Introduction to AI

Lecture 4

Uninformed Search Strategies

Dr. Tamal Ghosh
Department of CSE
Adamas University

Building Goal-Based Agents

- We have a **goal** to reach
  - Driving from point A to point B
  - Put 8 queens on a chess board such that no one attacks another
  - Prove that John is an ancestor of Mary
- We have information about the current **state**, where we are now at the beginning and after each action
- We have a set of **actions** we can take to move around (change from where we are) if the preconditions are met
- **Objective**: find a sequence of legal actions which will bring us from the start point to a goal

What is the goal to be achieved?

- Could describe a situation we want to achieve, a set of properties that we want to hold, etc.
- Requires defining a **“goal test”** so that we know what it means to have achieved/satisfied our goal.
- This is a hard part that is rarely tackled in AI, usually assuming that the system designer or user will specify the goal to be achieved.

What are the actions?

- Quantify all of the primitive actions or events that are sufficient to describe all necessary changes in solving a task/goal.
- No uncertainty associated with what an action does to the world. That is, given an action (aka operator or move) and a description of the current state of the world, the action completely specifies
  - **Precondition:** if that action CAN be applied to the current world (i.e., is it applicable and legal), and
  - **Effect:** what the exact state of the world will be after the action is performed in the current world (i.e., no need for "history" information to be able to compute what the new world looks like).

Actions

- Note also that actions can all be considered as **discrete events** that can be thought of as occurring at an **instant of time**.
  - That is, the world is in one situation, then an action occurs and the world is now in a new situation. For example, if "Mary is in class" and then performs the action "go home," then in the next situation she is "at home." There is no representation of a point in time where she is neither in class nor at home (i.e., in the state of "going home").
- The number of operators needed depends on the **representation** used in describing a state.
- Actions often are associated with **costs**

Representing states

- At any moment, the relevant world is represented as a **state**
  - **Initial (start) state**: S
  - An action (or an operation) changes the current state to another state (if it is applied): state transition
  - An action can be taken (**applicable**) only if the its precondition is met by the current state
  - For a given state, there might be **more than one** applicable actions
  - **Goal state**: a state satisfies the goal description or passes the goal test
  - Dead-end state: a non-goal state to which no action is applicable

Representing states

- **Stat space**:
  - Includes the initial state S and all other states that are reachable from S by a sequence of actions
  - A state space can be organized as a graph:

    nodes: states in the space

    arcs: actions/operations
- The **size of a problem** is usually described in terms of the **number of states** (or the size of the state space) that are possible.
  - Tic-Tac-Toe has about 3^9 states.
  - Checkers has about 10^40 states.
  - Rubik's Cube has about 10^19 states.
  - Chess has about 10^120 states in a typical game.
  - GO has more states than Chess

Closed World Assumption

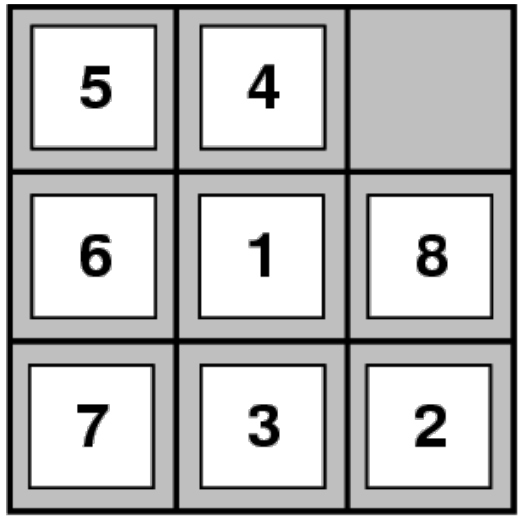
- We will generally use the **Closed World Assumption**. It is a formal logic that assumes that a statement that is true is also known to be true. This means that what is not known to be true is false.
- All necessary information about a problem domain is available in each percept so that each state is a complete description of the world.
- There is no incomplete information at any point in time.

