

# Uninformed Search

Scott Wallace

CS 440

WSU Vancouver

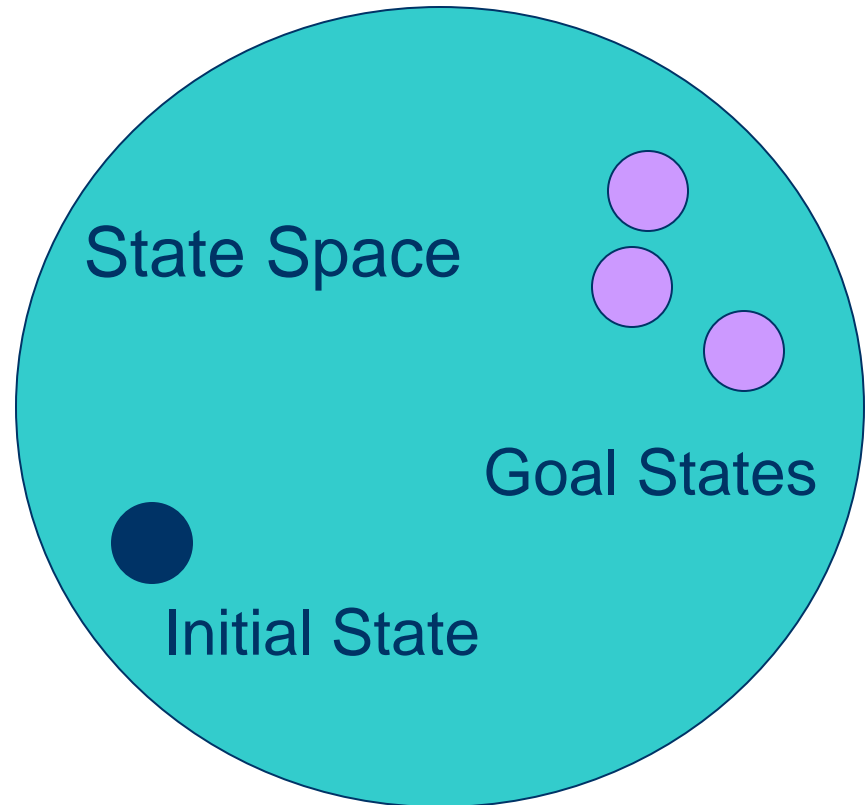


# Outline

- State Space Search
- Search Trees
- Important Properties of Search
- Uninformed Search
  - Breadth First
  - Depth First
  - Depth Limited / Iterative Deepening
  - Uniform Cost

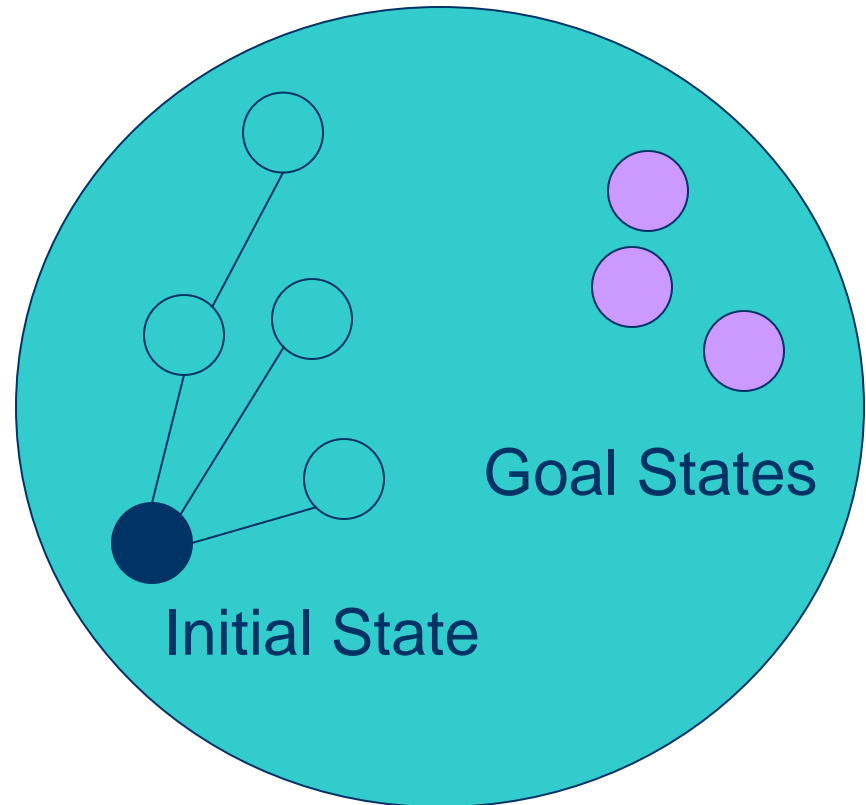
# State Space Search

- Potential world states are contained in a set
- Path (solution) leads through the state space



