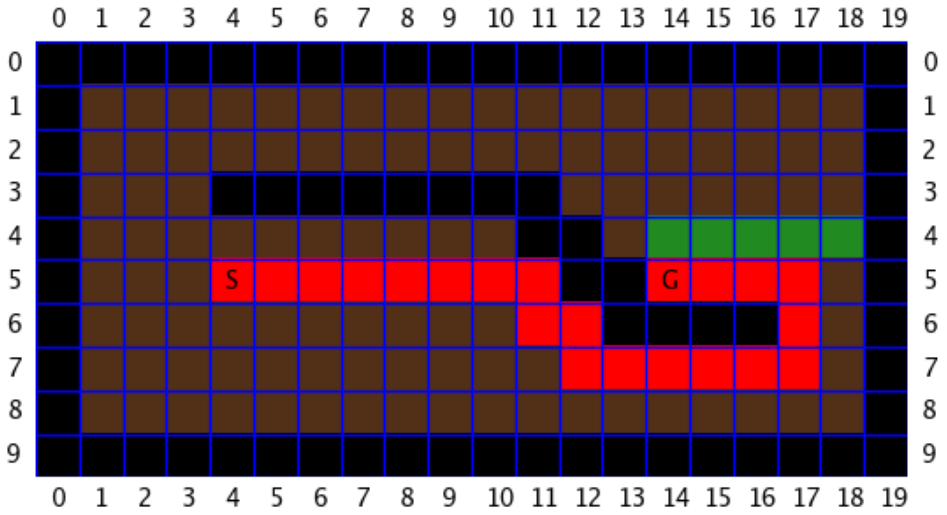
Results:
Solution cost 18.000000
122 states fully processed
0 states still pending
6 states never reached

Problem 1 (Dijkstra's considering only the number of steps)

Code:

```
def problem1(start, goal, show = None):
    # Use the start node to initialize the on-deck queue: it has no
    # parent (being the start), zero cost to reach, and has been seen.
    start.seen   = True
    start.cost   = 0
    start.parent = None
    onDeck = [start]

    # Continually expand/build the search tree.
    print("Starting the processing...")
    while True:
        # Show the grid.
        if show:
            show()

        # Make sure we have something pending in the on-deck queue.
        # Otherwise we were unable to find a path!
        if not (len(onDeck) > 0):
            return None

        # Grab the next state (first on the storted on-deck list).
        node = onDeck.pop(0)

        ####################
        for neighbor in node.neighbors:
            if neighbor.seen is False:
                neighbor.seen = True
                neighbor.done = True
                neighbor.cost = node.cost + 1
                neighbor.parent = node
                onDeck.append(neighbor)

        node.done = True
        if node is goal:
            print("Found goal")
            path = []
            path.insert(0, node)
            curr = node
            while curr.parent is not None:
                path.append(curr.parent)
                curr = curr.parent
            print(path.reverse())
            break
    return path
```

Problem 2 (Dijkstra's with move-dependent cost)



Results:
Solution cost 36.000000
123 states fully processed
5 states still pending
0 states never reached

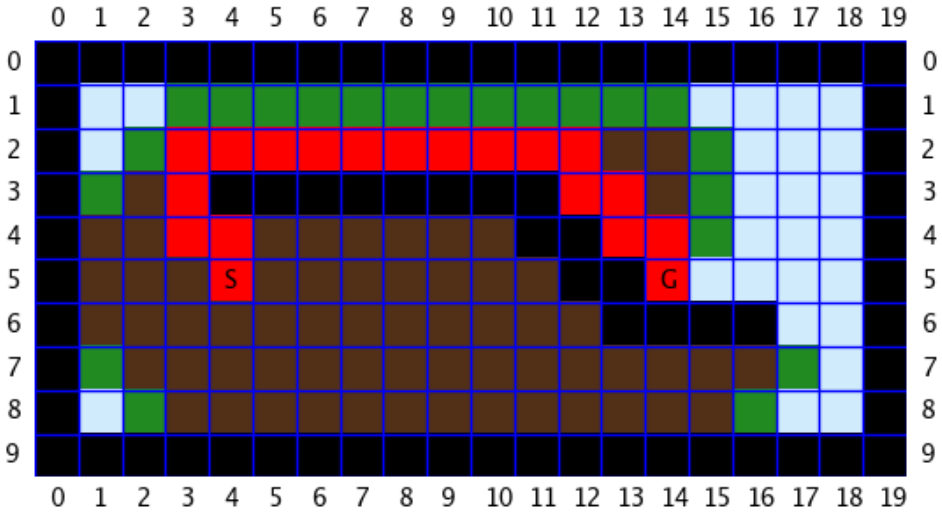
Problem 2 (Dijkstra's with move-dependent cost)

