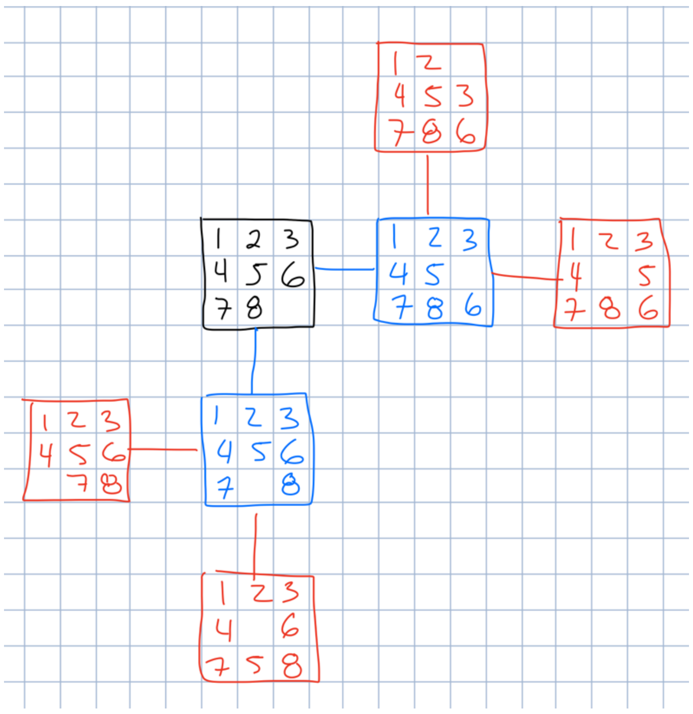
Module 2 Exercises | Joshua Shapiro | 2 March 2017

1. CODE: Download and execute Arm.java.
    a. How would you describe the initial configuration?
        i. The arm is zigzagging and the highest node is at the bottom left corner of the top red rectangle. The lowest node is in the bottom left corner.
    b. Find a configuration that meets the goal (where the arm tip is on the goal). How would you technically describe this particular final configuration?
        i. The second node is above and in the middle of the top red block.
    c. See if you can describe a few intermediate configurations. Then work with the person next to you: communicate the intermediate positions so that s/he follows the same sequence of actions to reach the goal.
        i. Lift the highest node in the y direction up until the 3rd node is where the top one used to be.
        ii. Then move the highest node to the right until its over the green dot
        iii. Then move the highest node down until it touches the green dot
2. CODE: Download and unpack planning.jar, then execute PlanningGUI. Familiarize yourself with the three planning problems, and experiment with different goal configurations: find easy and hard configurations to reach from the initial configuration. How can you tell whats easy or hard?
    a. The maze is easy when there are few obstacles and the end is close to the beginning
    b. The number puzzle is easy when the numbers are close to where they should be, and hard when they are farthest away
    c. The robot is easy when the goal is not behind obstacles. It is harder when the goal is behind an obstacle and it is extremely close to the obstacle.
3. Find a goal state for the arm problem. How does one specify a complete description of this state?
    a. You could use the relative angles of every link in the robot arm to specify a configuration.
4. How many possible states are there for each of the three problems?
    a. Maze: m*n – obstacle squares. Where m and n are the length and width of the maze
    b. Number puzzle: 9!
    c. Robot: infinite states
5. What is the size of the neighborhood for each of the three problems?
    a. Maze: 4 (assuming you can't go diagonal)
    b. Number puzzle: 4, 3, or 2 depending on the state of the puzzle
    c. Robot: infinite
6. Consider the starting state in the 8-puzzle demo. Draw this state on paper, draw all its neighbors and all neighbors of its neighbors.

a. 

7. CODE: Compile and load MazeHandPlanner into PlanningGUI for the maze problem.
    a. Click on Plan and then click on Next repeatedly to see what this algorithm produced.
    b. Examine the code in MazeHandPlanner. Create a different maze to solve, and hand-code the solution in MazeHandPlanner.
    c. Examine the code in the PuzzleHandPlanner. Observe how states are generated and entered into the plan.
8. CODE: Compile and load BFSPlanner into PlanningGUI for the maze problem. Verify that it finds the solution by clicking on the "next" button once the plan has been generated. Likewise, apply BFS to an instance of the puzzle problem and verify the correctness of the plan generated.
9. CODE: Implement DFS by modifying the BFS code.
    a. Compare BFS and DFS using a variety of puzzle-problem instances.
    b. Identify both the number of moves (time taken) and the number of steps in the path.
        i. Maze: BFS: solution length = 5, 10 moves, DFS: solution length = 5, 14 moves
        ii. 8-puzzle: BFS: solution length = 12, 939 moves, DFS: solution length = unsolved, 100000 moves
    c. Write recursive pseudocode for DFS. Is there an advantage (or disadvantage) to using recursion?

```
i. recursiveDFS(State currentState){
    1. if currentState == targetState
        a. return true
    2. else
        a. mark currentState visisted
```

            b. for each unvisited neighbor of currentState
                i. recursiveDFS(neighbor)
        ii. When using recursion if the problem is too large then you may blow the stack

