Introduction to Problem Solving and Search

References:

David Poole et al., *Computational Intelligence: A Logical Approach*, Oxford University Press, 1998. (Chapter 4)

Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall International, 1995. (Chapter 3)

Steps in Solving a Problem

Goal formulation — deciding on appropriate goal(s) to achieve in order to solve problem

Problem formulation — decision as to which actions and states to consider.

Search — examine different possible sequences of actions leading to states of known value and select best such sequence.

Solution — action sequence leading to goal.

Execution — carry out actions.

Search algorithm — input: problem; output: solution

Problem Types

Single-state — agent starts in known world state and knows which unique state it will be in after a given action

Multiple-state — limited access to world state means agent is unsure exactly which world state it is in but may be able to narrow it down to a set of states

Contingency problem — if agent does not know full effects of actions (or there are other things going on) it may have to sense during execution

Exploration problem — no knowledge of effects of actions (or state). Agent must experiment.

Search methods are capable of tackling single-state and multiple-state problems.

Defining Problems — Single-State

Problem — collection of information that agent uses to decide what to do

Initial state — world state agent knows itself to start in

Operators — set of possible actions at agent's disposal; describe state reached after performing action in current state

(Alternatively) *successor function* — $S_A(x)$= set of states reached from state x by performing any single action A

State space — set of all states reachable from initial state by any action sequence

Path (in state space) — sequence of actions leading from one world state to another

Goal test — test that can be applied to a world state to determine whether it is a goal state

Path cost function — assigns cost to a path so that agent can determine best solution

To define a problem we need to specify: initial state, operators, goal test, path cost function

Search algorithm takes problem as input and outputs solution (a path from the initial state to a state satisfying the goal test)

Defining Problems — Multiple-State

Initial state set

Operators — specify set of states reached from any state

Goal test, Path cost function — as above

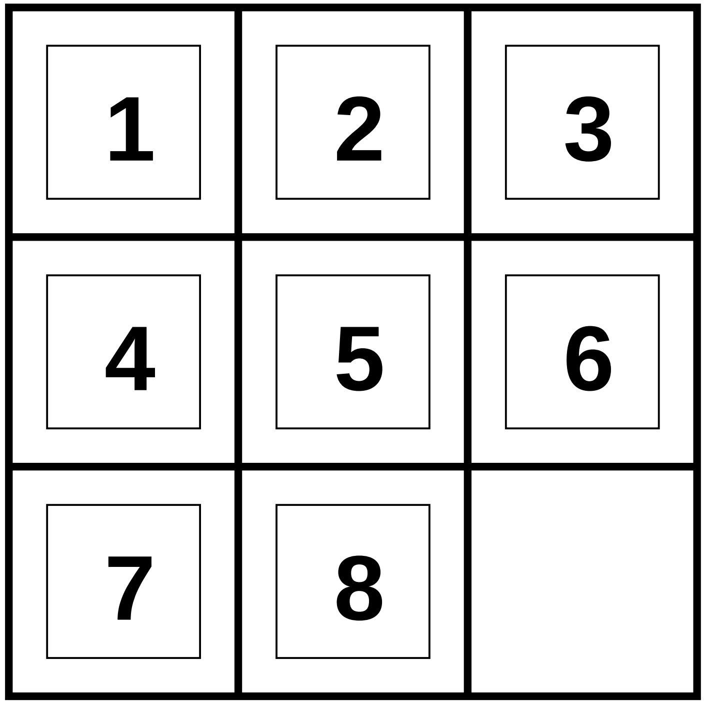
Operator is applied to a set of states by taking each state in turn and taking the union of the results

Path connects sets of states

Solution — path leading to set of states all of which satisfy goal test

State space — becomes state space set

Example Problem — 8-Puzzle



States: location of eight tiles plus location of blank

Operators: move blank left, right, up, down

Goal test: tiles arranged in sequence

Path cost: each step is of cost 1

Determining Effectiveness of Search

Does it find a solution?

Is it a good solution (low path cost)?

What is the search cost (time + memory)?

Total cost = path cost + search cost

Example Real-World Problems

Route finding — robot navigation, airline travel planning, computer networks

Travelling salesman problem — planning movement of automatic circuit board drills

VLSI layout — design silicon chips

Assembly sequencing — automatic assembly of complex objects

Search Strategies

Search — enumeration of potential partial solutions to a problem so that you can verify whether they are a solution or lead to a solution?