# State Space Search

- Potential world states are contained in a set
- Path (solution) leads through the state space



# Search Trees

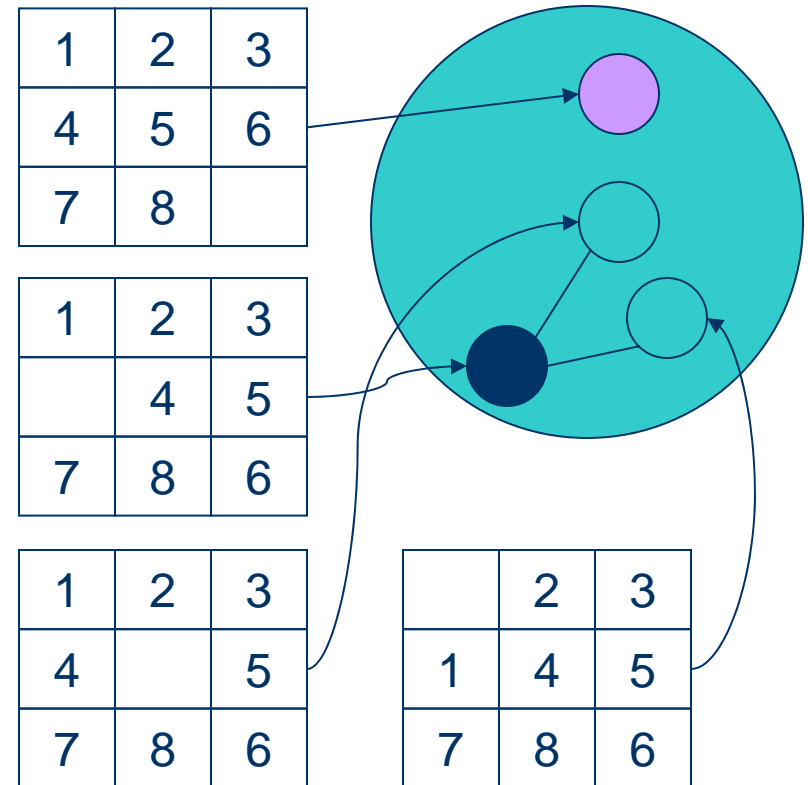
- Search generates a **tree**, even though the search space is a graph
- Nodes in tree are data-structures often containing:
  - A representation of a state in the search space
  - Additional information to help the search
- Root node corresponds to initial state
- Nodes are **expanded** with the successor function to generate new states (and thereby new nodes)
- Order of expansion is defined by **strategy**

# Search Trees

- Search trees are often infinite
- State spaces may be finite (small, if we're lucky)
- Often we want nodes to contain more than a representation of the state they correspond to:
  - Parent node
  - Action
  - Path Cost from initial state – usually called  $g(n)$
  - Depth

# Consider Eight Puzzle

- Configurations of puzzle are elements in state space
- State space is finite
- Search Tree is infinite



# Tree Search

**TreeSearch**( *problem*, *strategy* )

initialize search tree with initial state of *problem*

**loop**

**if** no candidates for expansion, **return** *failure*

choose a leaf for expansion based on *strategy*

**if** node corresponds to a goal state,

**return** *solution*

**else** expand node and add successors to tree

# Implementing Tree Search

- Need to track expansion candidates
  - Typically we use a queue, often a priority queue
  - Search strategy defines queue's ordering
- Queue often called **open-list**, or **fringe**
- Expanding takes one node off queue and puts its successor nodes onto the queue

# Comparing Search Strategies

- **Completeness**
  - Will a solution be found if there is one?
- **Optimality**
  - Will the optimal solution be found?
- Time Complexity
- Space Complexity