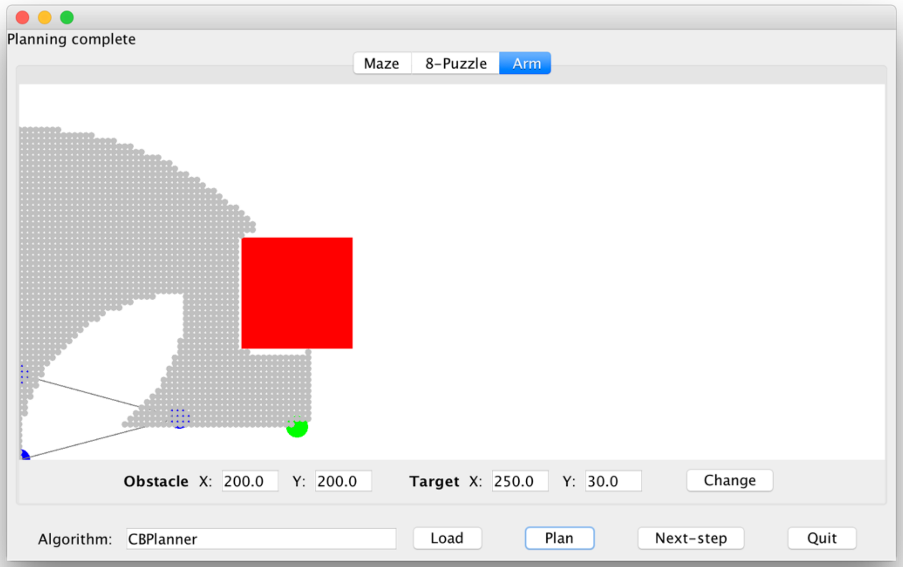
10. Compare the memory requirements of BFS and DFS. In general, which one will require more memory? Can you analyze (on paper) the memory requirements for each?
    a. BFS typically requires more memory as it stores more states temporarily. DFS however traverses down instead of out so it typically takes more time.
11. Examine the use of the two data structures in BFS and DFS. Identify the operations performed on each of these. How much time is taken for each operation performed on these data structures? Are there data stuctures which take less time?
    a. Frontier and visitedStates are both linked lists. They take O(1) time to add elements, and O(1) to remove first or last elements. To check if the list contains an element, that takes O(n). A balanced binary tree may be a better data structure since all of the actions take O(log(n)). While a stack or a queue may also be valid, they have the same access times as linked lists.
12. CODE: Implement the CBP by modifying the code in CBPlanner.java that is included in planning.jar. Most of the code has been written: you only need to extract the best node from the frontier using the costFromStart value in each state.
13. Compare BFS with cost based for the puzzle problem
    a. BFS: solution length = 12, 1111 moves
    b. CBP: solution length = 12, 1007 moves
14. Examine the operations on data structures in CBP. Estimate the time needed for these operations. Suggest alternative data structures.
    a. Frontier and visitedStates are both linked lists. They take O(1) time to add elements, and O(1) to remove first or last elements. To check if the list contains an element, that takes O(n). To find the best cost node takes O(n). If the frontier list was swapped with a priority queue based on costFromStart finding the best cost node would be O(1), and the rest of the operations would take O(log(n)).
15. What is a reasonable estimate of the cost to goal for the maze and puzzle problems? That is, from given a state and the goal, what is an estimate of how many moves it would take to get from the state to the goal? Show by example how the stimate can fail in each case.
    a. Maze: make cost the Manhattan distance from current state to the goal. If the current state has an obstacle between it and the goal, the Manhattan distance may not be a good measure.
    b. Puzzle: make current state cost the average distance between each tile and its desired location. There may be some states where while it seems close to the goal state it requires a lot of moves to get to the goal.
16. CODE: Modify the code in CBPlanner to implement the A* algorithm. Again, you do not need to perform the estimation. Simply use the estimatedCostToGoal value in each state.
17. Compare the time taken(number of moves) and the quality of solution produced by each of A* and CBP for the maze and puzzle problems. Generate at least 5 instance of

each problem and write down both measures (number of moves, quality) for each algorithm.