Uninformed (blind) search — no information about steps required or path cost

Informed (heuristic) search — makes use of additional information

Evaluation Criteria

Completeness: strategy guaranteed to find a solution when one exists?

Time complexity: how long to find a solution?

Space complexity: memory required during search?

Optimality: when several solutions exist, does it find the “best”?

Graph Searching

Many tasks can be viewed in terms of finding a path through a graph

State-space graph — node represents world state; arc represents change from one state to another

Problem-space graph — node represents problem; arcs represent alternate decompositions of problem

Graph Terminology - A Quick Tour

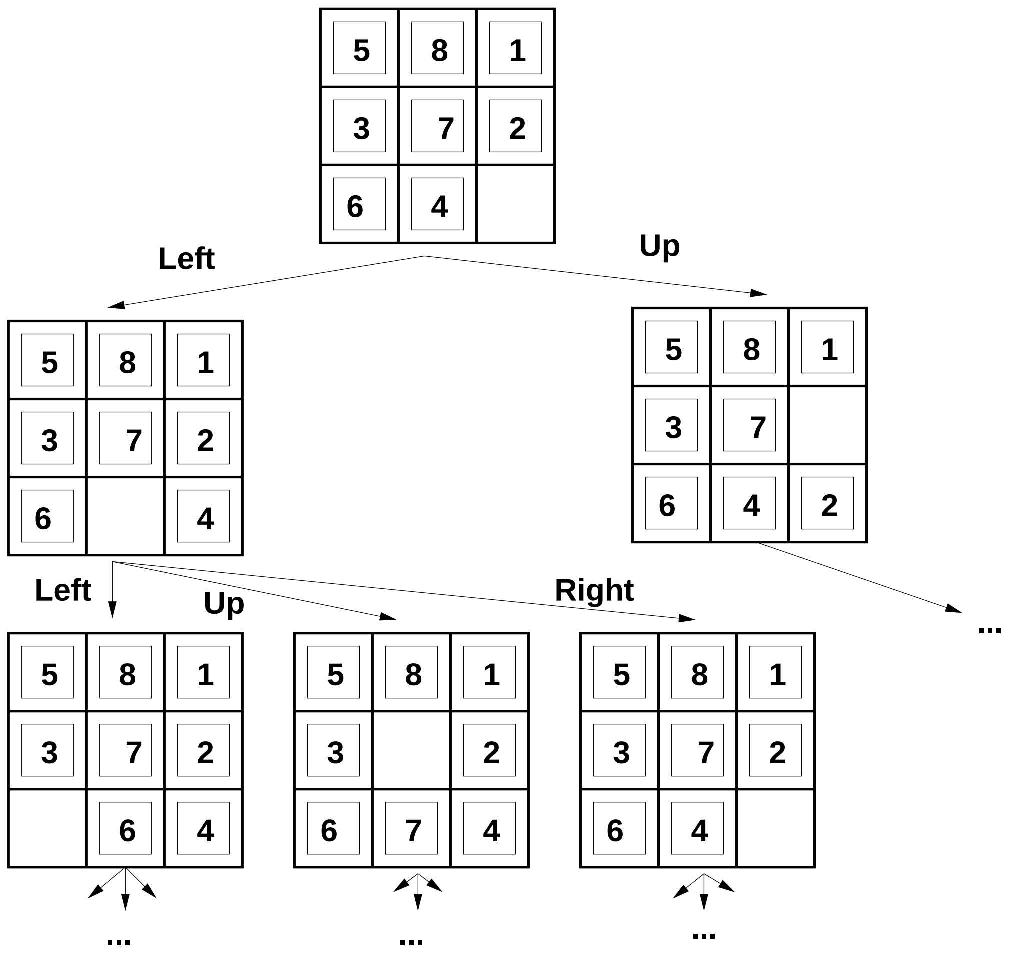
Graph $G = (V,\ E)$ — V: vertices (nodes); E: edges

Path from node s to g — sequence of nodes $s = n_0,\ n_1,\ \ldots, n_k = g$ such that n_{i-1} is connected to n

