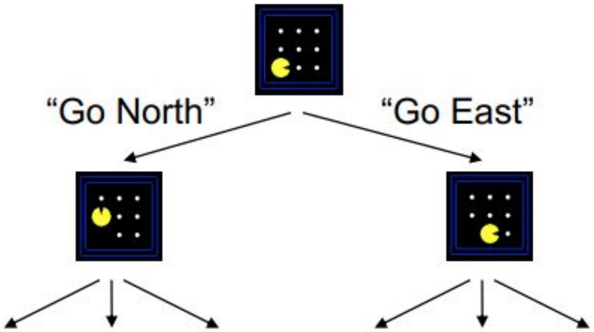
Intro to BFS/DFS

Credit

Some diagrams are taken from Dan Weld's CSE 473 class. I do not own any of this material.

Search Trees

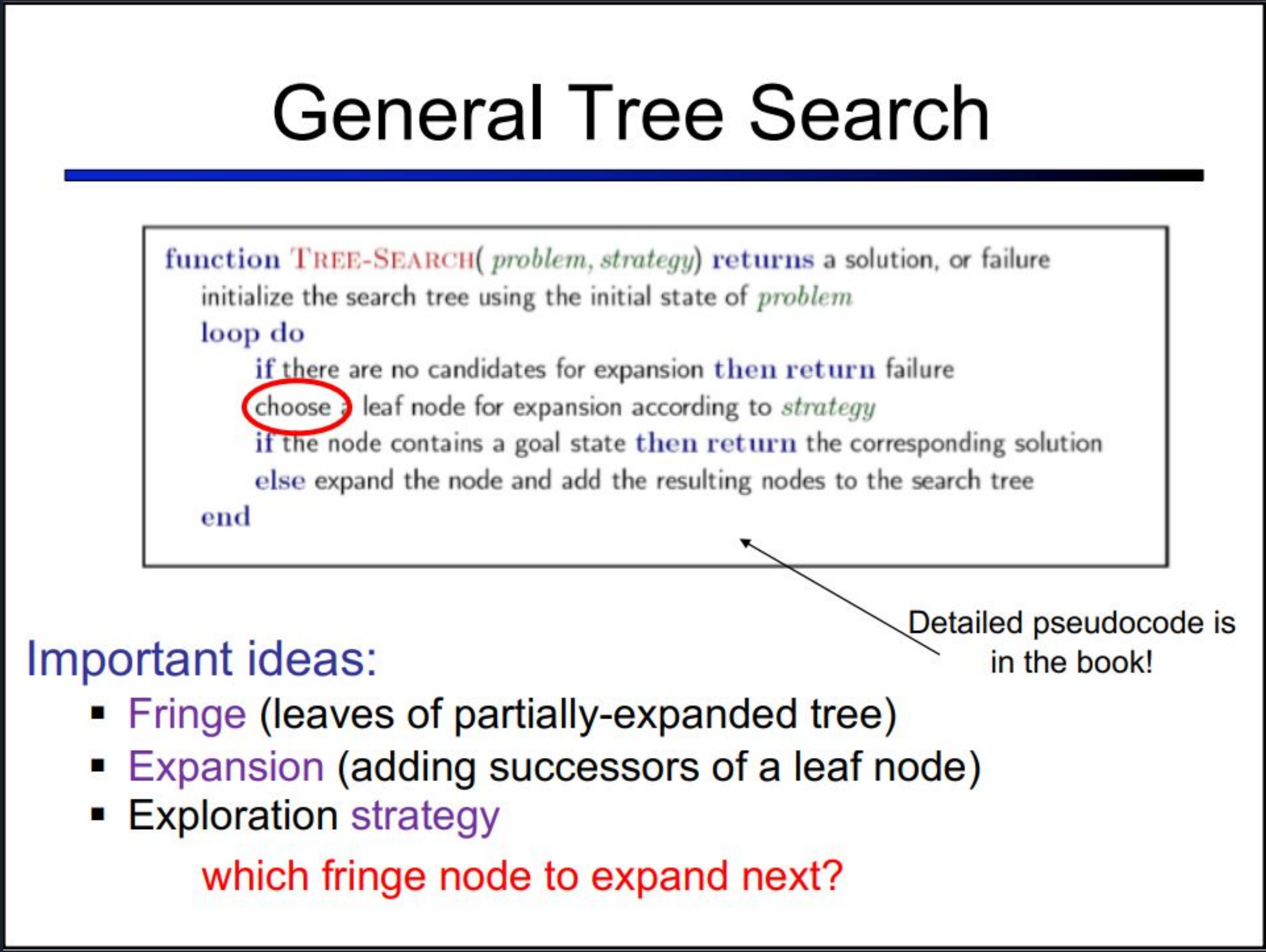


- A search tree:
  - Tree's root node corresponds to the start state
  - Children correspond to successors (application of operator)
  - Edges are labeled with actions/operators and costs
  - Nodes in the tree ***contain*** states, correspond to ***PATH*** to those states
  - For most problems, we can never actually build the whole tree

Fringe = holds states we still can explore

Strategy = which node do I choose from my fringe?

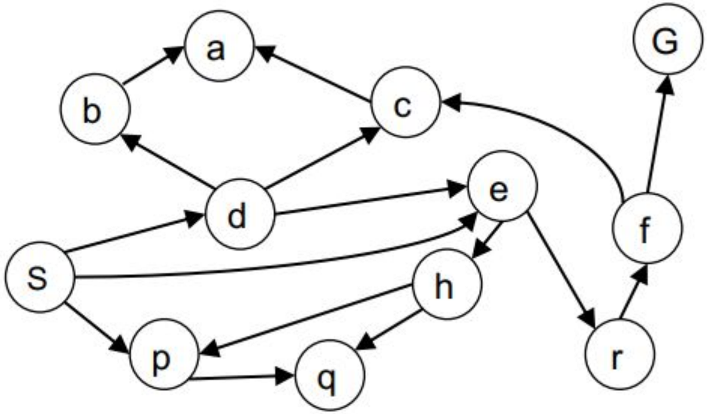
Many different possibilities:
First one we added to fringe,
Last one we added to fringe,
A random one?