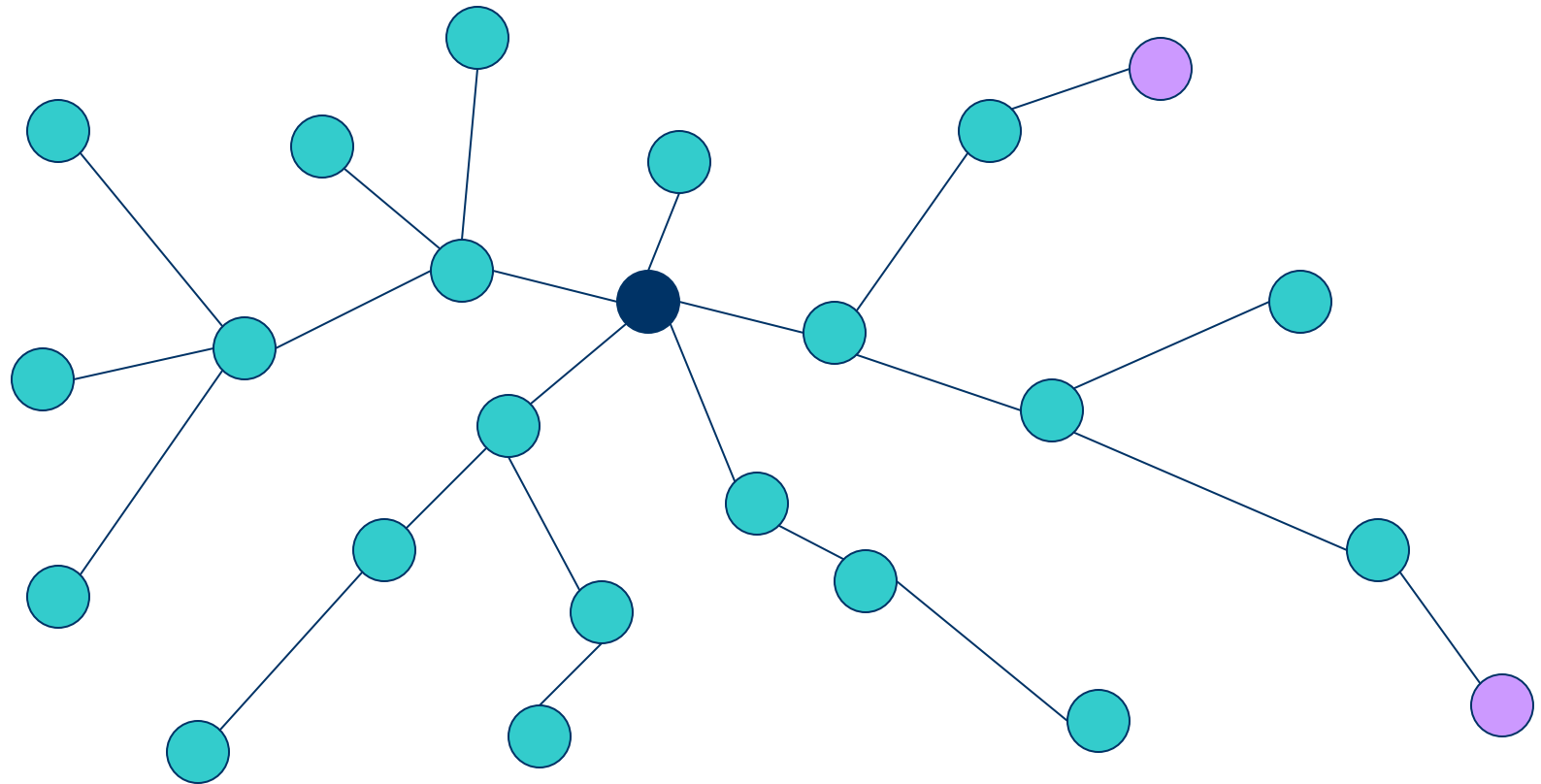
# Uninformed Search

- Requires minimal knowledge about the search space
  - Often called **blind search**
  - The weakest search methods
  - Sometimes, it's the best you can do