Edges may have an associated *cost*. *Path cost* = sum of edge costs in path

Forward (Backward) branching factor — # out-(in-)going arcs from node

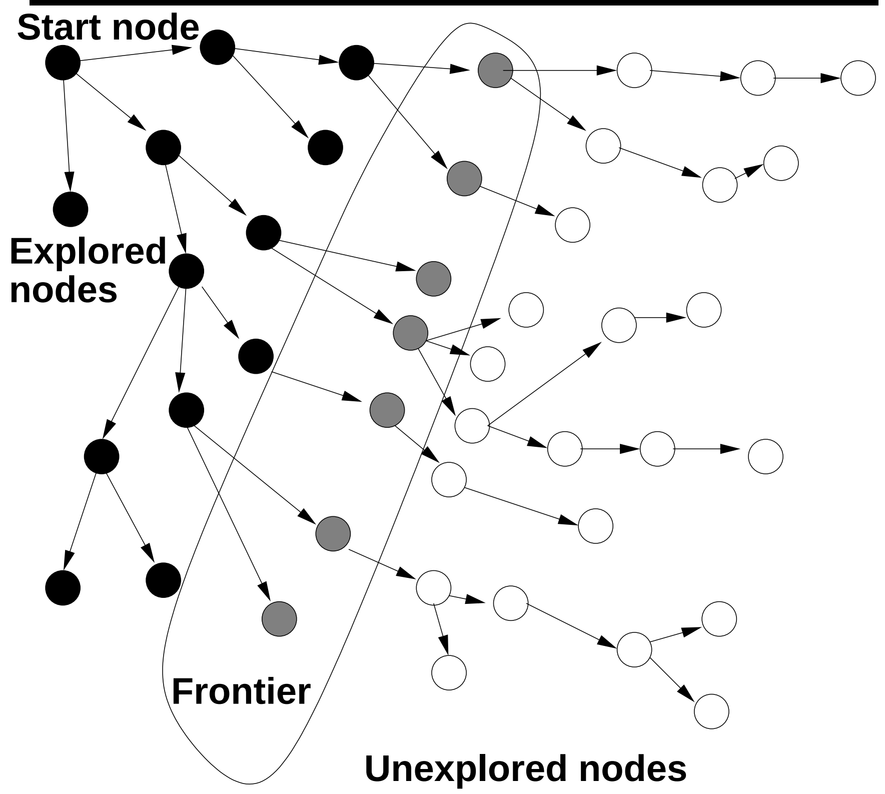
Search Graph — 8-Puzzle



A General Search Procedure

```
function GeneralSearch(problem, strategy) returns a solution or failure
    initialise search graph using the initial state of problem
    loop
        if there are no candidates for expansion then return failure
        choose a frontier node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search graph
end
```

A General Search Procedure



Search strategy — way in which frontier expands

See PMG pp. 119-120 for Prolog-like code.

$g(n)$ — cost of path to node n

Uninformed (Blind) Search

Many problems are amenable to attack by search methods

We begin by examining search methods that have no problem-specific knowledge to use as guidance