Some example problems

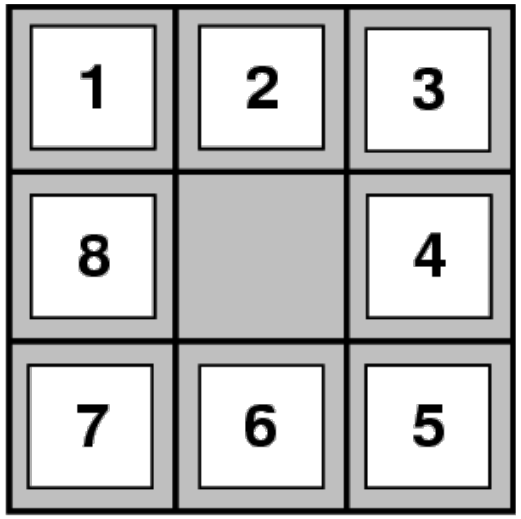
- Toy problems and micro-worlds
  - 8-Puzzle
  - Missionaries and Cannibals
  - Cryptarithmetic
  - Remove 5 Sticks
  - Traveling Salesman Problem (TSP)
- Real-world-problems

8-Puzzle

Given an initial configuration of 8 numbered tiles on a 3 x 3 board, move the tiles in such a way so as to produce a desired goal configuration of the tiles.



Start State

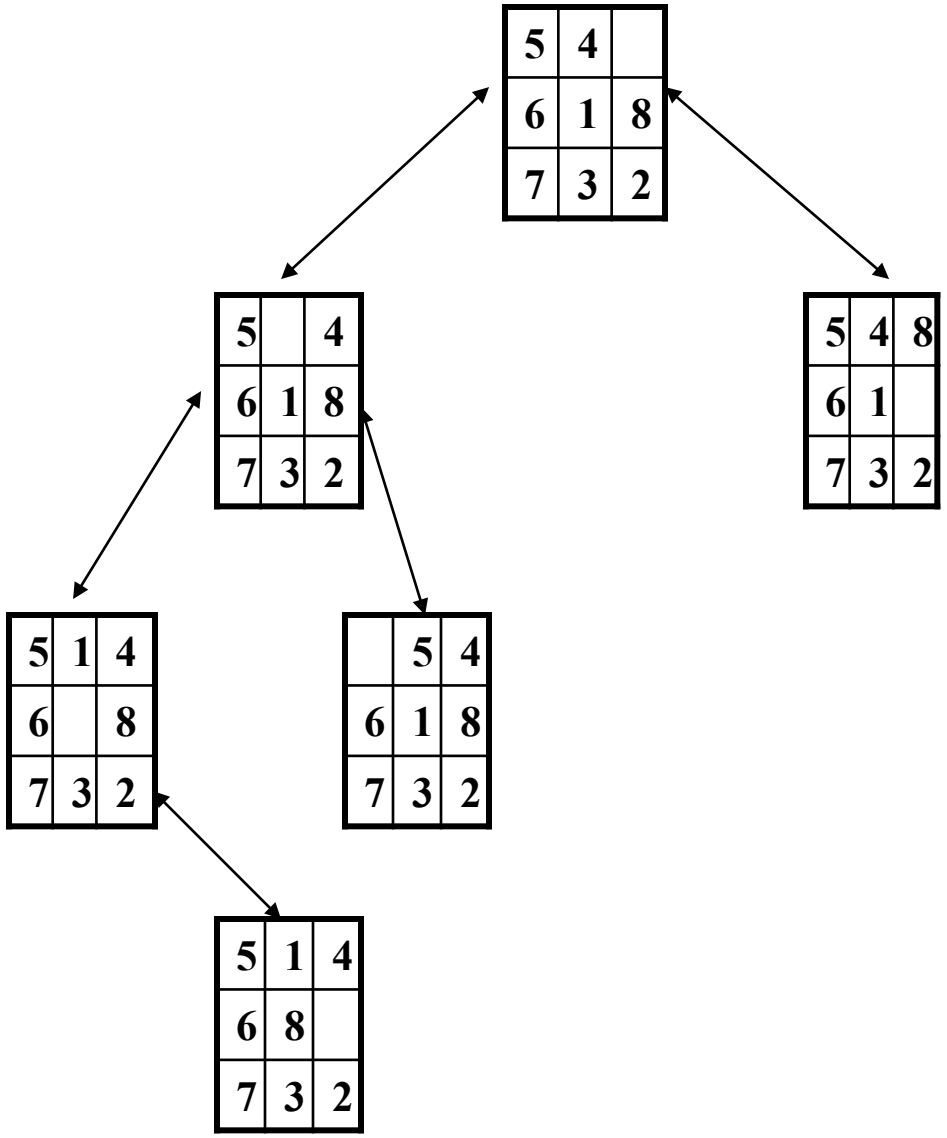


Goal State

8 puzzle

- **State:** 3 x 3 array configuration of the tiles on the board.
- **Operators:** Move Blank square Left, Right, Up or Down.
  - This is a more efficient encoding of the operators than one in which each of four possible moves for each of the 8 distinct tiles is used.
- **Initial State:** A particular configuration of the board.
- **Goal:** A particular configuration of the board.
- The state space is partitioned into two subspaces
- NP-complete problem, need to examine O(2^k) states where k is the length of the solution path.
- 15-puzzle problems (4 x 4 grid with 15 numbered tiles), and N-puzzles (N = n^2-1)