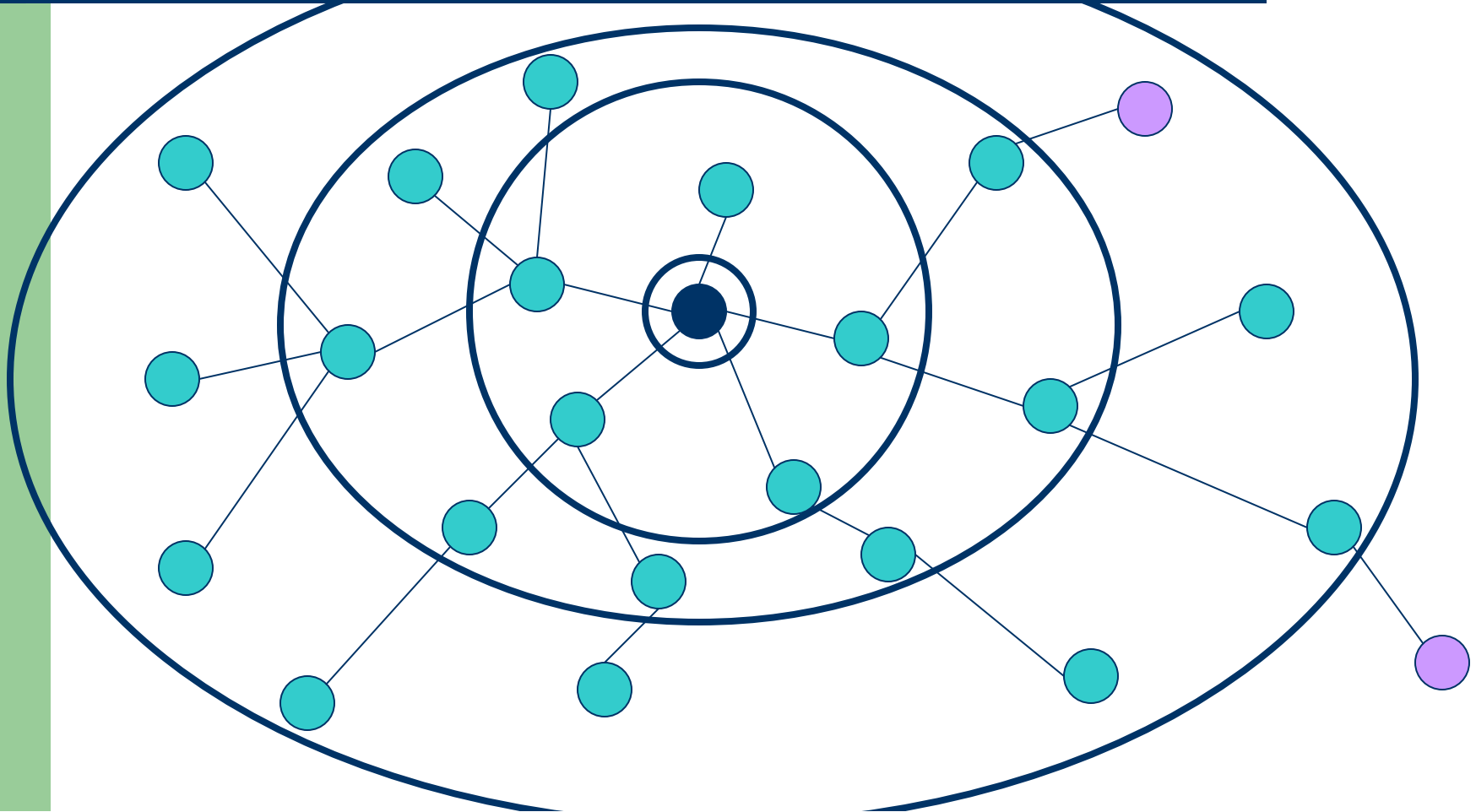
# Breadth First

- Expand away from the root in concentric circles, or plies
  - All nodes that are 1 step away from the root are expanded before any node that is 2 steps away
  - Open-list is a simple FIFO queue