Breadth-First Search

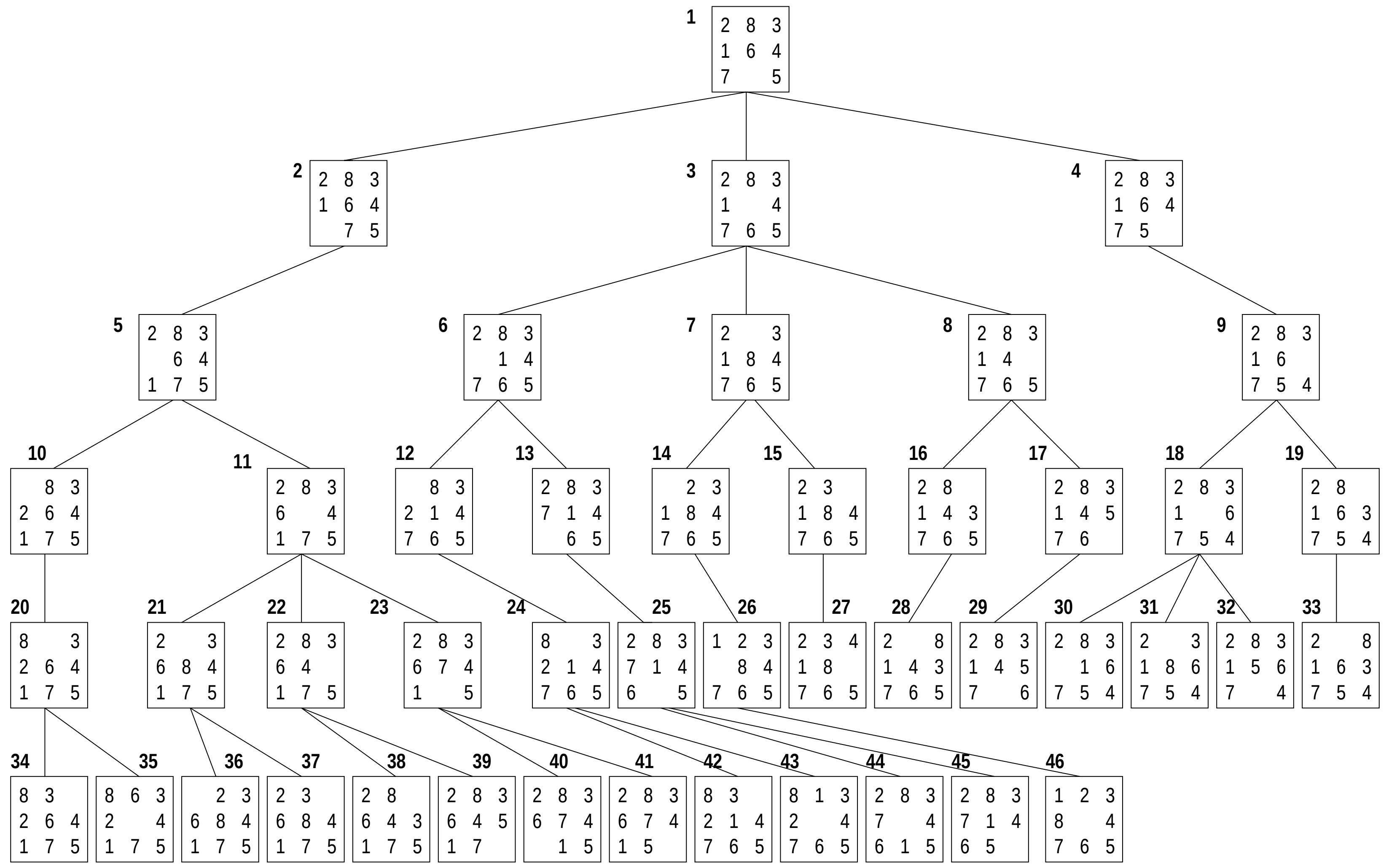
Idea: Expand root node, then expand all children of root, then expand their children, . . .

All nodes at depth d are expanded before nodes at $d + 1$

Can be implemented by using a queue to store frontier nodes

Breadth-first search finds shallowest goal state

Breadth-First Search



Breadth-First Search — Analysis

Complete

Optimal — provided path cost is nondecreasing function of the depth of the node

Maximum number of nodes expanded: $1 \, + \, b \, + \, b^2 \, + \, b^3 \, + \, \ldots + \, b^d$
(where b = forward branching factor; d = path length to solution)

Time and space requirements are the same $O(b^d)$.