Code:

```
def planner(start, goal, show = None):
    # Use the start node to initialize the on-deck queue: it has no
    # parent (being the start), zero cost to reach, and has been seen.
    start.seen   = True
    start.cost   = 0
    start.parent = None
    onDeck = [start]

    # Continually expand/build the search tree.
    print("Starting the processing...")
    while True:
        # Show the grid.
        if show:
            show()

        # Make sure we have something pending in the on-deck queue.
        # Otherwise we were unable to find a path!
        if not (len(onDeck) > 0):
            return None

        # Grab the next state (first on the storted on-deck list).
        node = onDeck.pop(0)

        ###################
        for neighbor in node.neighbors:
            if neighbor.seen is False:
                neighbor.seen = True
                neighbor.cost = node.cost + 5 * abs(neighbor.row - node.row) + abs(neighbor.col - node.col)
                neighbor.parent = node
                bisect.insort(onDeck, neighbor)
            elif neighbor.done is False:
                newcost = node.cost + 5 * abs(neighbor.row - node.row) + abs(neighbor.col - node.col)
                if newcost < neighbor.cost:
                    onDeck.remove(neighbor)
                    neighbor.cost = newcost
                    neighbor.parent = node
                bisect.insort(onDeck, neighbor)
            else:
                pass

        node.done = True
        if node is goal:
            print("Found goal")
            path = []
            path.insert(0, node)
            curr = node
            while curr.parent is not None:
                path.append(curr.parent)
                curr = curr.parent
            print(path.reverse())
            break

    return path
```

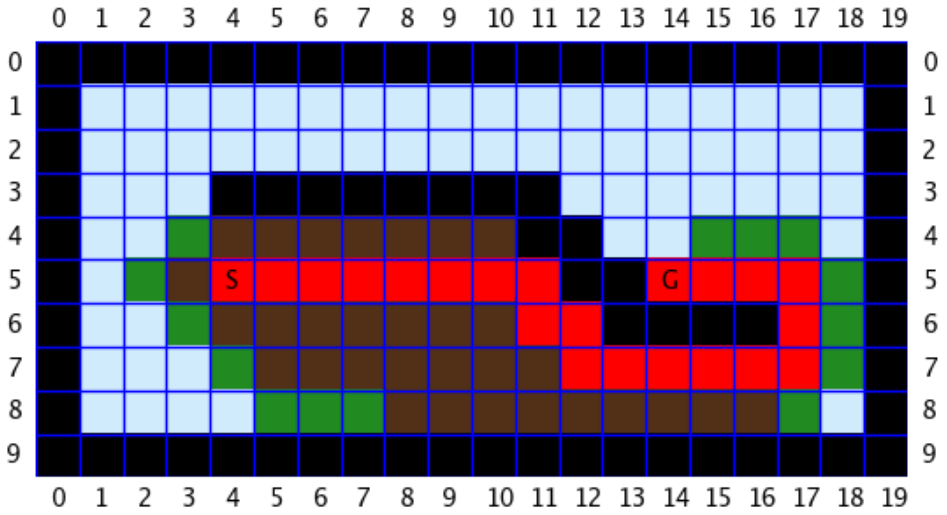
Problem 3 (A* estimating the remaining cost to the goal)

