# COMP 3200
# Artificial Intelligence


**Lecture 4**
Assignment 1

# Assignment 1

- Getting used to JS / HTML
- You are given UI, implement the algorithm
- Path-finding using BFS and DFS
- 4-directional grid movement
- Goal: implement BFS and DFS
- 6980: implement ID-DFS as well

# Assignment Groups

- Can work with up to 2 people total
- You must form your own groups
- One person submits each assignment
  - Add the other person’s name to comments
- You will both receive the same grade
- Please split work evenly between both
- Pair programming highly encouraged

# Assignment Grading

- Detailed grading scheme included
- Should know approx. grade before submit
- Will lose 5% per hour late (rounded up)
- One-time free 48-hour extension
  - Use it wisely, not on Assignment 1
- Come to me with problems BEFORE due
  - No excuse is valid on day assignment is due

MEMORIAL UNIVERSITY

COMPUTER SCIENCE 6980

# FASTEST A* SEARCH CODE AWARD

IS HEREBY GRANTED TO

# FIRSTNAME LASTNAME

Awarded for the fastest running A* search code among all graduate students

David Churchill — Dr. David Churchill

2025-XX-XX — Date

# JavaScript / HTML Knowledge

- Basic variable / object usage
- For / While loops
- If-statements
- Arrays / 2D arrays
- Functions / Classes
- GUI knowledge not necessary

# JavaScript / HTML Canvas Tutorial

- I used this series to jump-start me on how to use JS / HTML canvas
- Used this to make the assignment UI
- Videos use out-dated methods
- https://www.youtube.com/watch?v=XgK4YaMqQFg&list=PLcIaPHraYF7k9txsz2YprhrXmcnGpDsQI

# Programming Notes

- Do not try to write the entire assignment before testing, this will end poorly
  - You won't know where the error(s) are
- Build the house from the ground up
  - Build the floor, then walls, then roof
  - Test each before building next thing
  - Use the GUI to debug / test

# JavaScript Notes

- Use ‘let’ to create variables, not ‘var’
  - let node = new Node()
  - var uses a strange scope and can cause logical errors
- Do not compare arrays directly
  - let s1 = [7, 13]; let s2 = [7, 13]; let same = (s1 == s2);
  - This asks if they are the same array, not if content matches
  - Must use element by element comparison s1[0] == s2[0], etc
- Beware == and === (compares type)
  - console.log(2 == ‘2’);  // true, ‘2’ converted to number
  - console.log(2 === ‘2’);  // false, diff types

