Example: The 8-puzzle

```
1 2 3
8 4
7 6 5
```

```
1 2 3
4 5 6
7 8
```
Example: Route Planning

Example: N Queens

4 Queens
**Example: N Queens**

4 Queens

**State-Space Search Problems**

**General problem:**
Given a *start state*, find a path to a *goal state*

- Can test if a state is a goal
- Given a state, can generate its *successor* states

**Variants:**

- Find any path *vs.* a least-cost path
- Goal is completely specified, task is just to find the path
  - Route planning
- Path doesn’t matter, only finding the goal state
  - 8 puzzle
Tree Representation of 8-Puzzle Problem Space

Implementation: general tree search

```c
function Tree-Search(problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-Test[problem] applied to State(node) succeeds return node
        fringe ← INSERT-ALL(Expand(node, problem), fringe)
```

Implementation: general tree search

```
def Tree-Search(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem] applied to State(node) succeeds return node
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

```
def EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN(problem)(STATE[node]) do
    s ← a new Node
    PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
```

Implementation: states vs. nodes

A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree
  includes parent, children, depth, path cost \( g(x) \)

States do not have parents, children, depth, or path cost!
Implementation: states vs. nodes

A state is a (representation of) a physical configuration. A node is a data structure constituting part of a search tree. It includes parent, children, depth, path cost $g(x)$. States do not have parents, children, depth, or path cost!

The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.

Search strategies

A strategy is defined by picking the order of node expansion.

Strategies are evaluated along the following dimensions:
- completeness—does it always find a solution if one exists?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of
- $b$—maximum branching factor of the search tree
- $d$—depth of the least-cost solution
- $m$—maximum depth of the state space (may be $\infty$)
Uninformed search strategies

*Uninformed* strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

**Implementation:**

*fringe* is a FIFO queue, i.e., new successors go at end

![Breadth-first search tree](image)
Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end

```
  A
 / \   /
B   C
 |   /
D   E
```

Properties of breadth-first search

*Complete??*
Properties of breadth-first search

Complete?? Yes (if \( b \) is finite)

Time??

\[ 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}), \text{ i.e., exp. in } d \]

Space??
Properties of breadth-first search

**Complete??** Yes (if \( b \) is finite)

**Time??** \[ 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}), \text{i.e., exp. in } d \]

**Space??** \( O(b^{d+1}) \) (keeps every node in memory)

**Optimal??**

---

**Space is the big problem for BFS.**

**Example:** \( b = 10, \text{ 10,000 nodes/sec, 1KB/node} \)

\( d = 2 \) \( \downarrow \) \( 1100 \text{ nodes, 0.11 secs, 1MB} \)

\( d = 4 \) \( \downarrow \) \( 111,100 \text{ nodes, 11 secs, 106 MB} \)

\( d = 8 \) \( \downarrow \) \( 10^8 \text{ nodes, 31 hours, 1 TB} \)
Uniform-cost search

Expand least-cost unexpanded node

**Implementation:**
\[ fringe = \text{queue ordered by path cost} \]

Equivalent to breadth-first if step costs all equal

**Complete??** Yes, if step cost \( \geq \epsilon \)

**Time??** \# of nodes with \( g \leq C^* \) cost of optimal solution, \( O(b^{C^*/\epsilon}) \)
where \( C^* \) is the cost of the optimal solution

**Space??** \# of nodes with \( g \leq C^* \) cost of optimal solution, \( O(b^{C^*/\epsilon}) \)

**Optimal??** Yes—nodes expanded in increasing order of \( g(n) \)

---

Depth-first search

Expand deepest unexpanded node

**Implementation:**
\[ fringe = \text{LIFO queue, i.e., put successors at front} \]

---
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ fringe = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:

\[fringe = \text{LIFO queue, i.e., put successors at front}\]
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

```
A
  / \  
B   E
  /   /\  
C   F   G
     /\   
    L  M
   / \   /
  N   O
```
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

**Implementation:**

fringe = LIFO queue, i.e., put successors at front

Properties of depth-first search

**Complete??**
Properties of depth-first search

**Complete??** No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  ⇒ complete in finite spaces

**Time??**

$O(b^m)$: terrible if $m$ is much larger than $d$
  but if solutions are dense, may be much faster than breadth-first

**Space??**
Properties of depth-first search

**Complete??** No: fails in infinite-depth spaces, spaces with loops
   Modify to avoid repeated states along path
     ⇒ complete in finite spaces

**Time??** $O(b^m)$: terrible if $m$ is much larger than $d$
   but if solutions are dense, may be much faster than breadth-first

**Space??** $O(bm)$, i.e., linear space!

**Optimal??** No
Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if Goal-Test[problem][State[node]] then return node
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```
Iterative deepening search \( l = 0 \)

\[ \text{it} = 0 \]

Iterative deepening search \( l = 1 \)

\[ \text{it} = 1 \]
Iterative deepening search \( l = 2 \)

Iterative deepening search \( l = 3 \)
Properties of iterative deepening search

Complete?? Yes
Time??
Properties of iterative deepening search

**Complete?** Yes

**Time?** \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)

**Space?** \(O(bd)\)

**Optimal?**
Increasing path-cost limits instead of depth limits
This is called Iterative lengthening search (exercise 3.11)
Forwards vs. Backwards

Problem: Find the shortest route

Bidirectional Search

Motivation: $b^{d/2} + b^{d/2} << b^d$
Can use breadth-first search or uniform-cost search
Hard for implicit goals e.g., goal = “checkmate” in chess
Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one! (e.g., repeated states in 8 puzzle)

Graph search algorithm: Store expanded nodes in a set called closed and only add new nodes to the fringe

Graph Search

```plaintext
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(Make-Node(INITIAL-STATE(problem)), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem)(STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(Expand(node, problem), fringe)
```

56
Can we do better?

All these methods are slow (blind)

Solution use problem-specific knowledge to guide search ("heuristic function")
"informed search" (next lecture)