# Order of Expansion



# Order of Expansion



# Breadth First Search

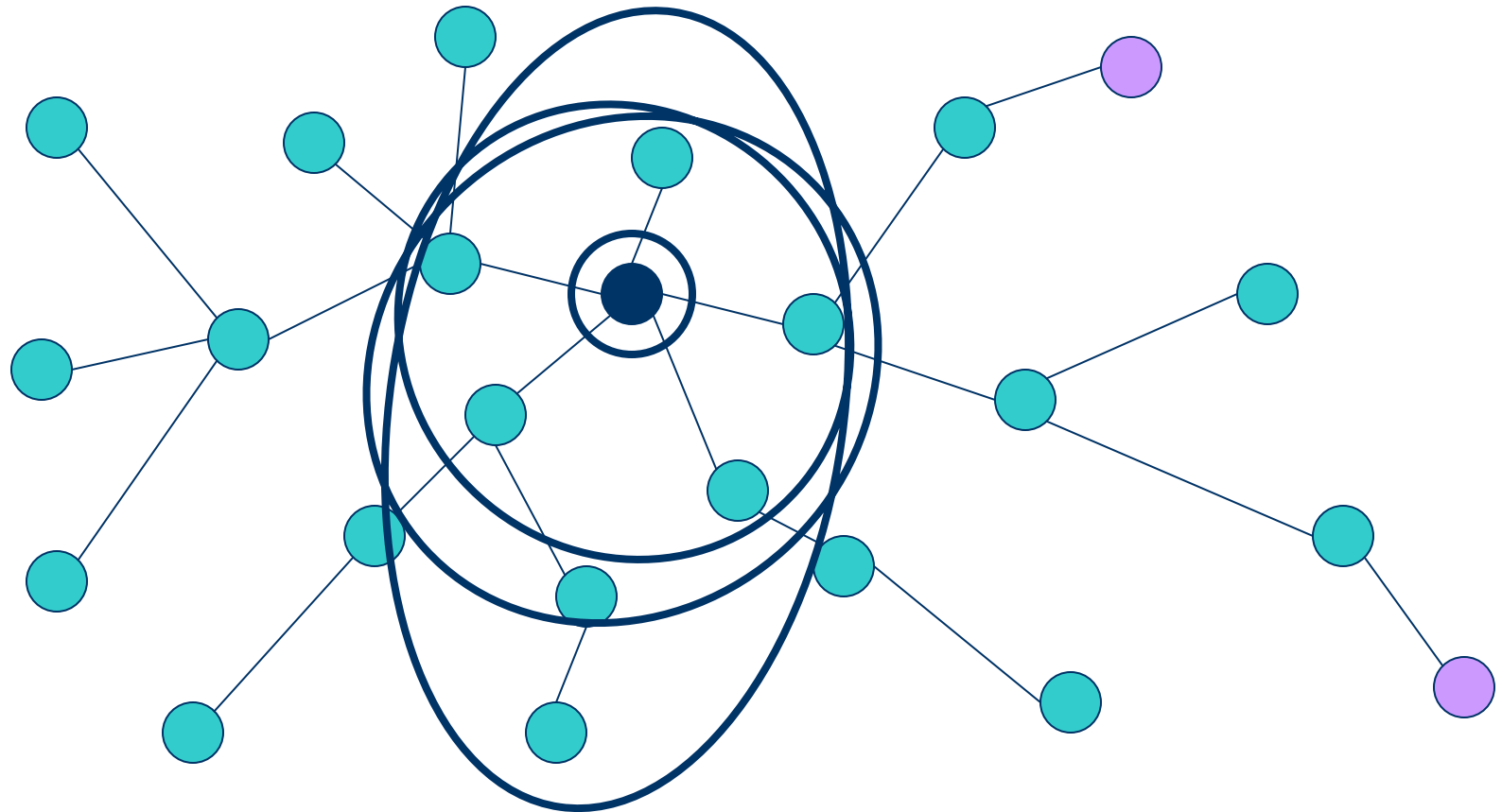
- Complete
- Finds shallowest goal
  - may be optimal, may not
- Open list can be quite large
  - $O(b^{d+1})$  space complexity
- Time is equally slow
  - $O(b^{d+1})$  time complexity