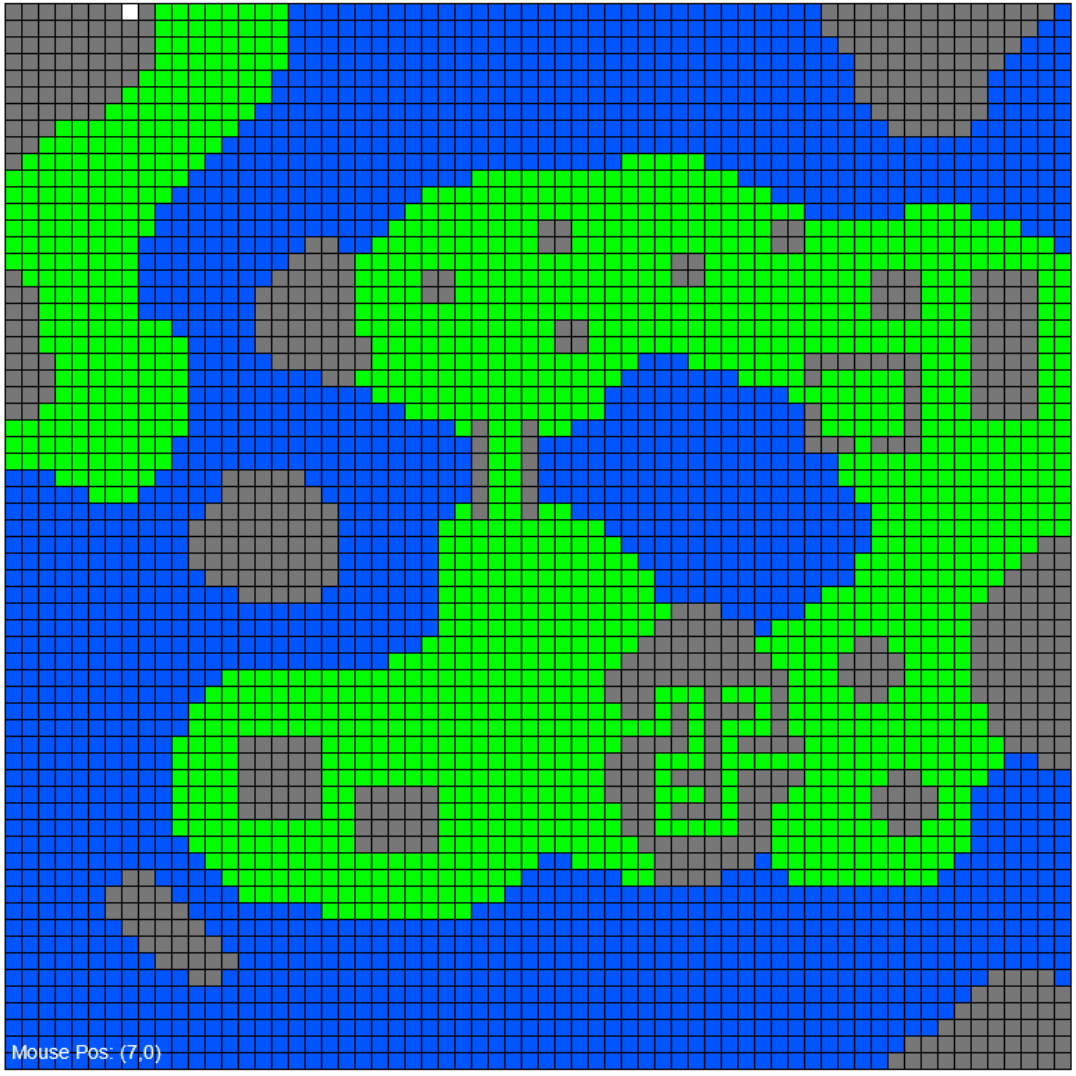
# JavaScript Notes

- Always test with the console debug window open
  - Chrome: `CTRL + SHIFT + J`
  - Firefox: `CTRL + SHIFT + K`
  - Mac: `Command + Option + I`
  - console.log(message or object) prints to console
  - Printing object/array will only show current values
    ```
    a = [1, 2, 3]; console.log(a); a.push(4); console.log(a);
    ```
- Any red message in console means error. Even if your answer is somehow correct, you lose marks
- Do not use try/catch anywhere in assignment code

# Brian Leroux - WTFJS

https://youtu.be/et8xNAc2ic8

"There's this thing called nodejs.
I don't know if it's gonna be cool or not."




Environment Map: Default (64 x 64) | Toggle Grid

Search Algorithm: Student BFS

Object Size: 1x1 Square

Legal Actions: 4 Cardinal (Up, Down, Left, Right)

Visualization: Instant Path + Open/Closed

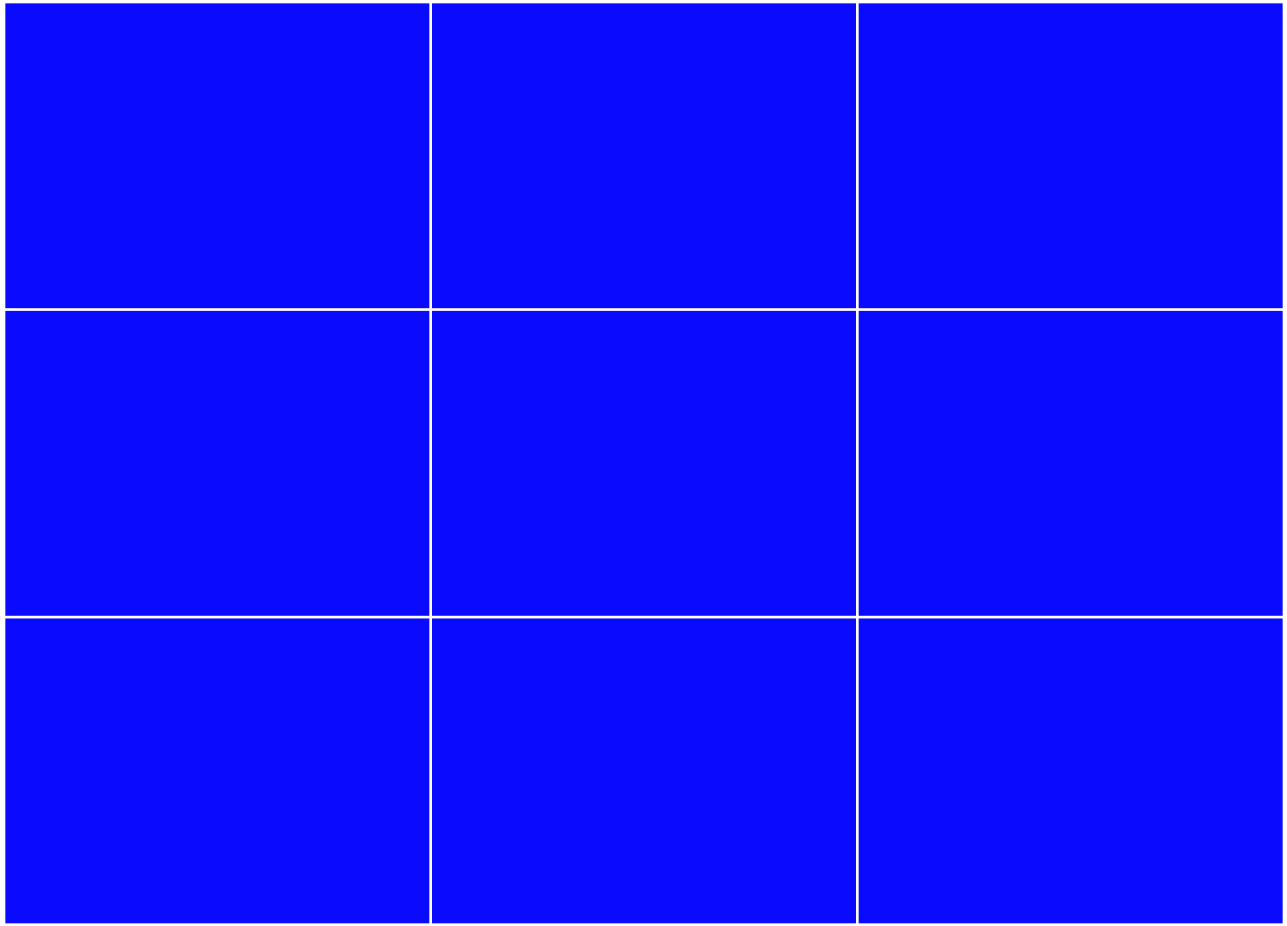
Rerun Previous Path | Run Tests | Random Tests

**Search Visualization Instructions:**

- **LEFT CLICK AND DRAG TO SET START AND GOAL TILE**
- Object can only move through same color tiles in the grid
- Click any drop-down menu above to change the search settings
- Choose 'Animate Search' visualization to see real-time search progress
- Re-Run Previous - Performs previous search again (useful when animating)
- Assignment Tests - Performs Assignment Tests
- Random Tests - Performs Random Tests Tests