Uniform Cost Search

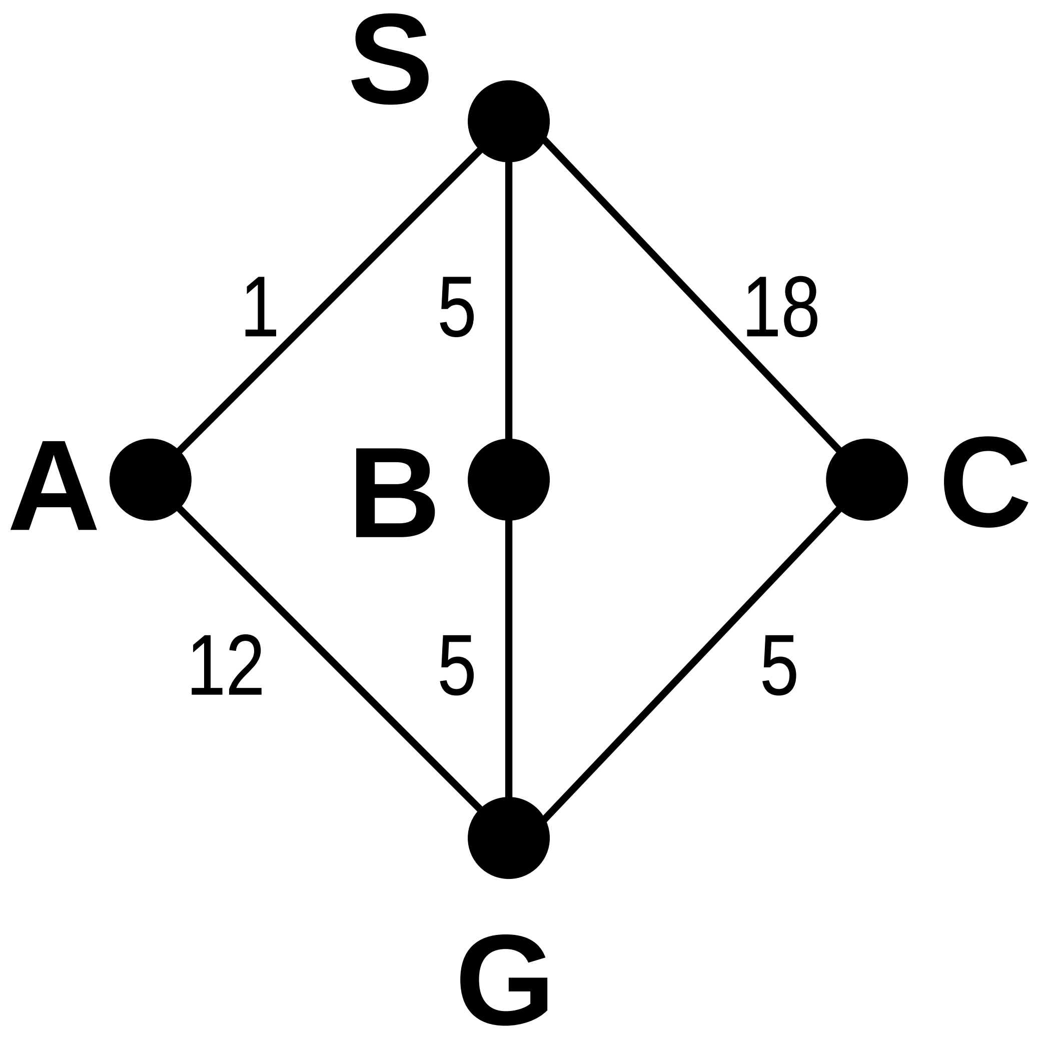
Also known as Lowest-Cost-First search

Shallowest goal state may not be the least-cost solution

Idea: Expand lowest cost (measured by path cost $g(n)$) node on the frontier

Breadth-first search = uniform cost search where $g(n) = depth(n)$

Uniform Cost Search



Uniform Cost Search — Analysis

Complete

Optimal — provided path cost does not decrease along path (i.e., $g(successor(n)) \geq g(n)$ for all n)

Reasonable assumption when path cost is cost of applying operators along the path

Performs like breadth-first search when $g(n) = depth(n)$

If there are paths with negative cost we would need to perform an exhaustive search