Review: Breadth First Search

Strategy: expand ***shallowest*** node first

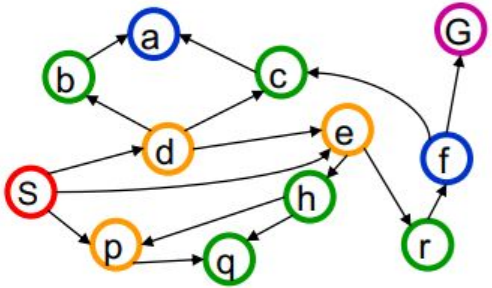
Implementation:
Fringe is a queue - FIFO

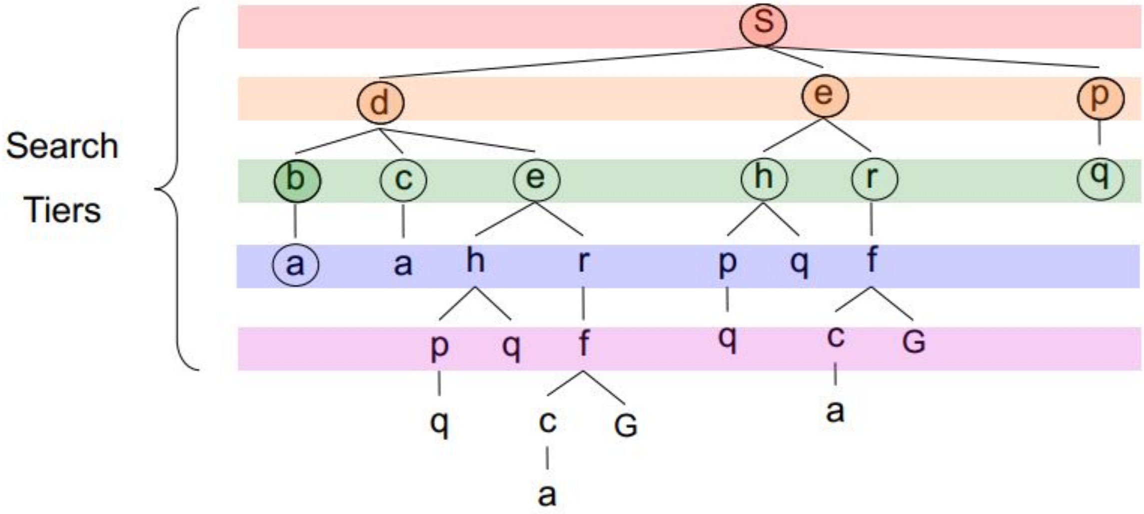


Review: Breadth First Search

Expansion order:

(S,d,e,p,b,c,e,h,r,q,a,a
,h,r,p,q,f,p,q,f,q,c,G)

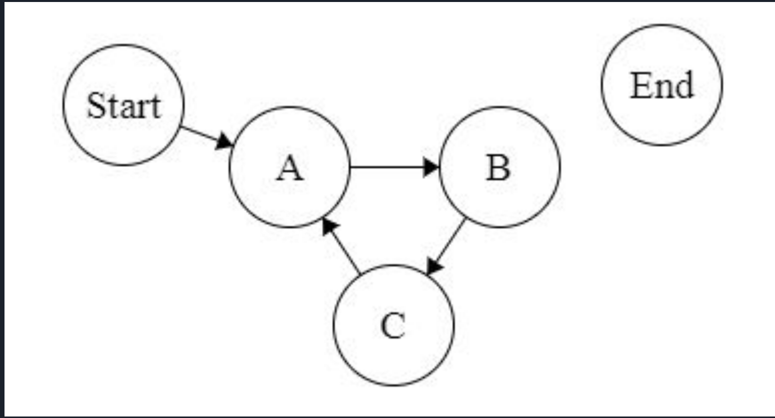




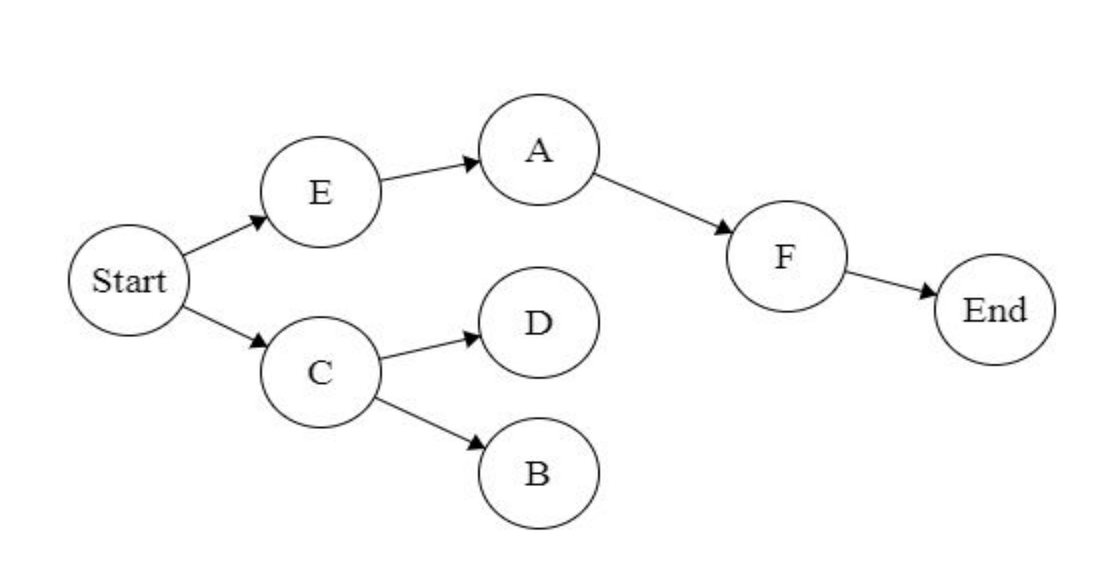
Don’t Explore the Same Node Twice!

- It doesn’t make sense to try exploring a node our search has already visited
  - Makes our fringe hold extra data we don’t need to explore
  - Could possibly lead to infinite loops with cycles
- Fix: Keep a Set of all the states we have already visited
  - Only add nodes to the fringe that we haven’t visited yet
- **Will assume throughout the rest of presentation that we have created this visited Set**

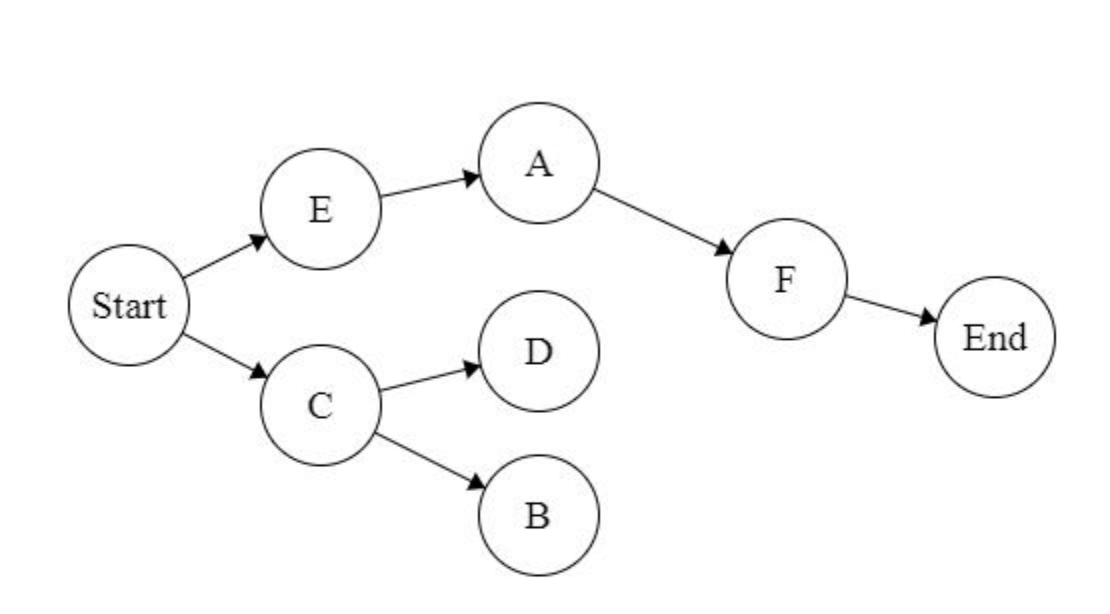
Will explore A, B, C, A, B, C, ...