**Visualization Legend:**

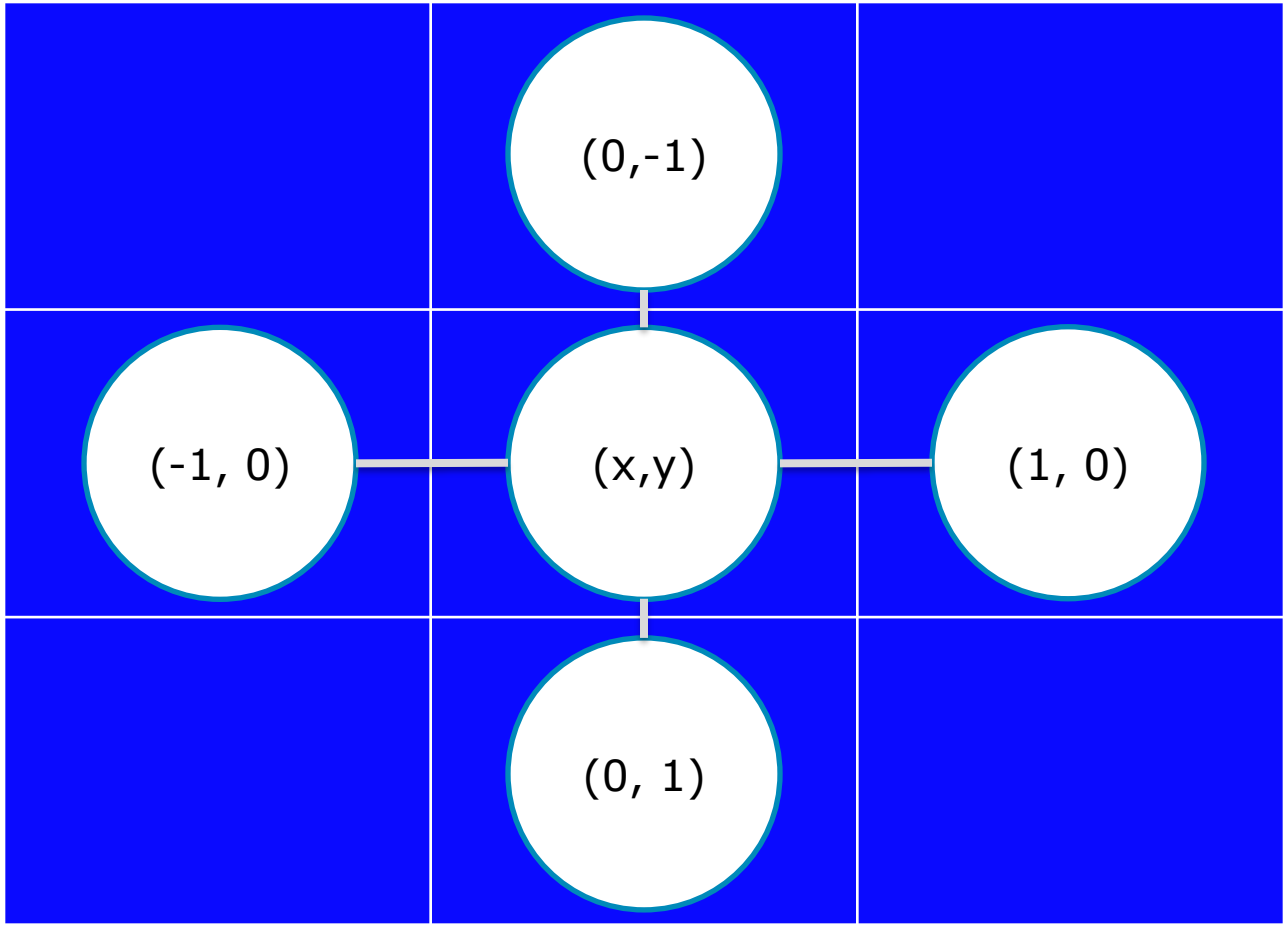
- Blue / Green / Grey Tile - Terrain type, object can move within a colour
- Red Tile - Node is in closed list (has been expanded)
- Orange Tile - Node is in open list (generated, but not expanded)
- White Tile - Node is on the generated path