   a. Maze
      i. CBP: Moves: 35, Quality: 10 | A*: Moves: 25, Quality: 10
      ii. CBP: Moves: 12, Quality: 4 | A*: Moves: 6, Quality: 4
      iii. CBP: Moves: 11, Quality: 4| A*: Moves: 4, Quality: 4
      iv. CBP: Moves:15, Quality: 4| A*: Moves: 6, Quality: 4
      v. CBP: Moves: 8, Quality: 4 | A*: Moves: 5, Quality: 4
   b. 8-puzzle
      i. CBP: Moves: 7, Quality: 3 | A*: Moves: 3, Quality: 3
      ii. CBP: Moves: 101, Quality: 7 | A*: Moves: 17, Quality: 7
      iii. CBP: Moves: 33, Quality: 5 | A*: Moves: 5, Quality: 5
      iv. CBP: Moves: 135, Quality: 7 | A*: Moves: 11, Quality: 7
      v. CBP: Moves: 5342, Quality: 15 | A*: Moves: 402, Quality: 15

18. CODE: Examine the code that produces the estimatedCostToGoal for the maze and puzzle problems. Can you suggest an alternative for the puzzle problem?
   a. The maze problem uses the Euclidean distance between the current place and the end goal.
   b. The puzzle problem uses the number of places where the current configuration differs from the goal configuration. An alternative cost could be the sum of how far each tile is from where it should be. So not simply adding 1 if a tile is not in the right place, but adding 3 if the tile is 3 cells away.
19. Create an example of the maze problem and show that DFS can be arbitrarily worse than optimal.
   a. Since DFS effectively pursues one route all the way to the end before trying another, if the maze was set up to have one route that went very long through a lot of obstacles (so the path was only one square wide at all points) and finally dead ended (after going around the maze a lot) the DFS would take a lot of time without ever getting to the end.
20. Compare BFS, CBP, and A* on the arm problem. Initially, use a simple target (a short distance up the y-axis). Gradually make the target harder.
   a. When target is at (100, 200) (easy)
      i. BFS: No solution found after 100,000 moves
      ii. CBP: Moves: 444, Cost: 120
      iii. A*: Moves: 24, Cost: 120
   b. When target is at (290, 80) (harder)
      i. BFS: Not tested since it failed easy
      ii. CBP: Moves: 2264, Cost: 455
      iii. A*: Moves: 1358, Cost: 455
21. Identify the part of the code that computes the neighbors of a state in ArmProblem.java. Change the 8-neighborhood to a 4-neighborhood (NEWS) and compare. In the CPB code, you can uncomment the draw line to see what the screen looks like when the algorithm is in action.

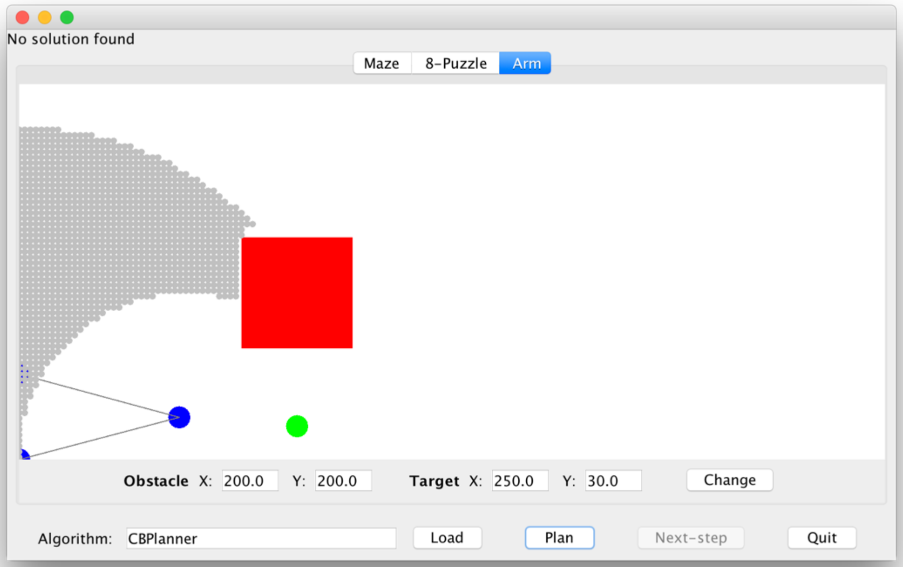
a. By limiting to NEWS, CPB cannot find some solutions that are possible. Below are two screenshots showing the difference between an 8 neighborhood and 4 neighborhood problem.
b. 8 neighborhood:


Planning complete

Obstacle X: 200.0 Y: 200.0 Target X: 250.0 Y: 30.0

Algorithm: CBPlanner

c. 4 neighborhood:


No solution found

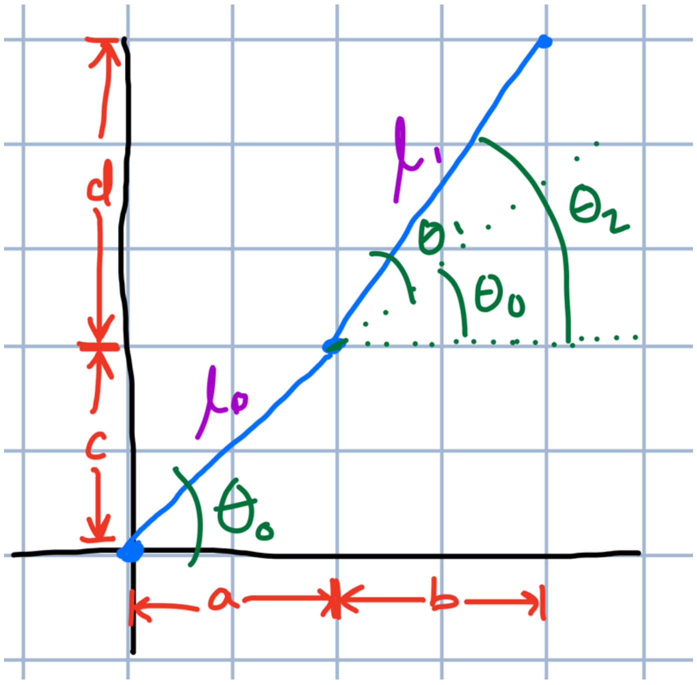
Obstacle X: 200.0 Y: 200.0 Target X: 250.0 Y: 30.0

Algorithm: CBPlanner

22. How do we know we have visited a state before? How and where is equality-testing performed in the code? What makes this equality-test different from the test used in the maze and puzzle problems?

a. The planner used keeps track of whether or not we have visited a state before. For Cost-Based Planner, whenever we remove the best state from the frontier, we mark it as visited.
b. The equality tests used for the maze and puzzle simply check if certain values match (maze current location and size). The code for equality testing is located in the problem-specific state class. For the arm state, instead of checking if values match (like the x and y positions of the arm links) it checks to see if the values lie within a range of epsilon. This is important since the arm problem is continuous. We need to make sure we don't go down a rabbit hole of checking endless states when state A of position (0.1,0.2) is considered not equal to state B of position (0.12,0.22). This epsilon equality testing mitigates this issue.

23. Why is this true?

a. 

b. The x coordinate of the end effector is clearly the sum of a and b. A can be calculated using basic trigonometry as l_0*cos(theta$_0$). To calculate b, it is a little more complicated. We can't simply use theta$_1$ in the same equation above. Instead, we must see that l_1*cos(theta$_2$) will yield our b value. From the image it can be seen that theta$_2$ = theta$_0$ + theta$_1$. Therefore, the total formula for calculating x = l_0*cos(theta$_0$) + l_1*cos(theta$_0$+theta$_1$).

c. The y coordinate of the end effector can be calculated in a similar way to the x coordinate above. Y is the sum of c and d. C can be calculated using basic trigonometry as l_0*sin(theta$_0$). To calculate d, it is a little more complicated. We can't simply use theta$_1$ in the same equation above. Instead, we must see that l_1*sin(theta$_2$) will yield our d value. From the image it can be seen that theta$_2$ = theta$_0$ + theta$_1$. Therefore, the total formula for calculating y = l_0*sin(theta$_0$) + l_1*sin(theta$_0$+theta$_1$).

24. Within each of the above phases, identify the signs (positive or negative) of delta theta0 and delta theta1. For example, when straightening, what are the signs of each likely to be?
    a. Initial position: $theta_0$ = positive, $theta_1$ = positive
    b. Straighten: $theta_0$ = positive, $theta_1$ = 0
    c. Move second link down: $theta_0$ = positive, $theta_1$ = negative
    d. Move effector horizontally: $theta_0$ = positive, $theta_1$ = negative
25. CODE: Download and unpack twolinkexample.zip
    a. Examine the code in TwoLinkController.java to confirm that the above controls are implemented.
    b. Add code for the last part, to compute theta1' nor newTheta1 in the code so that the end effector reaches (30,10).
    c. To execute, run ArmSimulator, then click on tracing then reset then go
26. Create an example of the maze problem in which Greedy performs badly
    a. If the correct path requires going farther away from the goal and there is an alternate path that moves closer to the goal but ends up in a dead end, greedy will perform poorly since it will go down the dead end path first.