BFS Exercise:
What order are nodes explored in?
Assume ties are settled by alphabetical order, C will be added to fringe before E in this example, assume search ends when we see END



Correct Answer:
Start, C, E, B, D, A, F, End



Current node: fringe

Start: [C, E]

C: [E, B, D]

E: [B, D, A]

B: [D, A]

D: [A]

A: [F]

F: [End]

BFS Pseudo-code

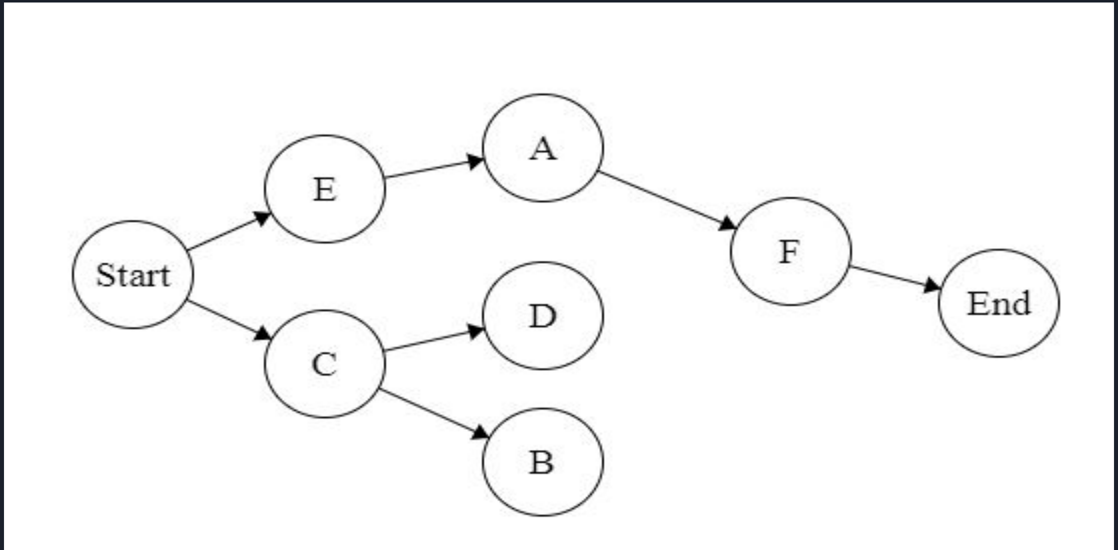
- Add the start state to the queue
- While the queue is not empty
    - Remove the next state from the queue and set as current
    - Mark the current state as visited
    - If current state is the goal
        - return the path to the goal
    - otherwise
        - for each successor state
            - if successor has not been marked as visited
                - save path and add the successor to the queue

Let’s Implement BFS in Friends.java

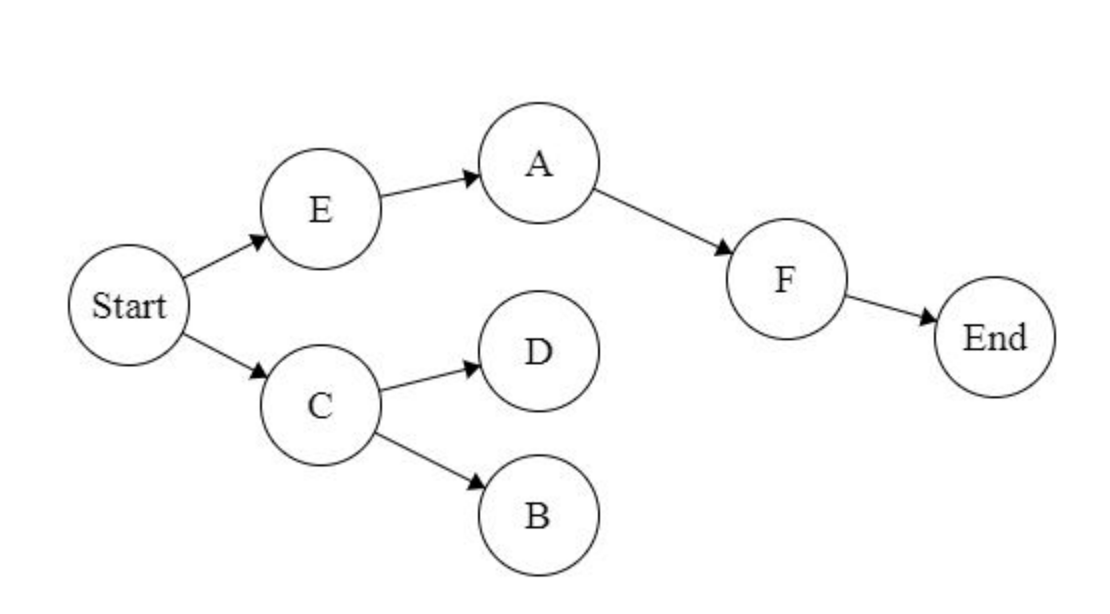
Depth-First-Search (DFS)

- Same application of the general Tree Search but our STRATEGY changes
- New Strategy : Expand the DEEPEST NODE FIRST
- Fringe is implemented as a Stack instead
    - The most recently discovered node will be expanded next

DFS Exercise:
What order are nodes explored in?
Assume ties are settled by alphabetical order, C will be added to fringe before E in this example, assume search ends when we see END



Correct Answer: bot [1, 2, 3, 4, 5] top
Start, E, A, F, End



Current node: fringe

Start: [C, E]

E: [C, A]