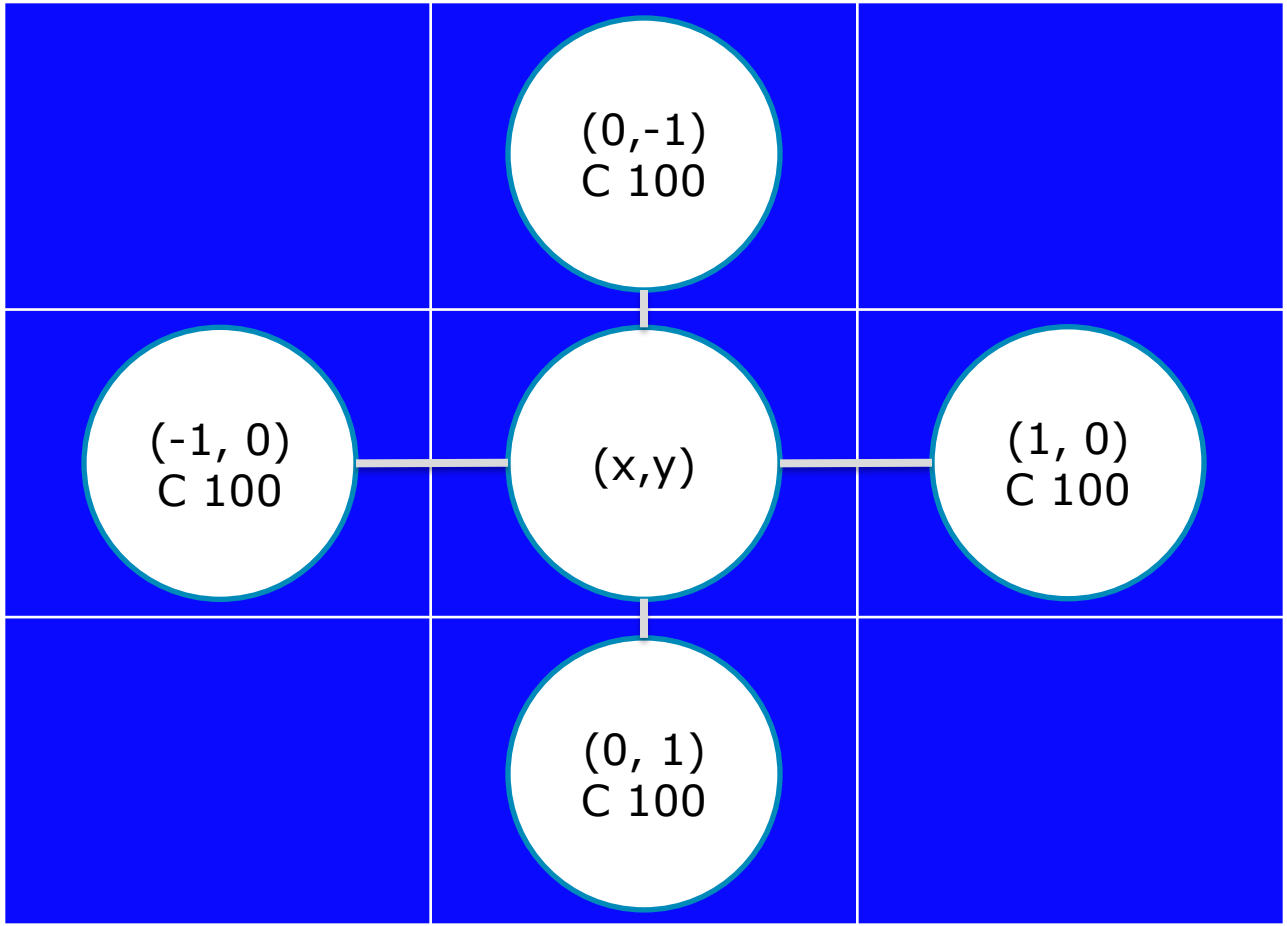
[x,y] States

| | | |
|---|---|---|
| 0,0 | 1,0 | 2,0 |
| 0,1 | 1,1 | 2,1 |
| 0,2 | 1,2 | 2,2 |

Action [x,y]