# Depth First Search

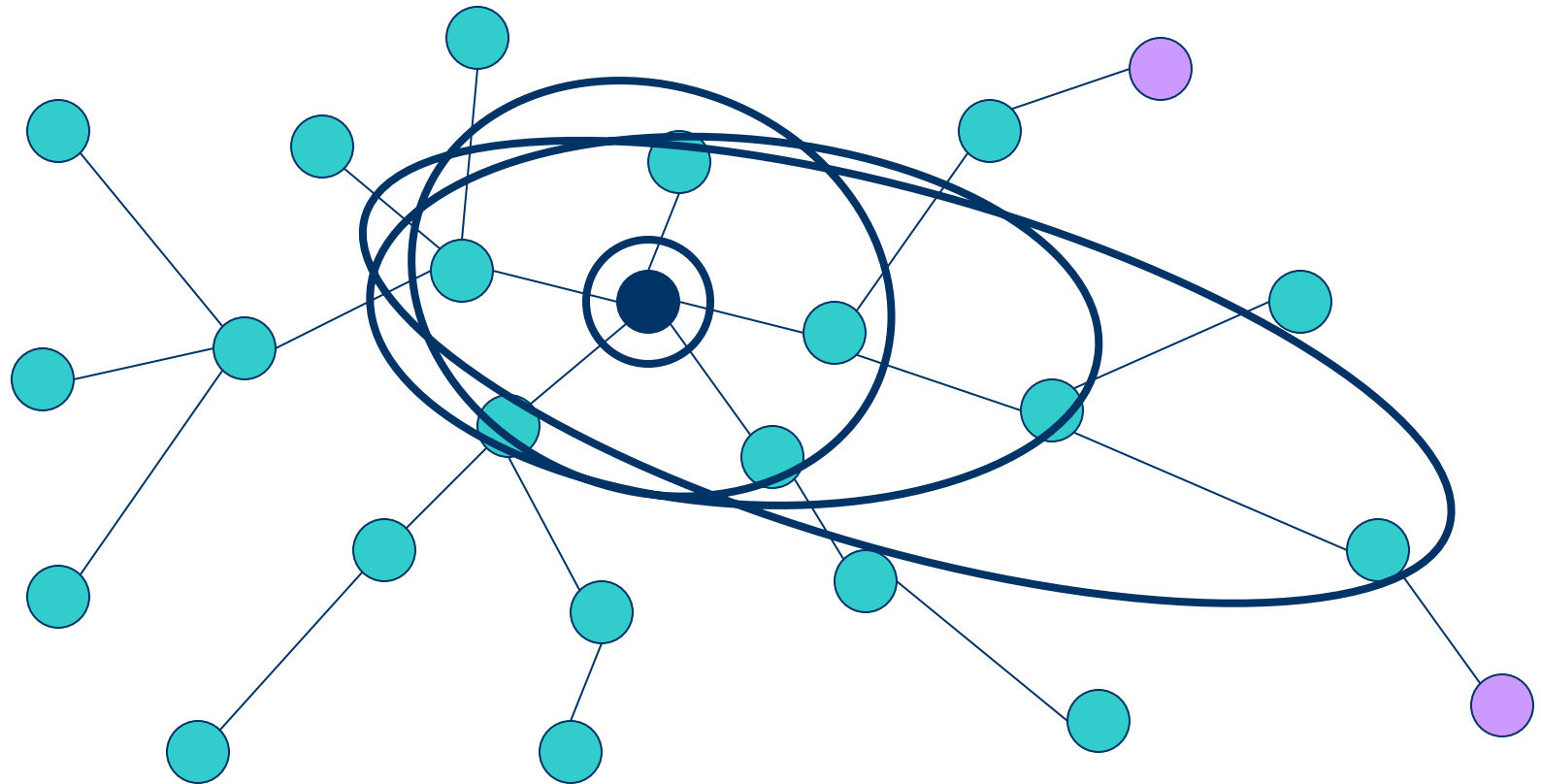
- Expand away from the root in long branchy paths
  - Pick a node to expand, follow a single, deep path until the goal is reached or it terminates
  - Then backtrack to the most recent choice point and try again
  - Open-list is LIFO queue (stack)



# Order of Expansion



# Order of Expansion



# Depth First Search

- Not Complete
  - May get stuck on an infinite path
- Not Optimal
  - Unclear where the goal will be found
- Open list is relatively restricted
  - $O(bm)$  space complexity
- Time is slow, perhaps slower than BFS
  - $O(b^m)$  time complexity

# Depth Limited DFS

- Main Idea:
  - Prevent DFS from getting stuck on an infinite path by using a depth limit
- Implementation:
  - If a node is at the depth limit, don't put its successors on the open-list
- Caveats:
  - Need to choose depth limit wisely