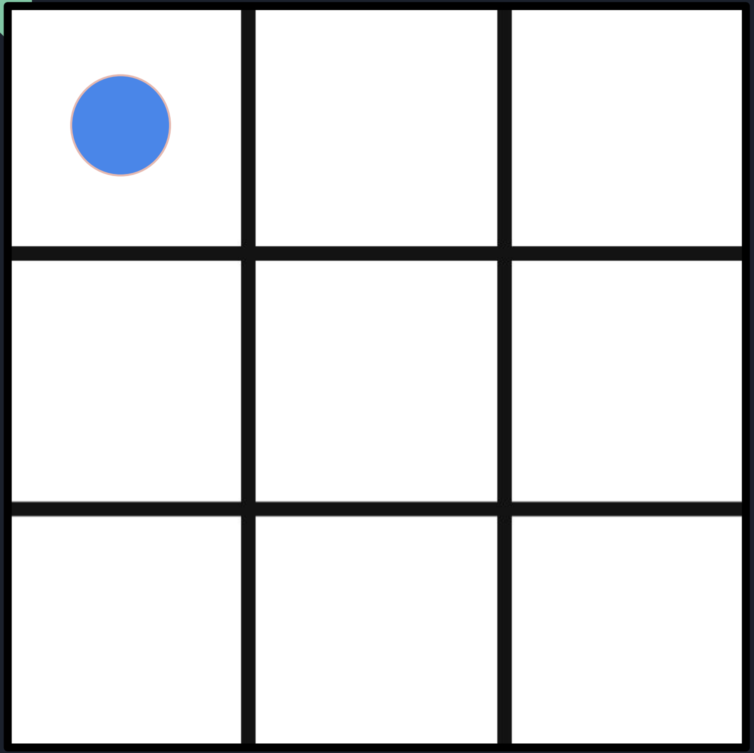
A: [C, F]

F: [C, End]

DFS Pseudo-code

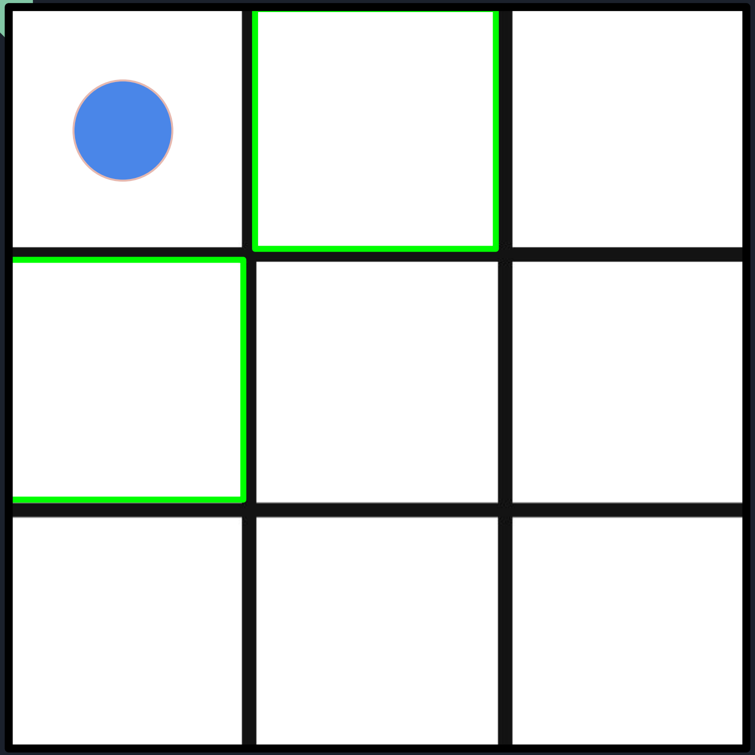
- Push the start state to the stack
- While the stack is not empty
  - Pop the next state from the stack and set as current
  - Mark the current state as visited
  - If current state is the goal
    - return the path to the goal
  - otherwise
    - for each successor state
      - if successor has not been marked as visited
        - save path and push successor to the stack

Maze Generation



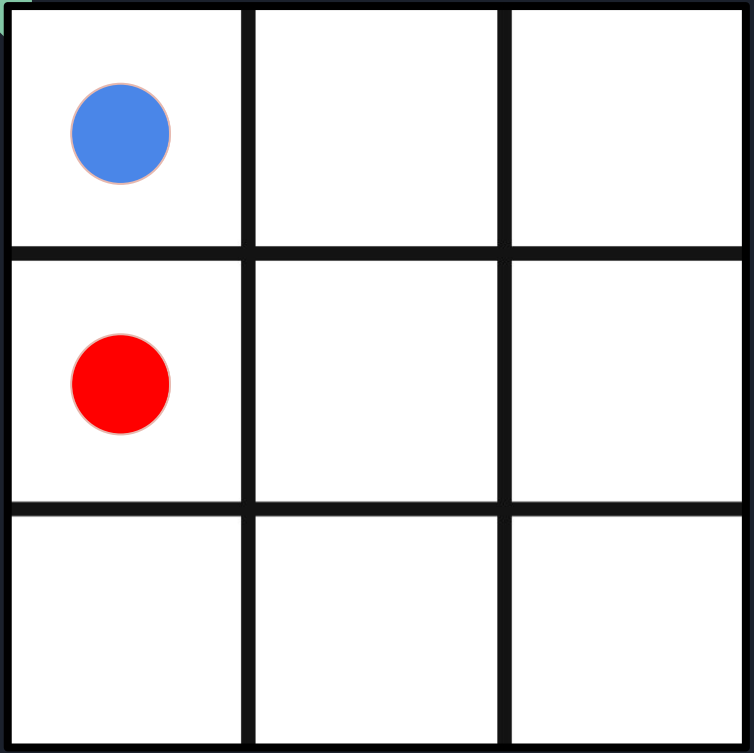
- Set a random cell as the current cell

Maze Generation



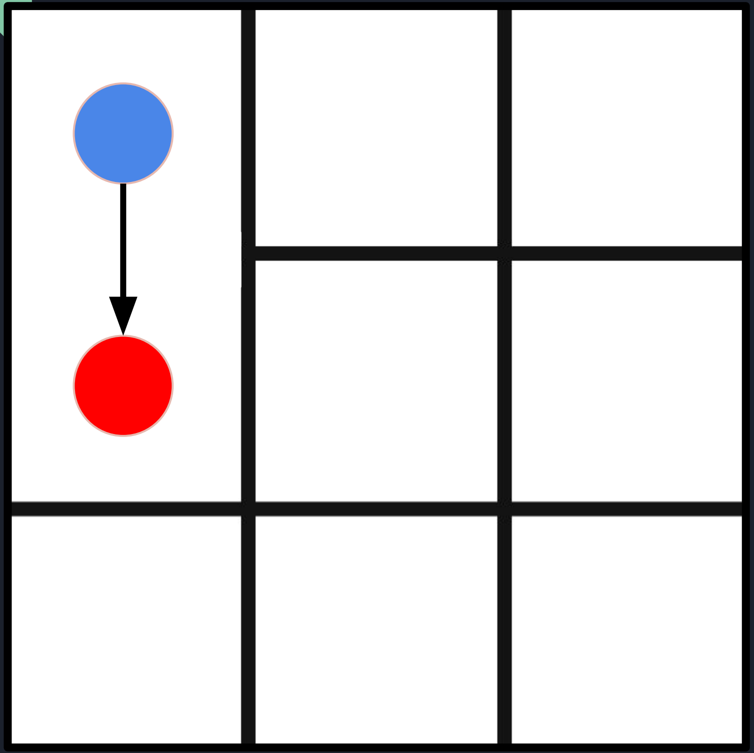
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet

Maze Generation



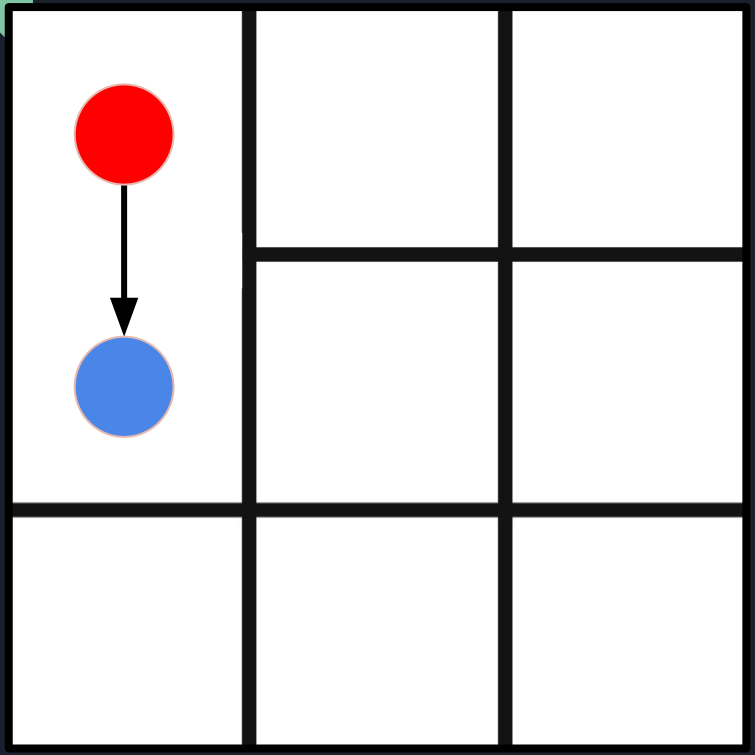
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet

Maze Generation



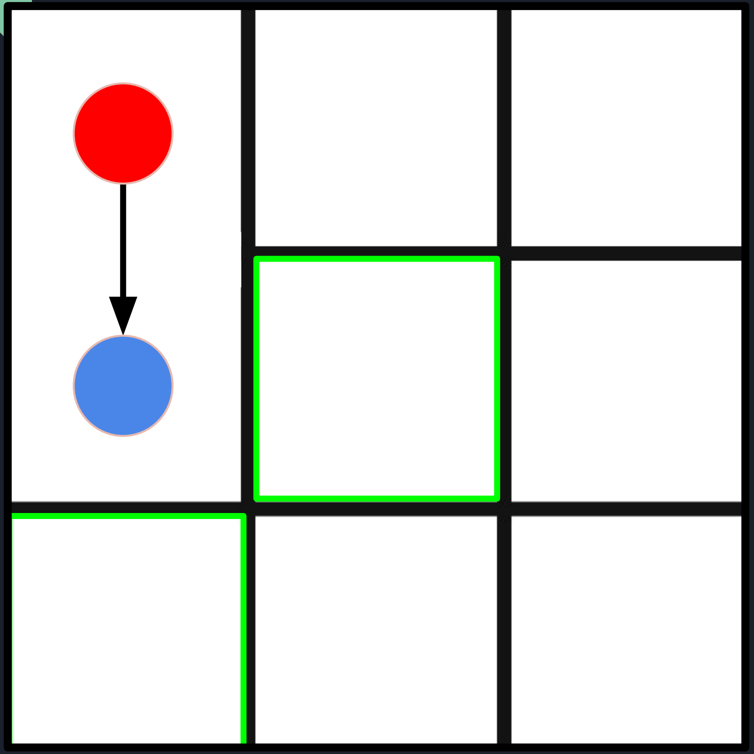
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet
  - Remove the wall between them

Maze Generation



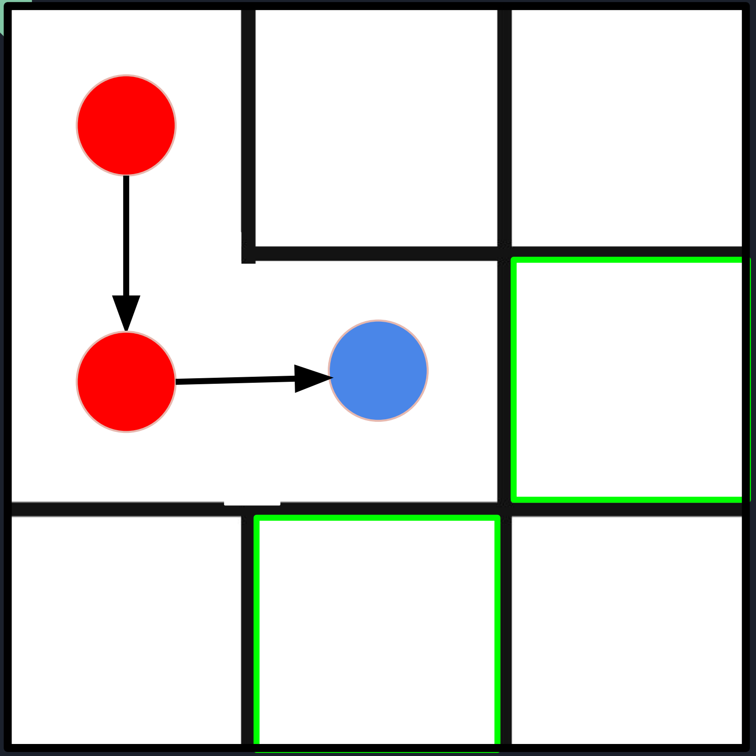
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet
  - Remove the wall between them
  - Set new cell to current