A portion of the state space of a 8-Puzzle problem



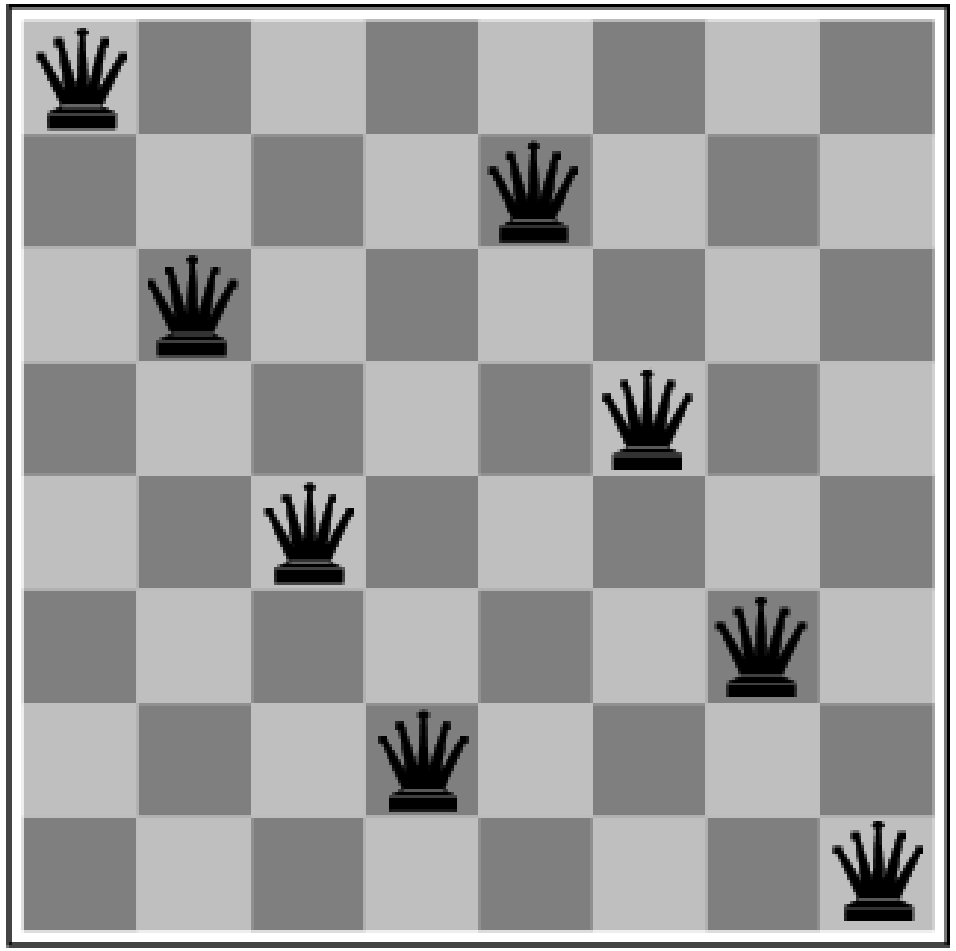
The 8-Queens Problem

Place eight queens on a chessboard such that no queen attacks any other!

Total # of states: 4.4x10^9

Total # of solutions:

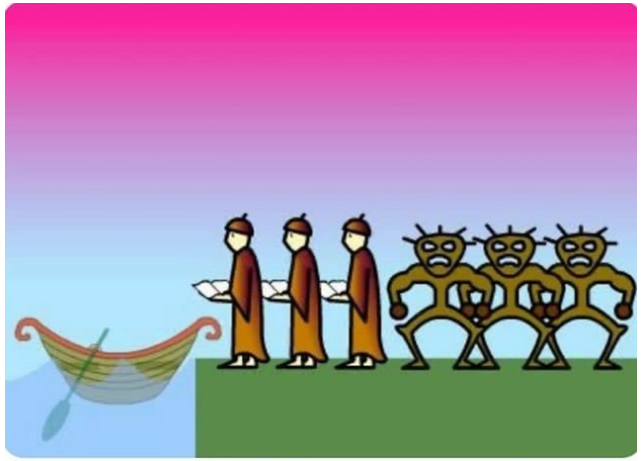
12 (or 96)



Missionaries and Cannibals

There are 3 missionaries, 3 cannibals, and 1 boat that can carry up to two people on one side of a river.