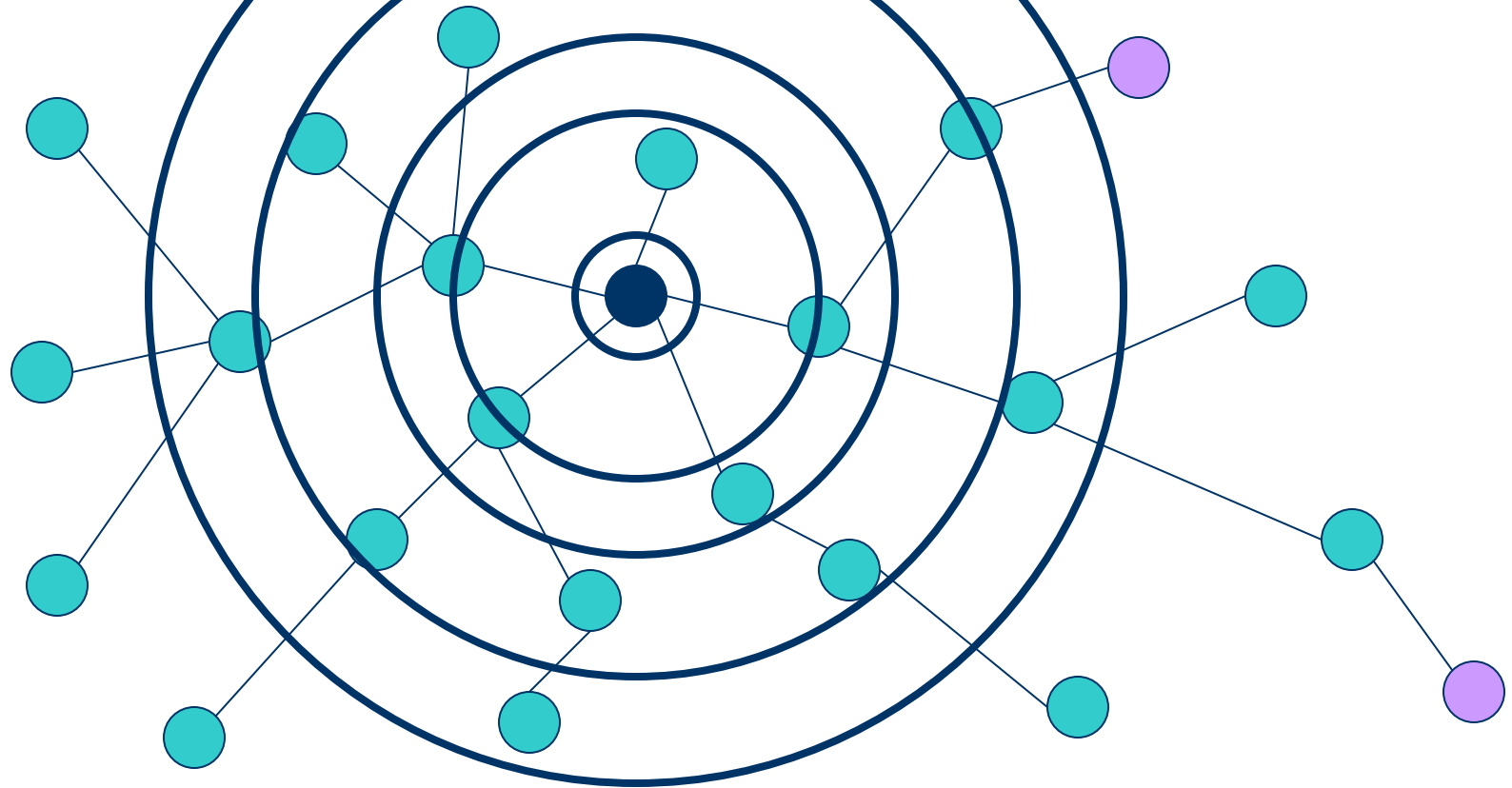
# Iterative Deepening

- If you don't know how to choose the depth limit, try incrementally increasing values
- Sounds a lot worse than it really is!
  - **IDS:**  $db + (d-1)b^2 + (d-2)b^3 + \dots + (1)b^d$
  - **BFS:**  $b + b^2 + b^3 + \dots + b^d + b^{d+1} - b$
- Typically, IDS is preferred over BFS and DFS

# Uniform Cost Search

- Breadth First is optimal if shortest path is best
- What about more general measures?
- We can associate a cost with each action (recall the path cost function from before)
- We can then search in