Maze Generation



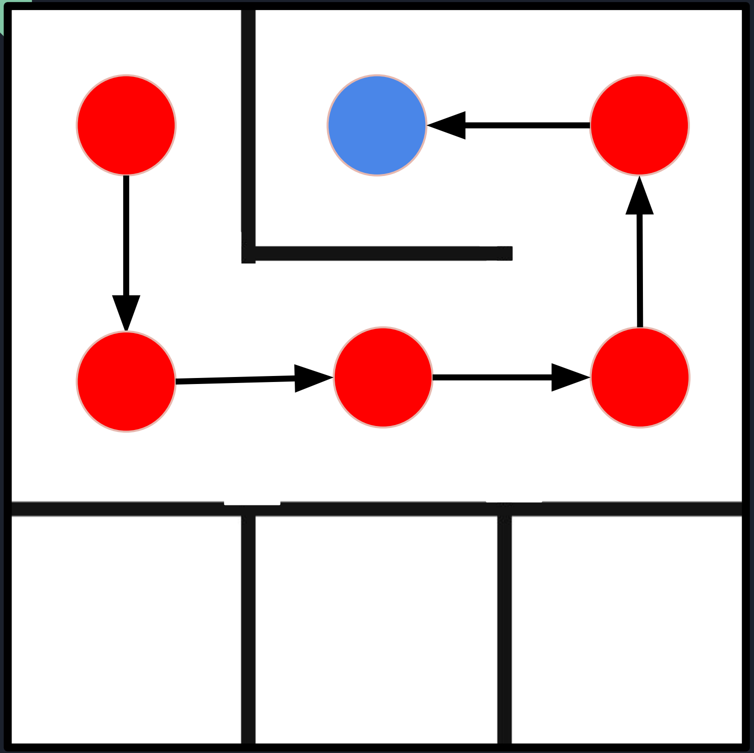
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet
  - Remove the wall between them
  - Set new cell to current
  - Repeat until we reach a dead end

Maze Generation



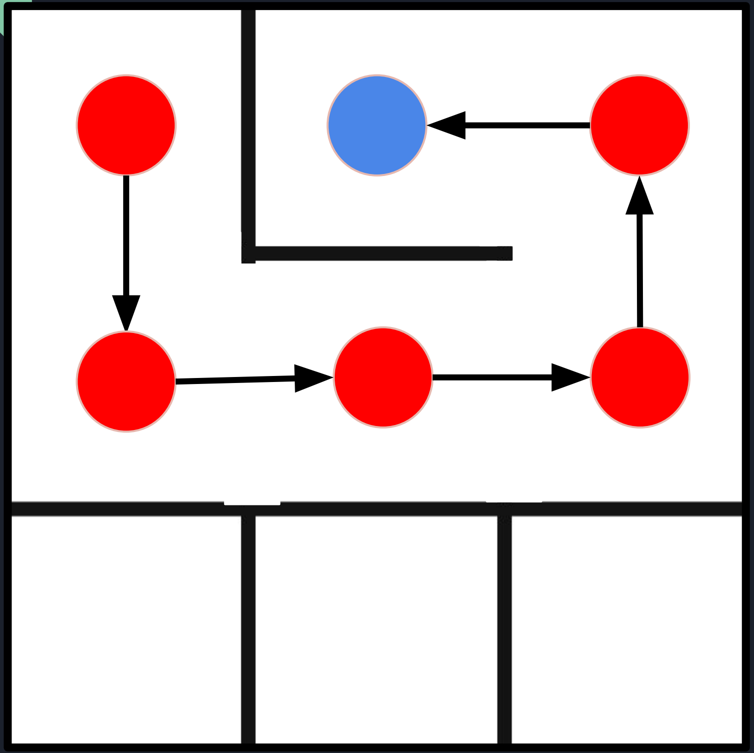
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet
  - Remove the wall between them
  - Set new cell to current
  - Repeat until we reach a dead end

Maze Generation



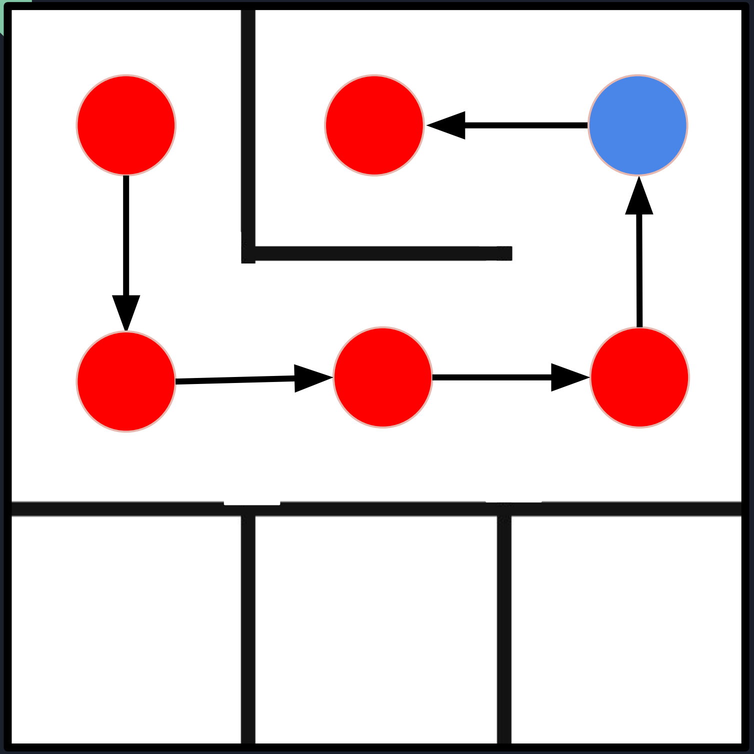
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet
  - Remove the wall between them
  - Set new cell to current
  - Repeat until we reach a dead end

Maze Generation



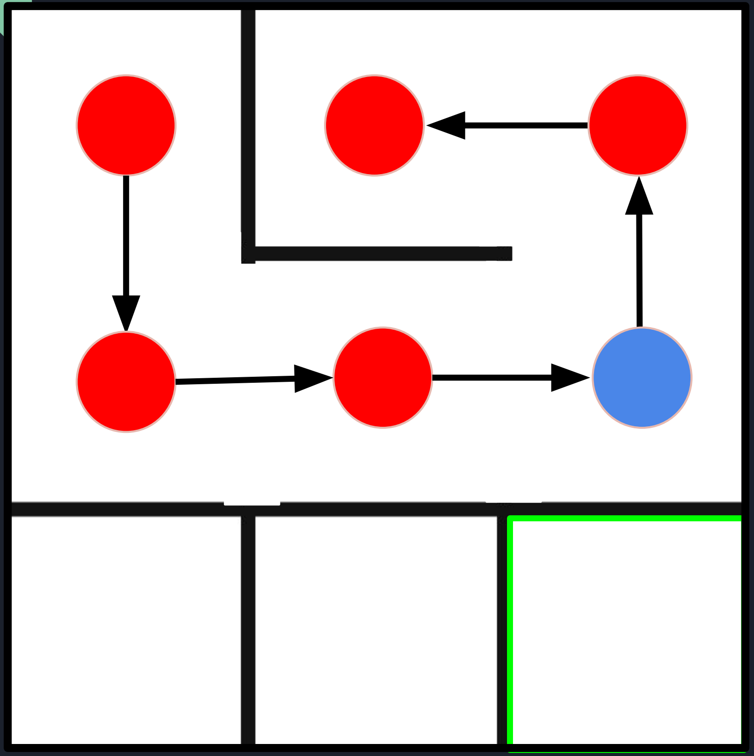
- Set a random cell as the current cell
- Check its neighbors
    - Pick a random neighbor we haven't gone to yet
    - Remove the wall between them
    - Set new cell to current
    - Repeat until we reach a dead end
- At dead end, backtrack until we find a cell with a neighbor we haven't gone to yet.