- **Goal**: Move all the missionaries and cannibals across the river.
- **Constraint:** Missionaries can never be outnumbered by cannibals on either side of river, or else the missionaries are killed.
- **State:** configuration of missionaries and cannibals and boat on each side of river.
- **Operators:** Move boat containing 1 or 2 occupants across the river (in either direction) to the other side.



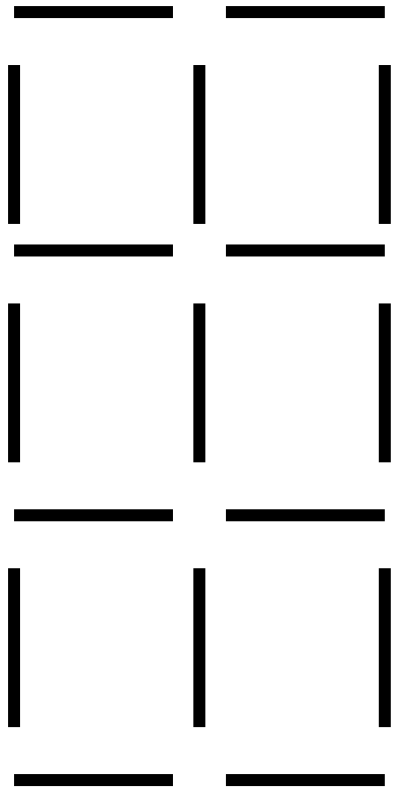
3 Missionaries and 3 Cannibals wish to cross the river. They have a boat that will carry two people. Everyone can navigatethe boat. If at any time the Cannibals outnumber the missionaries on either bank of the river, they will eat the Missionaries. Find the smallest number of crossings that will allow everyone to cross the river safely.
The problem can be solved in 11 moves. But people rarely get the optimal solution, because the MC problem contains a 'tricky' state at the end, where two people move back across the river.

Missionaries and Cannibals Solution

	Near side		***Far side***	
0 Initial setup:	MMMCCC	B		-
1 Two cannibals cross over:	MMMC		B	CC
2 One comes back:	MMMCC	B		C
3 Two cannibals go over again:	MMM		B	CCC
4 One comes back:	MMMC	B		CC
5 Two missionaries cross:	MC		B	MMCC
6 A missionary & cannibal return:	MMCC	B		MC
7 Two missionaries cross again:	CC		B	MMMC
8 A cannibal returns:	CCC	B		MMM
9 Two cannibals cross:	C		B	MMMCC
10 One returns:	CC	B		MMMC
11 And brings over the third:	-		B	MMMCCC

Remove 5 Sticks

- Given the following configuration of sticks, remove exactly 5 sticks in such a way that the remaining configuration forms exactly 3 squares.



Traveling Salesman Problem

- Given a road map of n cities, find the **shortest** tour which visits every city on the map exactly once and then return to the original city (*Hamiltonian circuit*)
- (Geometric version):
  - a complete graph of n vertices.
  - n!/2n legal tours
  - Find one legal tour that is shortest

Formalizing Search in a State Space

- A state space is a **graph**, (V, E) where V is a set of **nodes** and E is a set of **arcs**, where each arc is directed from a node to another node
- **node:** corresponds to a **state**
  - state description
  - plus optionally other information related to the parent of the node, operation to generate the node from that parent, and other bookkeeping data)
- **arc:** corresponds to an applicable action/operation.
  - the source and destination nodes are called as **parent (immediate predecessor)** and **child** (**immediate successor**) nodes with respect to each other
  - ancestors((predecessors) and descendents (successors)
  - each arc has a *fixed, non-negative* **cost** associated with it, corresponding to the cost of the action

- **node generation:** making explicit a node by applying an action to another node which has been made explicit
- **node expansion:** generating **all** children of an explicit node by applying **all** applicable operations to that node
- One or more nodes are designated as **start nodes**
- A **goal test** predicate is applied to a node to determine if its associated state is a goal state
- A **solution** is a sequence of operations that is associated with a path in a state space from a start node to a goal node
- The **cost of a solution** is the sum of the arc costs on the solution path