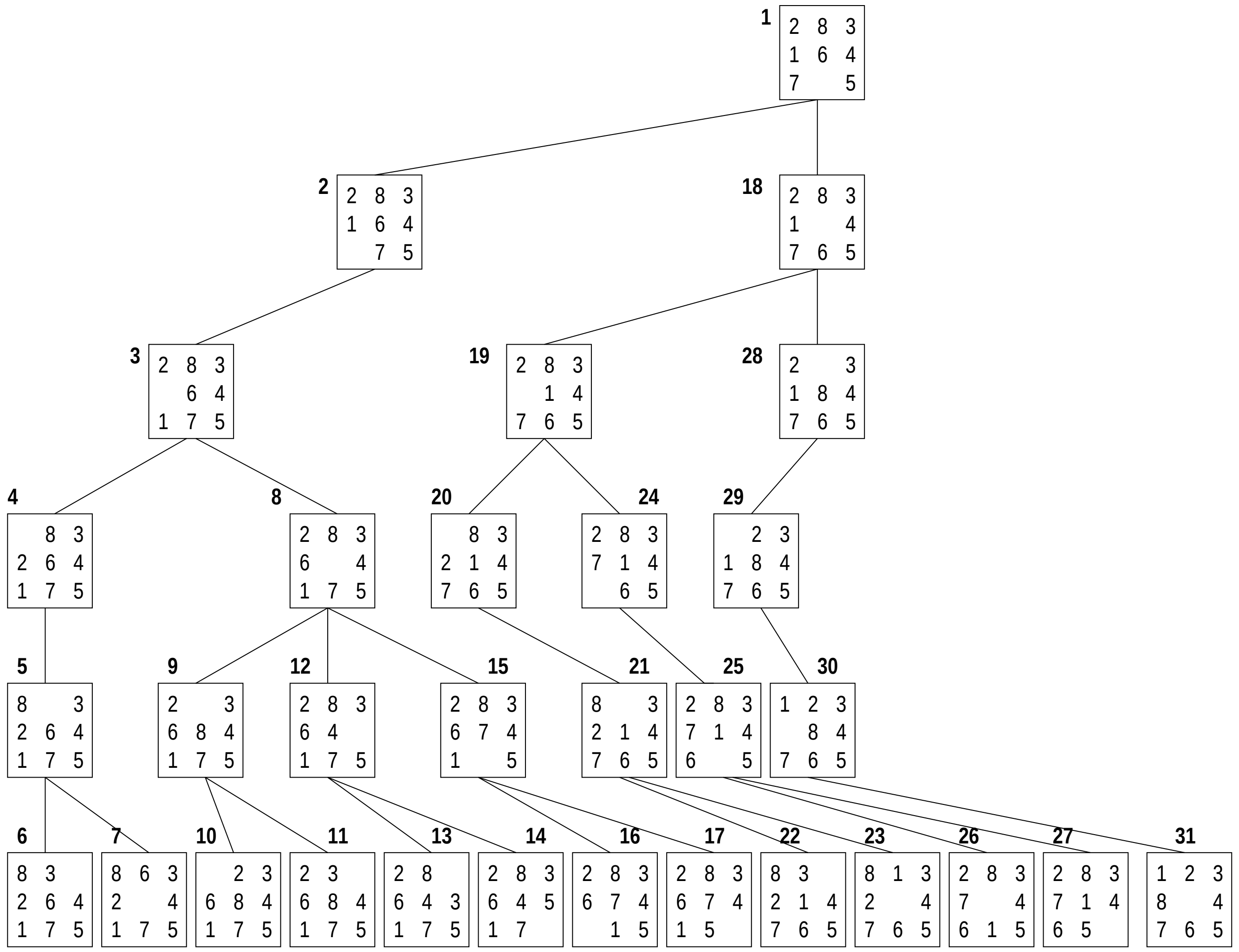
Depth-First Search

Idea: Always expand node at deepest level of tree and when search hits a dead-end return back to expand nodes at a shallower level

Can be implemented by using a stack to store frontier nodes

At any point depth-first search stores single path from root to leaf together with any remaining unexpanded siblings of nodes along path

Depth-First Search



Depth-First Search — Analysis

Storage: $O(bm)$ nodes (where m = maximum depth of search tree)

Time: $O(b^m)$

In cases where problem has many solutions depth-first search may outperform breadth-first search because there is a good chance it will happen upon a solution after exploring only a small part of the search space

However, depth-first search may get stuck following a deep or infinite path even when a solution exists at a relatively shallow level

Therefore, depth-first is not complete and not optimal

Avoid depth-first search for problems with deep or infinite paths

Depth-Limited Search

Idea: impose bound on depth of a path

In some problems you may know that a solution should be found within a certain cost (e.g., a certain number of moves) and therefore there is no need to search paths beyond this point for a solution

Depth-Limited Search — Analysis

Complete but not optimal (may not find shortest solution)

However, if the depth limit chosen is too small a solution may not be found and depth-limited search is incomplete in this case

Time and space complexity similar to depth-first search (but relative to depth limit rather than maximum depth)