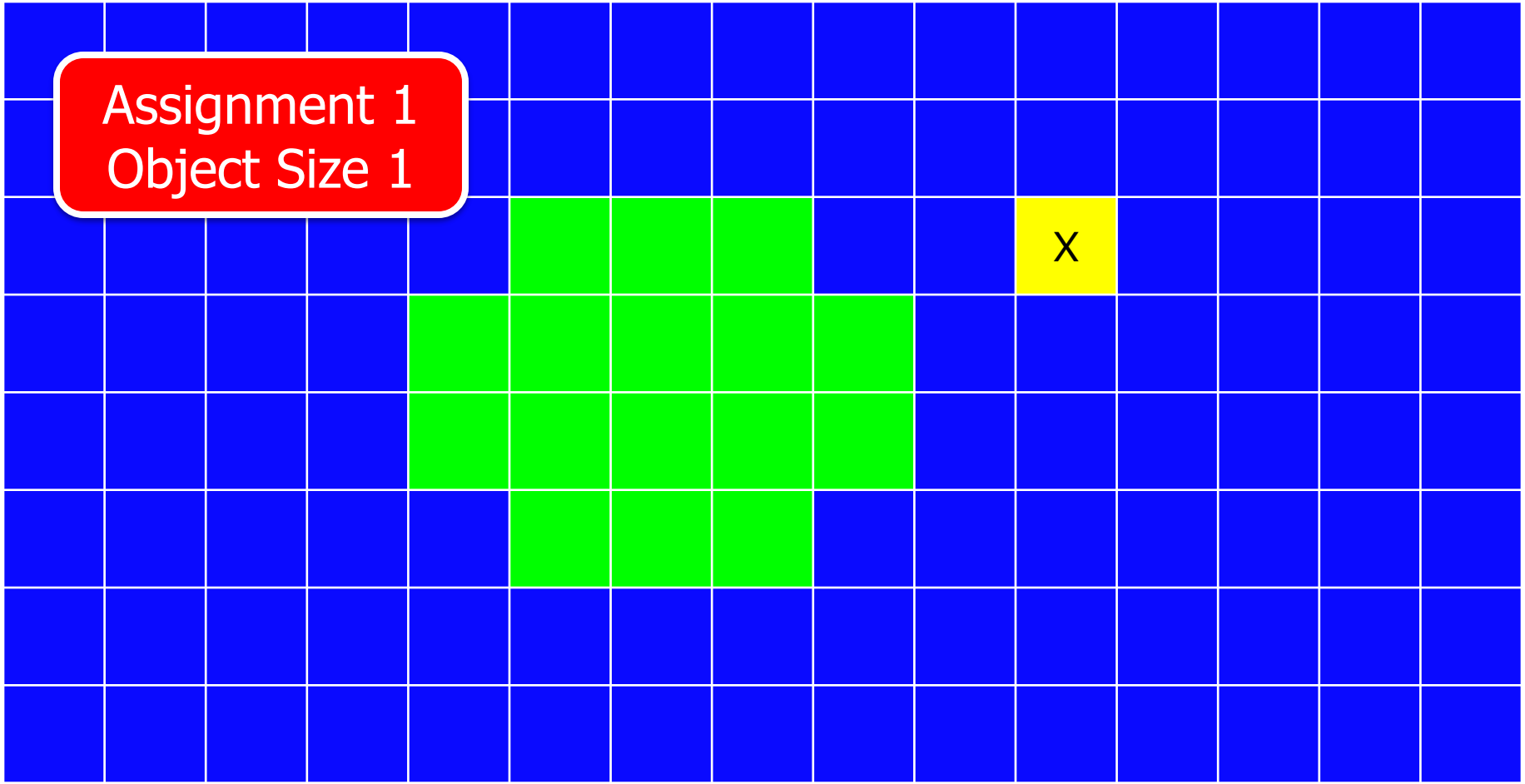


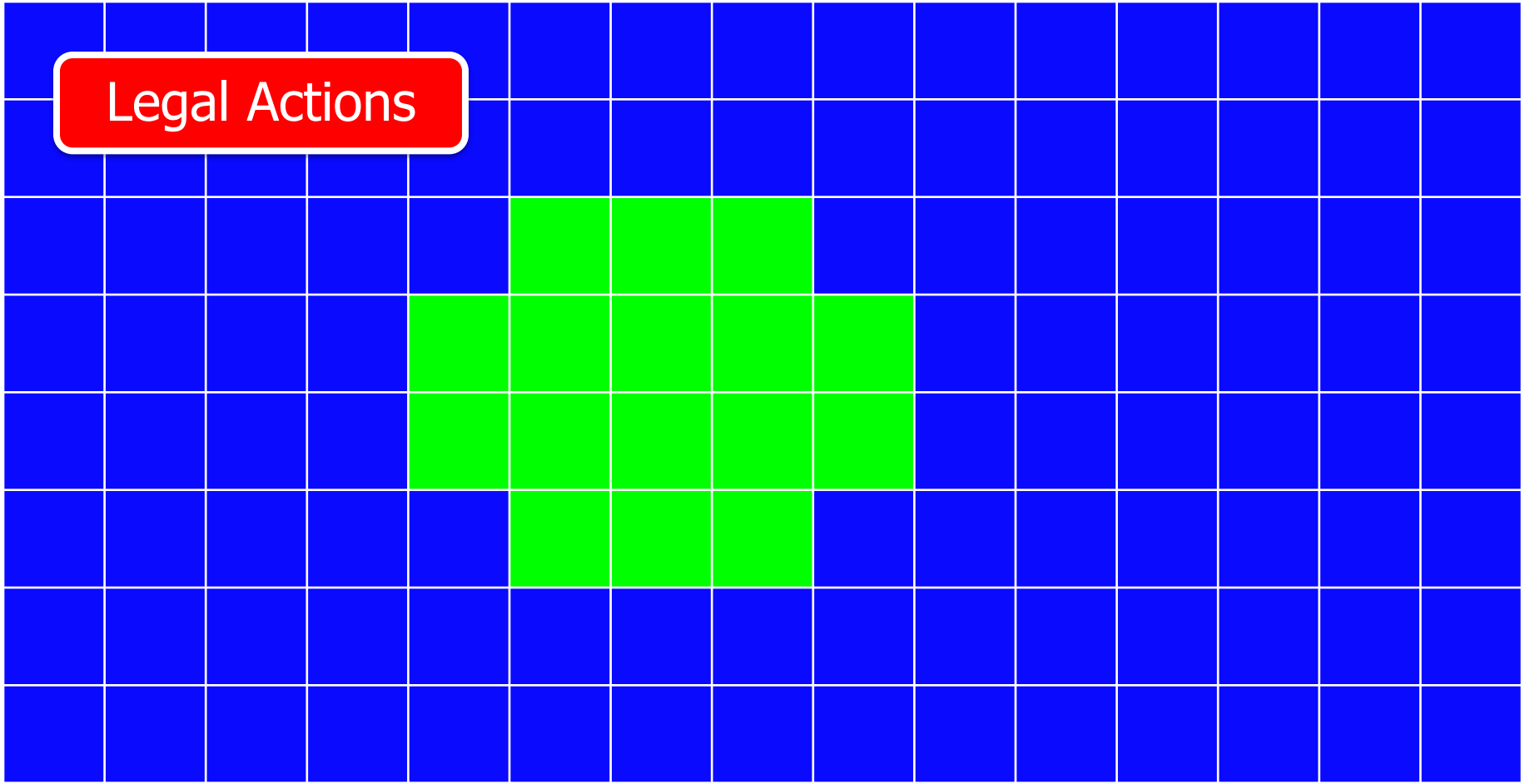
# Action Costs

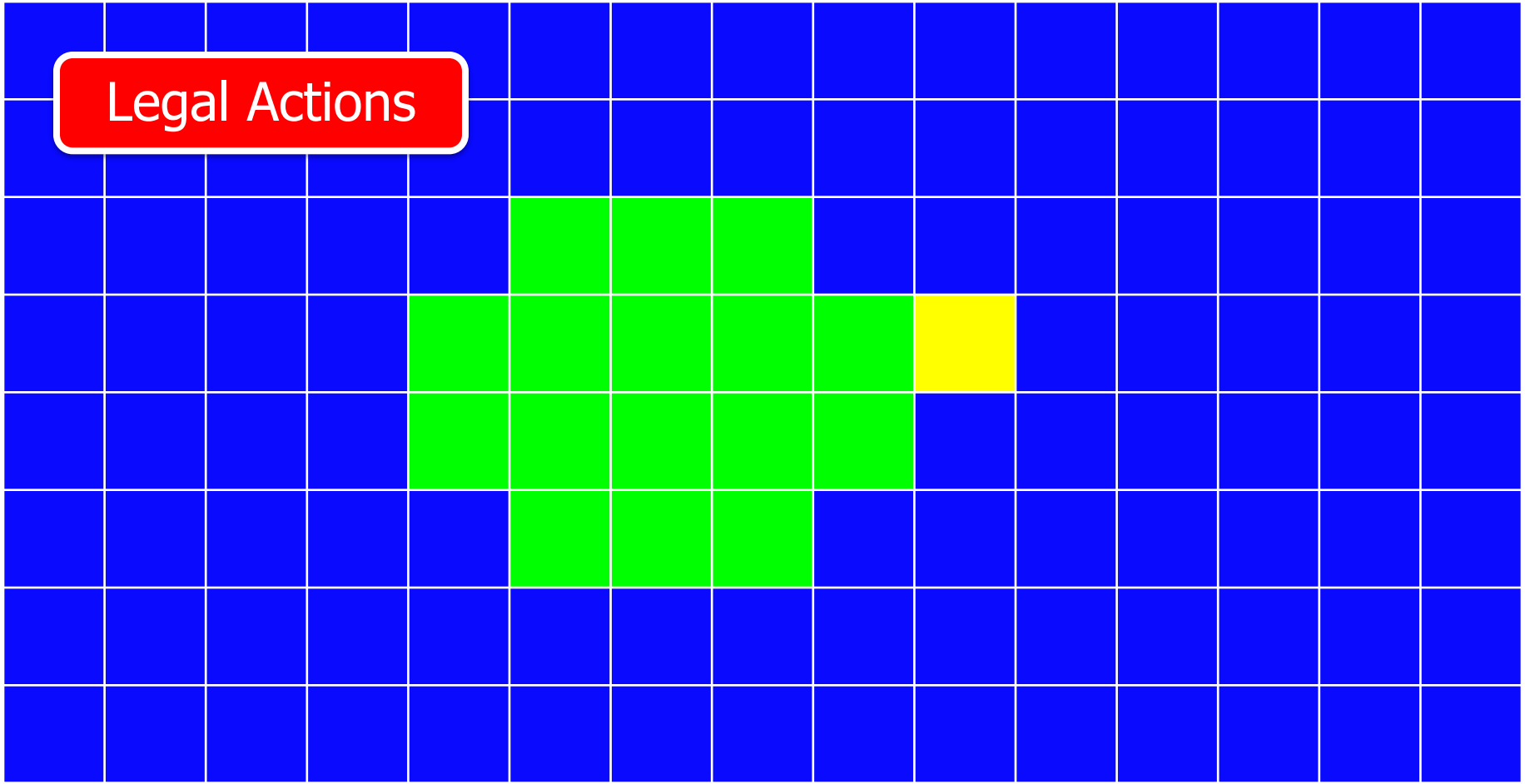


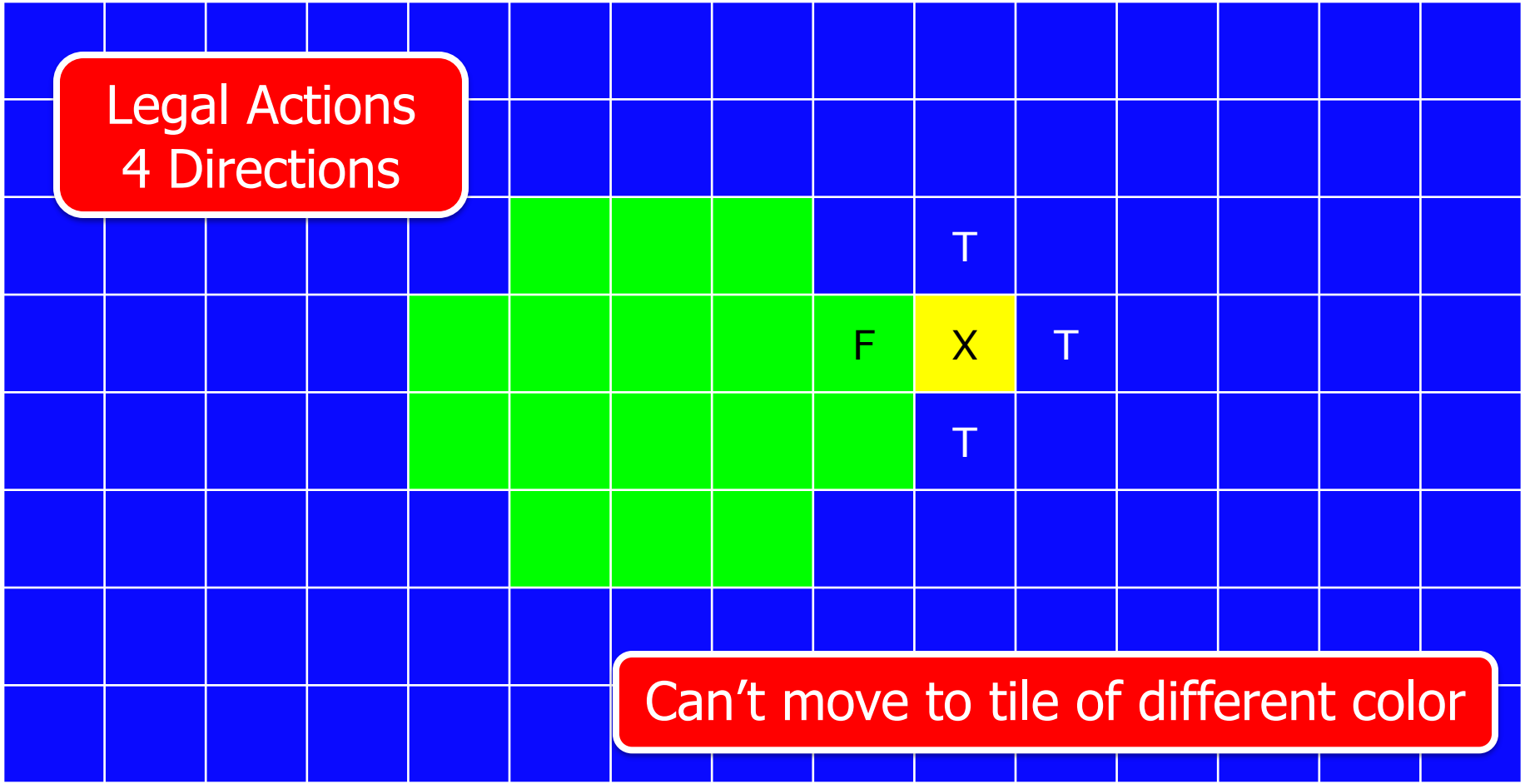
# Legal Actions

1. Object must be within the map bounds
2. Object can move 4 directions (UDLR)