Part a



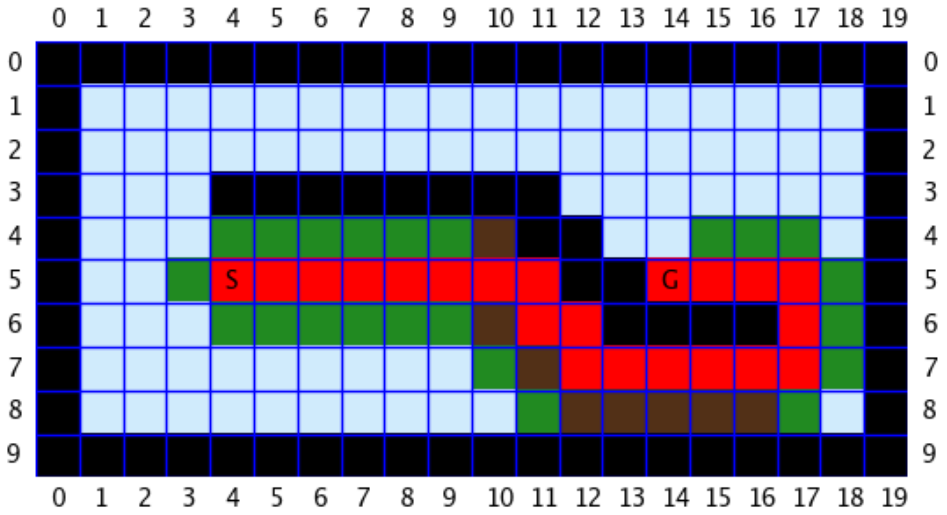
Results for part a:
Solution cost 18.000000
81 states fully processed
22 states still pending
25 states never reached

Part b



Results for part b:
Solution cost 20.000000
52 states fully processed
15 states still pending
61 states never reached

Part c



Results for part c:
Solution cost 20.000000
29 states fully processed
23 states still pending
76 states never reached

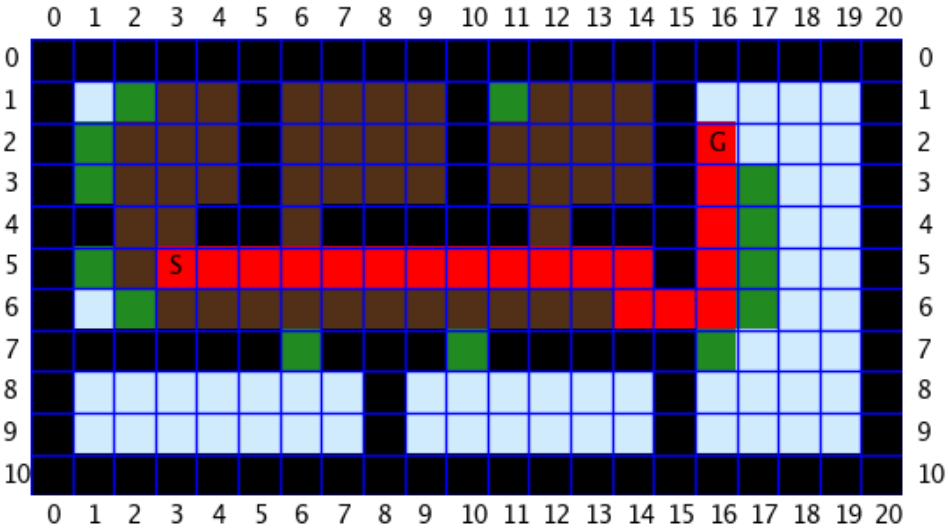
Problem 3 (A* estimating the remaining cost to the goal)