Iterative Deepening Search

It can be very difficult to decide upon a depth limit for search

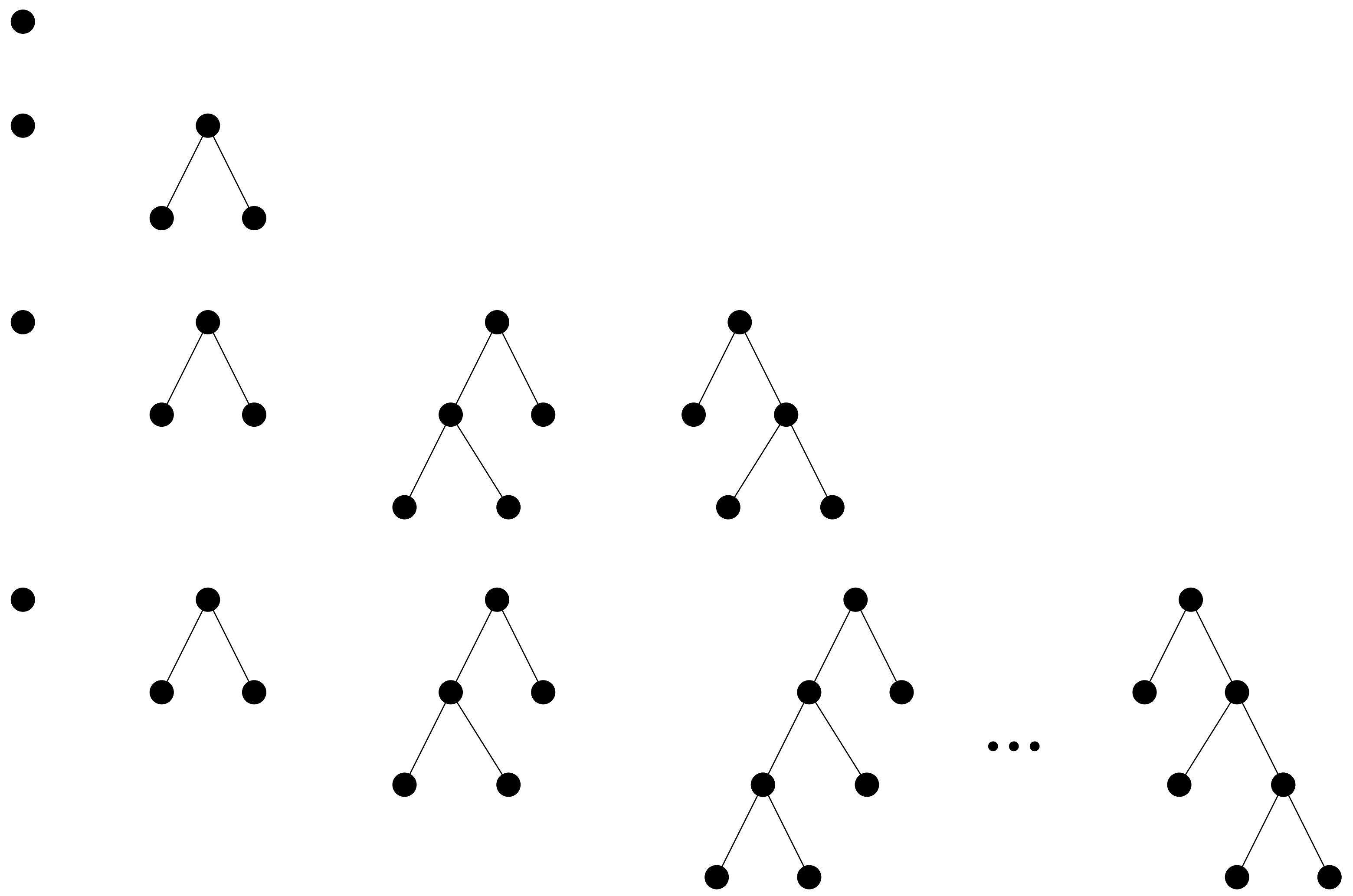
The maximum path cost between any two nodes is known as the *diameter* of the state space

This would be a good candidate for a depth limit but it may be difficult to determine in advance

Idea: try all possible depth limits in turn

Combines benfits of depth-first and breadth-first search

Iterative Deepening Search



Iterative Deepening Search — Analysis

Optimal; Complete; Space — $O(bd)$

Some states are expanded multiple times. Isn't this wasteful?

Number of expansions to depth d = $1 + b + b^2 + b^3 + \ldots + b^d$

Therefore, for iterative deepening, total expansions =
$(d+1)1 + (d)b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d$

The higher the branching factor, the lower the overhead (even for $b = 2$, search takes about twice as long)

Hence time complexity still $O(b^d)$

May consider doubling depth limit at each iteration — overhead $O(d\text{log}d)$

In general, iterative deepening is the preferred search strategy for a large search space where depth of solution is not known

Bidirectional Search

Idea: search forward from initial state and backward from goal state at the same time until the two meet

To search backwards we need to generate predecessors of nodes (this is not always possible or easy)