Maze Generation



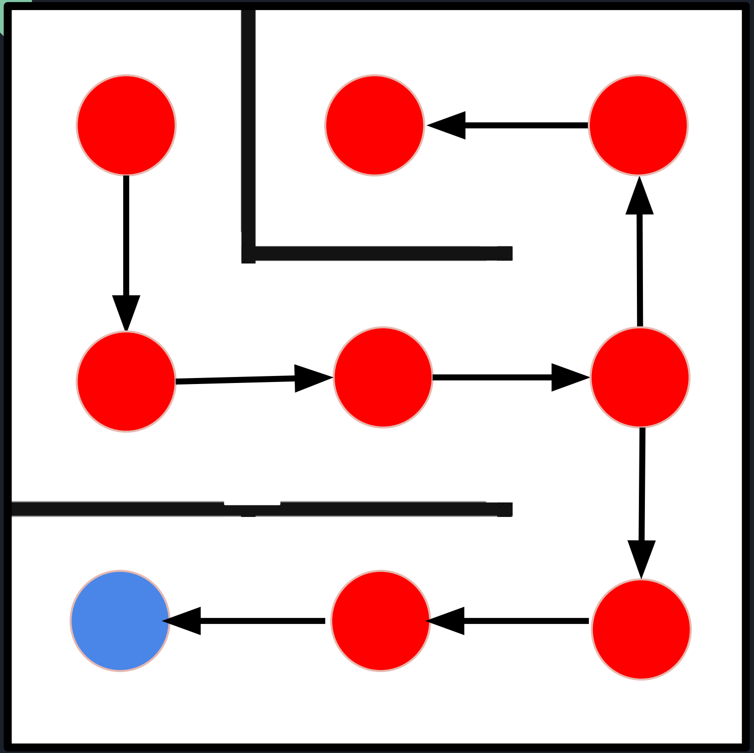
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet
  - Remove the wall between them
  - Set new cell to current
  - Repeat until we reach a dead end
- At dead end, backtrack until we find a cell with a neighbor we haven't gone to yet.

Maze Generation



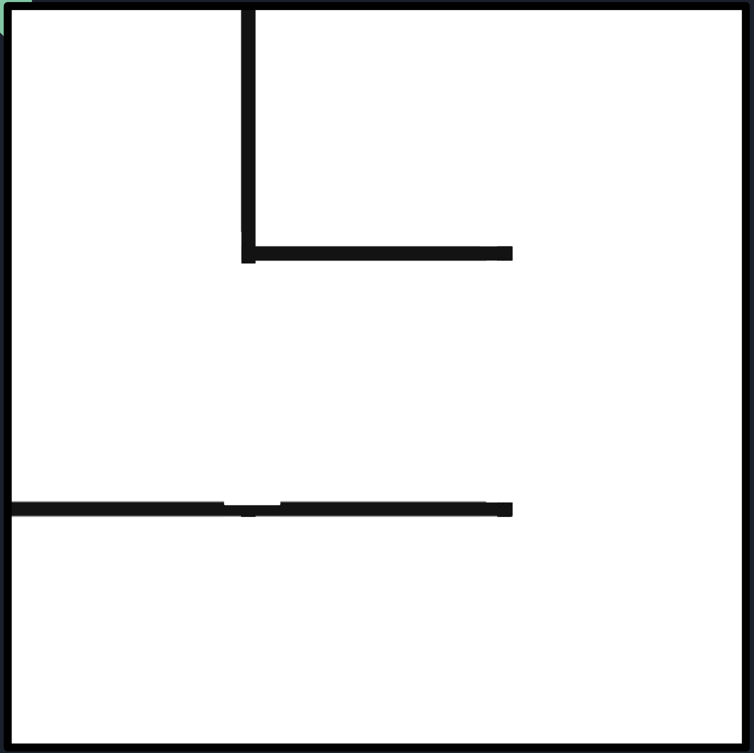
- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet
  - Remove the wall between them
  - Set new cell to current
  - Repeat until we reach a dead end
- At dead end, backtrack until we find a cell with a neighbor we haven't gone to yet.
- Repeat from step 2 until we visit every node

Maze Generation



- Set a random cell as the current cell
- Check its neighbors
  - Pick a random neighbor we haven't gone to yet
  - Remove the wall between them
  - Set new cell to current
  - Repeat until we reach a dead end
- At dead end, backtrack until we find a cell with a neighbor we haven't gone to yet.
- Repeat from step 2 until we visit every node

Maze Generation



- Wow! What a fun maze!

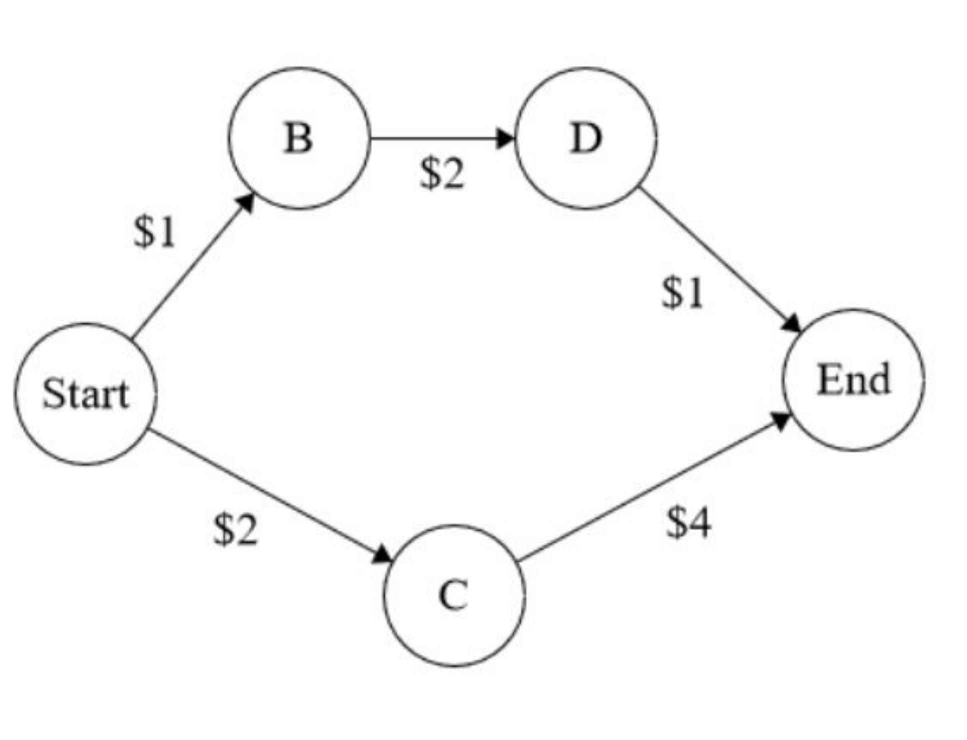
Maze Generation Pseudo-code

- Choose an initial cell, mark it as visited and push it to the stack
- While the stack is not empty
  - Pop a cell from the stack and mark it as the current cell
  - If the current cell has any neighbors which have not been visited
    - Push the current cell to the stack
    - Randomly choose one of the unvisited neighbors
    - Remove the wall between the current cell and the chosen cell
    - Mark the chosen cell as visited and push it to the stack