3. Object can only move to a space if it is the same color as where it is currently
4. An object cannot move off of the map


Assignment 1
Object Size 1
X


Legal Actions


Legal Actions

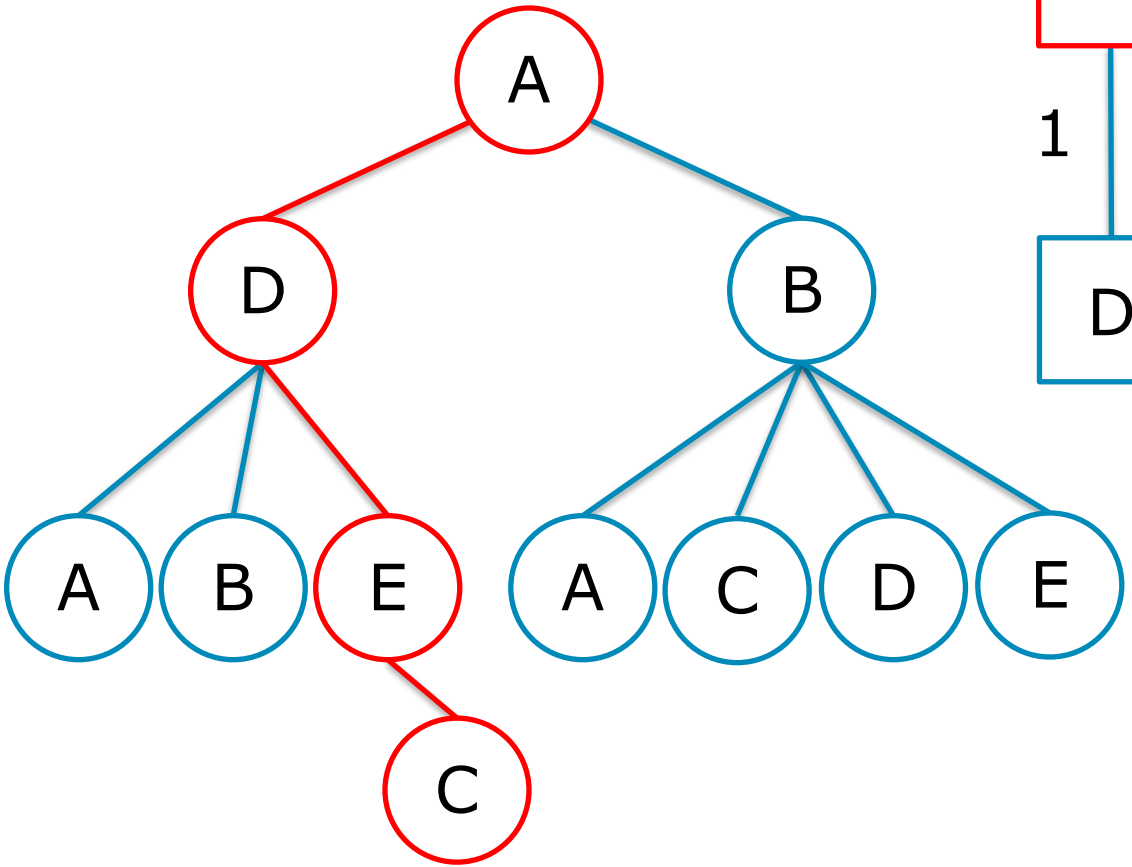

Legal Actions
4 Directions
T
F
X
T
T
Can’t move to tile of different color