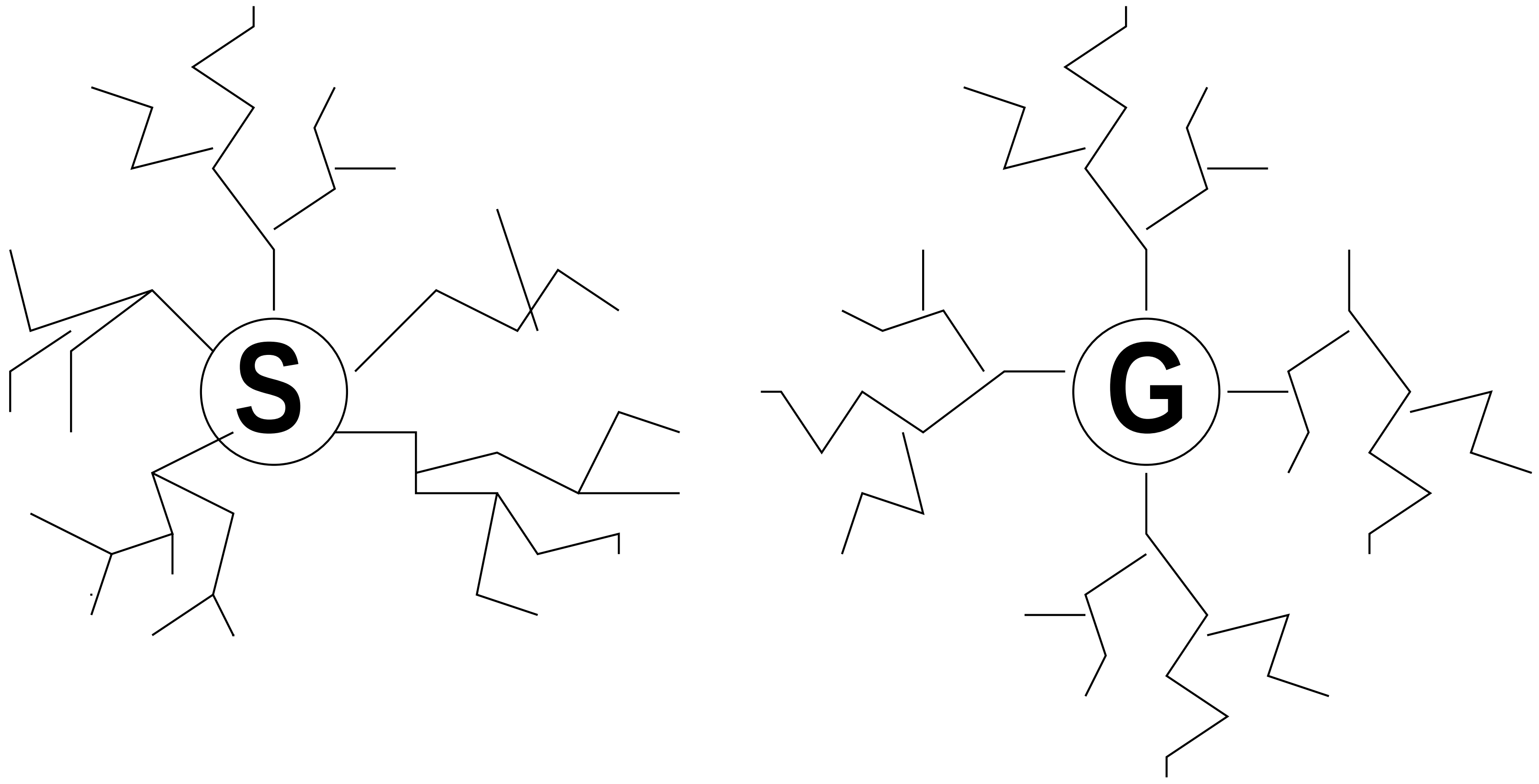
If operators are reversible successor sets and predecessor sets are identical

If there are many goal states we can try a multi-state search (how would you do this in chess, say?)

Need to check whether a node occurs in both searches — may not be easy

Which is the best search strategy for each half?

Bidirectional Search



Bidirectional Search — Analysis

If solution exists at depth d then bidirectional search requires time $O(2b^{\frac{d}{2}}) = O(b^{\frac{d}{2}})$ (assuming constant time checking of intersection)

To check for intersection must have all states from one of the searches in memory, therefore space complexity is $O(b^{\frac{d}{2}})$

Summary — Blind Search

Criterion	Breadth First	Uniform Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional
Time	b^d	b^d	b^m	b^l	b^d	$b^{\frac{d}{2}}$
Space	b^d	b^d	bm	bl	bd	$b^{\frac{d}{2}}$
Optimal	Yes	Yes	No	No	Yes	Yes
Complete	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

b — branching factor

d — depth of shallowest solution

m — maximum depth of tree

l — depth limit

Conclusion

We have surveyed a variety of uninformed search strategies

All can be implemented within the framework of the general search procedure

There are other considerations we can make like trying to save time by not expanding a node which has already been seen on some other path

There are a number of techniques available and often use is made of a hash table to store all nodes generated