Code: k is the constant to control the level of aggressiveness

```
def planner(start, goal, show = None):
    k = 1 #constant used for agressiveness of manhattan distance

    # Use the start node to initialize the on-deck queue: it has no
    # parent (being the start), zero cost to reach, and has been seen.
    start.seen   = True
    start_cost_togo = k * (abs(goal.row - start.row) + abs(goal.col - start.col))
    start.cost   = 0 + start_cost_togo
    start.parent = None
    onDeck = [start]

    # Continually expand/build the search tree.
    print("Starting the processing...")
    while True:
        # Show the grid.
        if show:
            show()

        # Make sure we have something pending in the on-deck queue.
        # Otherwise we were unable to find a path!
        if not (len(onDeck) > 0):
            return None

        # Grab the next state (first on the storted on-deck list).
        node = onDeck.pop(0)

        ####################
        for neighbor in node.neighbors:
            if neighbor.seen is False:
                node_cost_togo = k * (abs(goal.row - node.row) + abs(goal.col - node.col))
                neighbor.seen = True
                cost_togo = k * (abs(goal.row - neighbor.row) + abs(goal.col - neighbor.col))
                neighbor.cost = node.cost + 1 + cost_togo - node_cost_togo
                neighbor.parent = node
                bisect.insort(onDeck, neighbor)
            elif neighbor.done is False:
                node_cost_togo = k * (abs(goal.row - node.row) + abs(goal.col - node.col))
                cost_togo = k * (abs(goal.row - neighbor.row) + abs(goal.col - neighbor.col))
                newcost = node.cost + 1 + cost_togo - node_cost_togo
                if newcost < neighbor.cost:
                    onDeck.remove(neighbor)
                    neighbor.cost = newcost
                    neighbor.parent = node
                bisect.insort(onDeck, neighbor)
            else:
                pass

        node.done = True
        if node is goal:
            print("Found goal")
            path = []
            path.insert(0, node)
            curr = node
            while curr.parent is not None:
                path.append(curr.parent)
                curr = curr.parent
            print(path.reverse())
            break

    return path
```

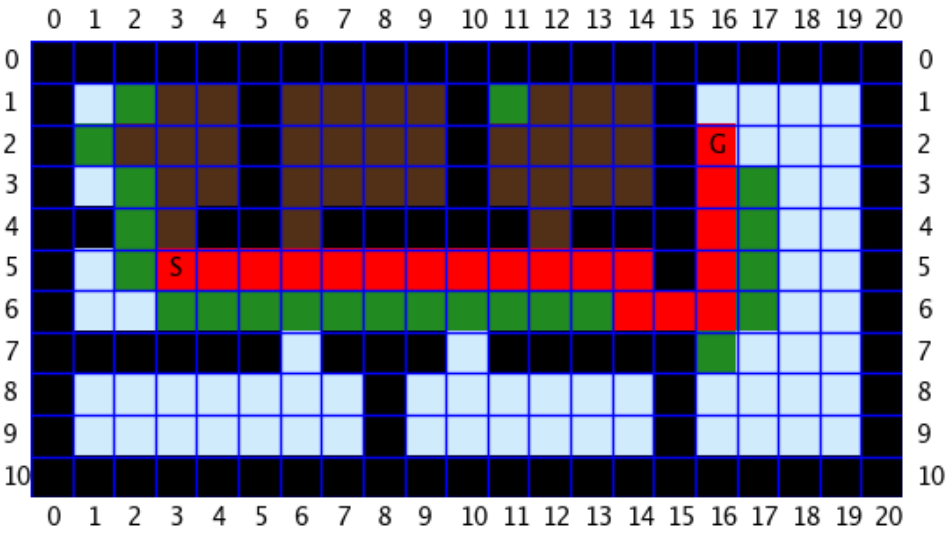
Problem 4 (Inverting the Search)

Rerun of 3a on grid 2 (not inverted)