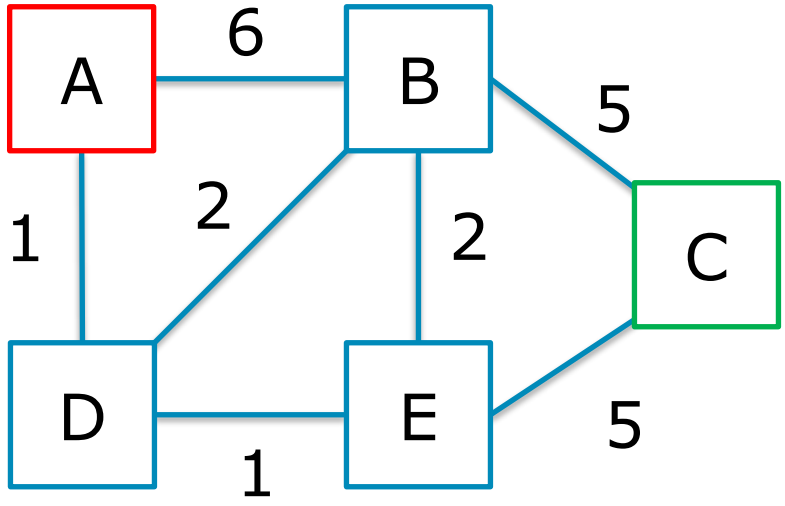
# Path Costs

- A1: Action cost = 100 for all actions UDLR
- Path cost = sum(action costs)
- No path possible?
  - Set cost = -1
  - Set path to empty
- Assignment 2
  - Cardinal = 100
  - Diagonal = 141 = approx sqrt(2) * 100