# Order of Expansion



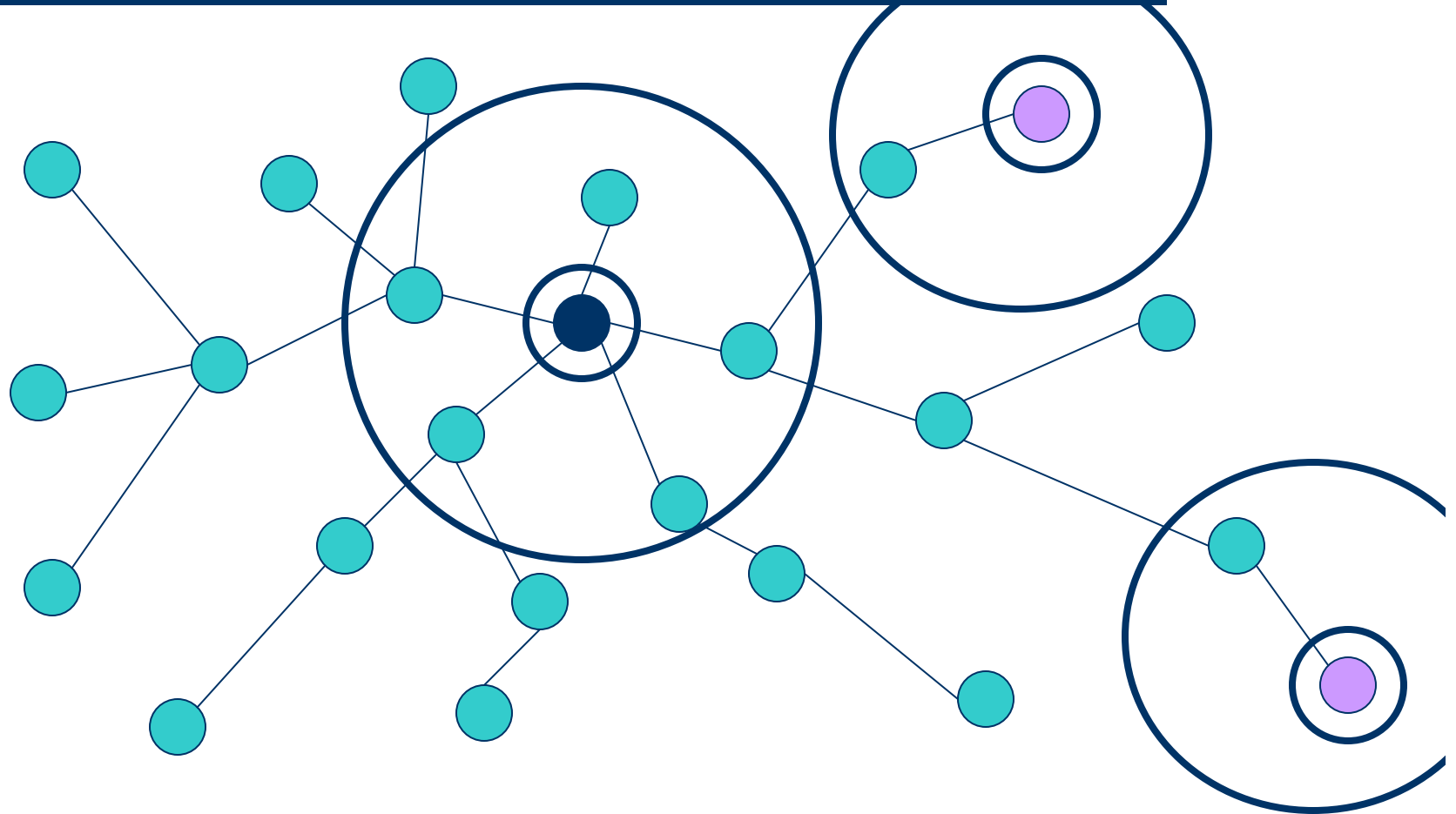
# Uniform Cost Search

- Complete
  - Will find a solution if it exists at finite depth
- Optimal
  - Expansion order guarantees optimal solution
- Complexity difficult to pin down,
  - $O(b^{1 + (C/e)})$  space complexity
- Time also difficult to pin down
  - $O(b^{1 + (C/e)})$  time complexity

# Bidirectional Search

- In some cases, we can search from goal to initial state (as well as the other direction)
- Difficulties:
  - Checking for contact in the ‘middle’
  - Multiple goal states
  - Calculating inverse actions & state (predecessors)

# Order of Expansion



# Bi-Directional Search

- Complete
  - If we use breadth first from both directions
- Optimal
  - With same caveats as breadth first
- Complexity better than breadth first,
  - $O(b^{d/2})$  space complexity
- Time is also better
  - $O(b^{d/2})$  time complexity

# Repeated States

- Search tree can visit the same 'state' in state space multiple times (via different paths)
- We know this state can't:
  - Be the goal (we've already tested)
  - Lead us to the goal (we've already gone that way)
- We want to avoid these loops
  - Check for duplicate states using a closed list

# Closed List

- Basic idea:
  - Keep a list of states we've visited, and don't go to those again.
- Caveats:
  - Basic idea works for Breadth First Search
  - What about Depth First? Iterative Deepening?
  - Closed list will affect complexity

# Graph Search

**GraphSearch**( *problem*, *strategy* )

closed-list = [ ]

initialize open-list with initial state of *problem*

**loop**

**if** no candidates for expansion, **return** *failure*

choose a leaf for expansion based on *strategy*

**if** node corresponds to a goal state,

**return** *solution*

**if** node(state) **not** on closed-list

add node(state) to closed-list

expand node and add successors to open-list