Dijkstra’s Algorithm

- Very similar to Breadth First Search, but the strategy is slightly different
- New Strategy : Expand the *cheapest* shallow node first
  - Each node has a cost
  - Choose node with the least-cost path to it
  - Expanded nodes include previous node's cost
- Uniform Cost Search
  - Variation of Dijkstra's
  - Insert nodes into fringe only when encountered

UCS Example *(using $ as cost)*



Current node: fringe

Start: [B ($1), C ($2)]

choose B for $1 and expand nodes (include cost of B; e.g. D = $2 + $1)

B: [C ($2), D ($3)]

choose C for $2 and expand nodes

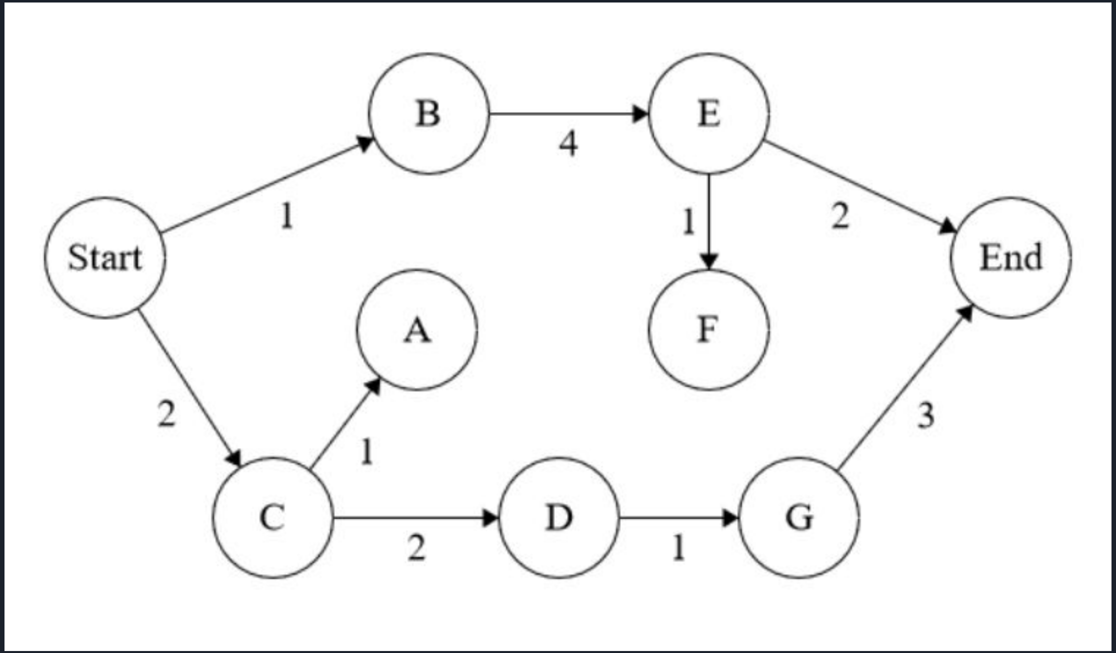
C: [D ($3), End ($6)]

choose D for $3 and expand nodes

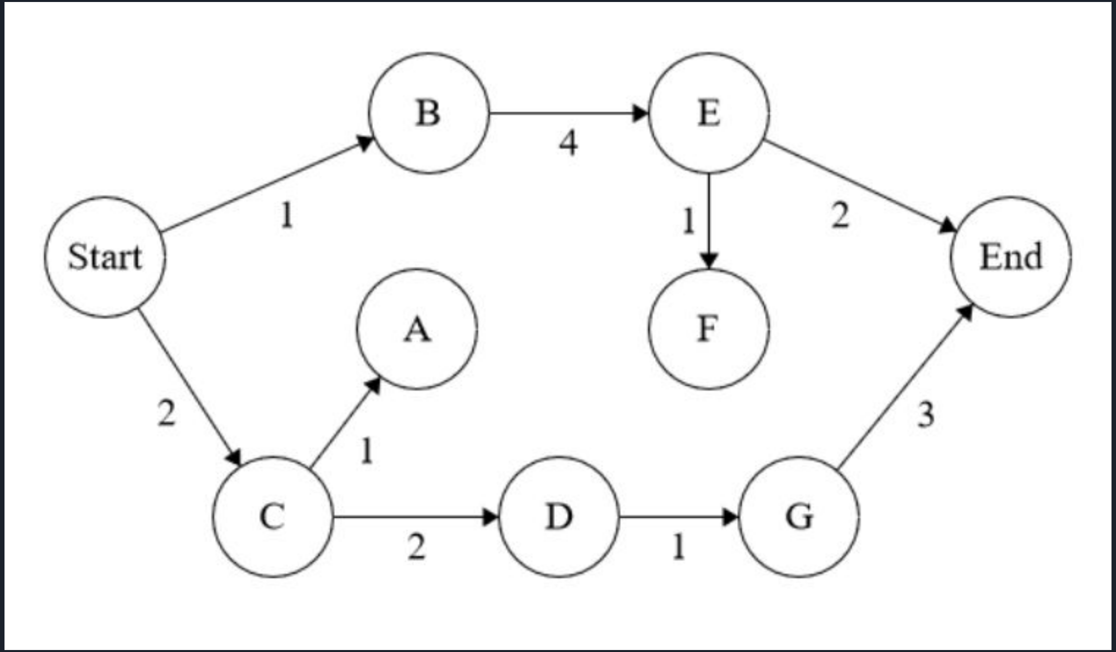
D: [End ($4), End ($6)]

choose End for $4 - this is the cheapest cost path!

UCS Exercise:
What order are the nodes explored in?
Ties are settled alphabetically.



Correct Answer:
Start-B-C-A-D-E-G-F-End



Current node: fringe
Start: [B (1), C (2)]
B: [C (2), E (5)]
C: [A (3), D (4), E (5)]
A: [D (4), E (5)]
D: [E (5), G (5)]
E: [G (5), F (6), End (7)]
G: [F (6), End (7), End(8)]
F: [End (7), End(8)]