# Obtaining Final Path

- Once you select a Node from the open list that contains the goal state, you must construct the path that got you there
- Paths are a sequence of actions
  - path = [[0,1],[0,1],[1,0]] // DDR
- Follow parent pointers from goal node back to root node of search tree


A
D
B
A
B
E
A
C
D
E
C


A
6
B
5
1
2
2
C
D
E
5
1

# Obtaining Final Path

```
1. Function computePath(node)
2.     path = []
3.     while (node has parent)
4.         path.add(node.action)
5.         node = node.parent
6.     path.reverse()
7.     return path
```

# Node Class

```
class Node {
    constructor(x, y, action, parent) {
        this.x = x;
        this.y = y;
        this.action = action;
        this.parent = parent;
        this.depth = (parent == null) ? 0 : parent.depth + 1;
    }
}
```

Hint: node.depth is very helpful for ID-DFS

# Assignment 1: BFS

```
1.  Function Graph-Search(problem, BFS)
2.      closed = {}
3.      open = Queue{Node(problem.initial_state) }
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop()
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

# Assignment 1: BFS

```
1.  Function Graph-Search(problem, BFS)
2.      closed = {}
3.      open = Queue{Node(problem.initial_
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop() // get from front of queue
7.          if (node.state is goal) return solution
8.          if (node.state in closed) cont
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

startSearch()

searchIteration()

# Assignment 1 GUI Architecture

```
1. while (true)
2.     drawGUI()
3.     handleUserInput()
4.
5.     if (mode == instantPath)
6.         while (search.inProgress)
7.             this.search.searchIteration()
8.     else if (mode == animatePath)
9.         this.search.searchIteration()
```

# Assignment 1: BFS

```
1.  Function Graph-Search(problem, BFS)
2.      closed = {}
3.      open = Queue{Node(problem.initial_st
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop()
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

Set Class Variable Instead of Return

# Assignment 1: BFS

```
1.  Function Graph-Search(problem, BFS)
2.      closed = {}
3.      open = Queue{Node(problem.initial_state) }
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop()
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

# Assignment 1: DFS

```
1.  Function Graph-Search(problem, DFS)
2.      closed = {}
3.      open = Stack{Node(problem.initial_state) }
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop()
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

# Node Expansion

```
1. Function Expand(node)
2.   for (a in legalActions)
3.     newState = node.state.doAction(a)
4.
5.     newNode = new Node(state, a, node)
6.     open.add(newNode)
```

# Node Expansion

Memory Optimization

```
1. Function Expand(node)
2.   for (a in legalActions)
3.     newState = node.state.doAction(a)
4.     if (newState in closed) continue;
5.     newNode = new Node(state, a, node)
6.     open.add(newNode)
```

# Assignment 1: BFS/DFS

- BFS and DFS are both done in the searchIteration() function, and should differ by one line of code
- JS Array can simulate stack or queue:
  - Array.push() – adds to end of array
  - Array.pop() – removes and returns last element
  - Array.shift() – removes and returns first element
- BFS should pass tests
- **DFS test will not pass, it is not optimal**
- DFS path costs may not match the solution DFS costs

# Assignment 1: BFS

```
1.  Function Graph-Search(problem, BFS)
2.      closed = {}
3.      open = Queue{Node(problem.initial_state) }
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop()
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

# Assignment 1: DFS

```
1.  Function Graph-Search(problem, DFS)
2.      closed = {}
3.      open = Stack{Node(problem.initial_state) }
4.      while (true)
5.          if (open.empty) return fail
6.          node = open.pop()
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

# Assignment 1: BFS/DFS

```
1.  Function Graph-Search(problem, strategy)
2.      closed = {}
3.      open = Array[Node(problem.initial_state)]
4.      while (true)
5.          if (open.empty) return fail
6.          node = bfs ? open.shift() : open.pop()
7.          if (node.state is goal) return solution
8.          if (node.state in closed) continue
9.          closed.add(node.state)
10.         open.add(Expand(node, problem))
```

# Assignment 1: ID-DFS

- The algorithm is different enough that its own function is probably necessary
- ID-DFS done in searchIterationIDDFS()
- See tips inside Search_Student.js
- Should animate properly like solution
- In practice will be much slower than BFS

# Assignment 1: ID-DFS

- You may need to add additional class variables to Search_Student
  - Current maximum search depth d
  - Whether you have hit a cutoff or fail
  - More if necessary
- ID-DFS should produce paths the same length as BFS, but probably slower

# Iterative Deepening DFS

```
1.  Function ID-DFS(node, problem)
2.    for d in (1, infty)
3.      result = DLS(node, d, problem)
4.      if (result != cutoff) return result

// cutoff = some node reached depth limit d
//          instead of failure or solution
```

# Depth-Limited Search (for ID-DFS)

```
1.  Function DLS(problem)
2.    fringe = Stack{Node(problem.initial_state)}
3.    had_cutoff = false
4.    while (true)
5.      if (fringe.empty) return had_cutoff ? cutoff : fail
6.      node = fringe.pop()
7.      if (node.state is goal) return solution
8.      if (node.depth > L) had_cutoff = true; continue
9.      else fringe.add(Expand(node, problem))
```

# Grading Notes

- Only BFS will be tested using Run Tests for the accuracy of path lengths
- DFS will only be tested by visual inspection to make sure a path exists
- It is ok if your paths look different than the solution paths, the only thing that matters is that the costs match