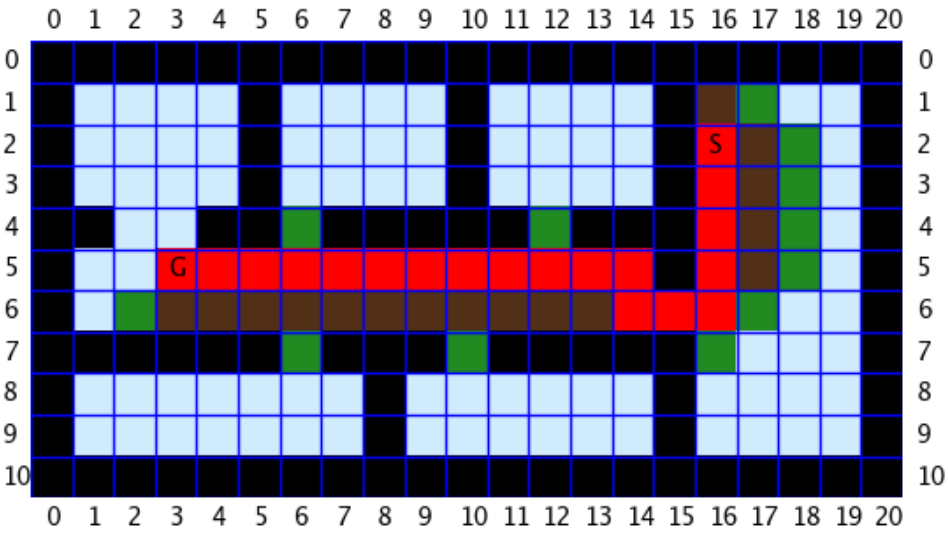
Results
Solution cost 18.000000
66 states fully processed
15 states still pending
52 states never reached

Rerun of 3c on grid 2 (not inverted)



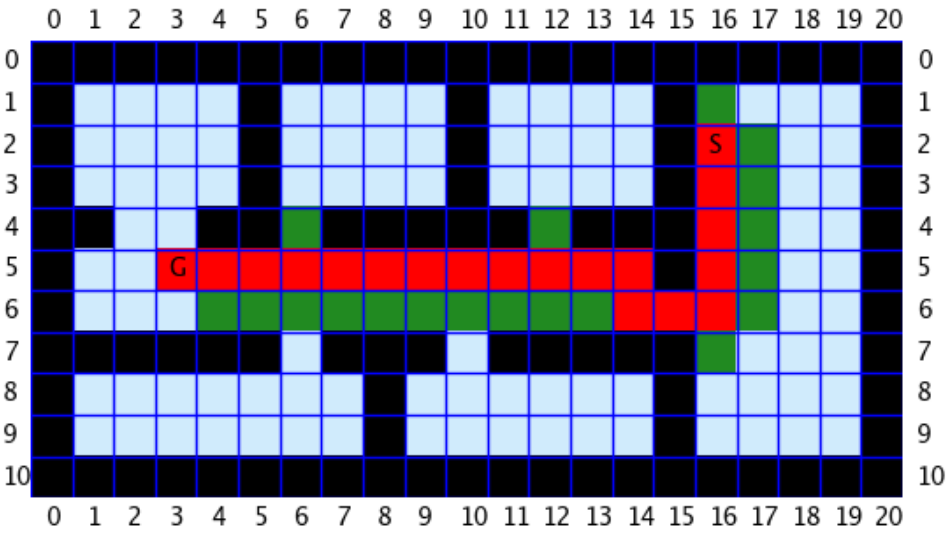
Results
Solution cost 18.000000
52 states fully processed
24 states still pending
57 states never reached

Rerun of 3a on grid 2 (inverted)



Results
Solution cost 18.000000
35 states fully processed
14 states still pending
84 states never reached

Rerun of 3c on grid 2 (inverted)



Results
Solution cost 18.000000
19 states fully processed
22 states still pending
92 states never reached

Code: Same as the one used for problem 3, to swap start and goal, the following line was used in the main function (main code):

```
start, goal = goal, start
```

Problem 5 (High-Dimensional State Space = Sokoban)

Nodes created: 6404 (9.123% done)
Number of steps: 89 steps
Code:

```
def transition(robot, boxes, direction):

    newrobot = robot.adjacent[direction]
    # if there is a wall in the way
    if newrobot is None:
        return None

    # no boxes in the way
    if newrobot not in boxes:
        return (newrobot, boxes)

    else:
        oldbox = newrobot
        # check if box can be pushed
        newbox  = oldbox.adjacent[direction]

        # there is a wall
        if newbox is None:
            return None

        # there is a box in the way
        if newbox in boxes:
            return None

        else:
            boxes.remove(oldbox)
            boxes.append(newbox)
            return (newrobot, boxes)
```

Problem 7 (Time Spent) - 4 points:

Time spent was about 3 hours, no major obstacles.