- **State-space search** is the process of searching through a state space for a solution by making explicit a sufficient portion of an implicit state-space graph to include a goal node.
  - Hence, initially V={S}, where S is the start node; when S is expanded, its successors are generated and those nodes are added to V and the associated arcs are added to E. This process continues until a goal node is generated (included in V) and identified (by goal test)
- During search, a node can be in one of the three categories:
  - Not generated yet (has not been made explicit yet)
  - **OPEN**: generated but not expanded
  - **CLOSED**: expanded
  - Search strategies differ mainly on **how to select an OPEN node** for expansion at each step of search

A General State-Space Search Algorithm

- Node n
  - state description
  - parent (may use a backpointer) (if needed)
  - Operator used to generate n (optional)
  - Depth of n (optional)
  - Path cost from S to n (if available)
- **OPEN** list
  - initialization: {S}
  - node insertion/removal depends on specific search strategy
- **CLOSED** list
  - initialization: {}
  - organized by backpointers

A General State-Space Search Algorithm

```
open := {S}; closed :={};
repeat
   n := select(open);              /* select one node from open for expansion */
   if n is a goal
        then exit with success;  /* delayed goal testing */
   expand(n)
            /* generate all children of n
               put these newly generated nodes in open (check duplicates)
               put n in closed (check duplicates) */
until open = {};
exit with failure
```

Evaluating Search Strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time Complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**
- **Space Complexity**
  - How much space is used by the algorithm? Usually measured in terms of the **maximum size that the “OPEN" list** becomes during the search
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

Uninformed vs. Informed Search

- **Uninformed Search Strategies**
  - Breadth-First search
  - Depth-First search
  - Uniform-Cost search
  - **Depth-First Iterative Deepening search**
- **Informed Search Strategies**
  - Hill climbing
  - Best-first search
  - Greedy Search
  - Beam search
  - Algorithm A
  - **Algorithm A***

Breadth-First

- Algorithm outline:
  - Always select from the OPEN, the node with the **smallest depth** for expansion, and put all newly generated nodes into OPEN
  - OPEN is organized as **FIFO** (first-in, first-out) list, i.e., a **queue**.
  - Terminate if a node selected for expansion is a goal
- **Properties**
  - **Complete**
  - **Optimal** (i.e., admissible) if all operators have the same cost. Otherwise, not optimal but finds solution with **shortest path length** (shallowest solution).
  - **Exponential time and space complexity**,

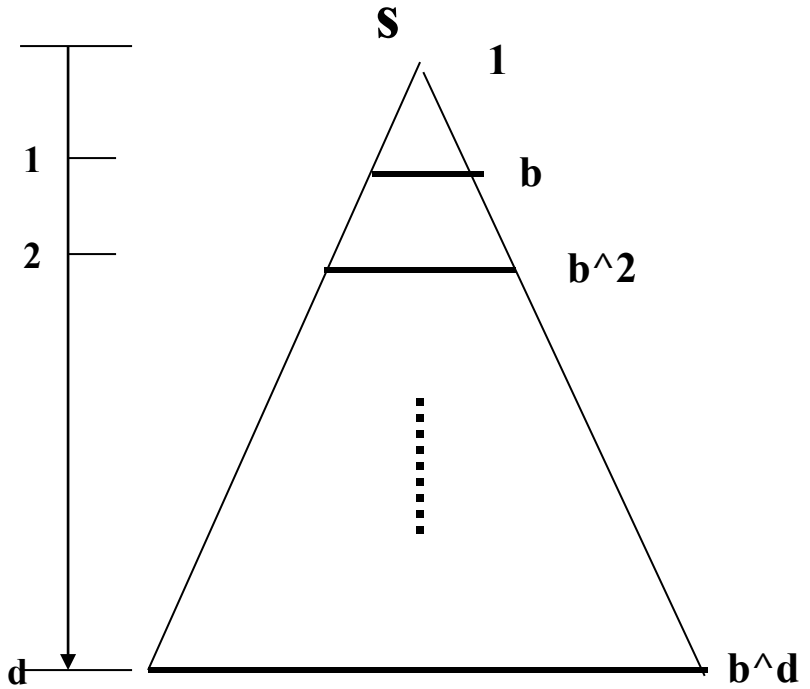
    O(b^d) nodes will be generated, where

    d is the depth of the solution and

    b is the branching factor (i.e., number of children) at each node

Breadth-First

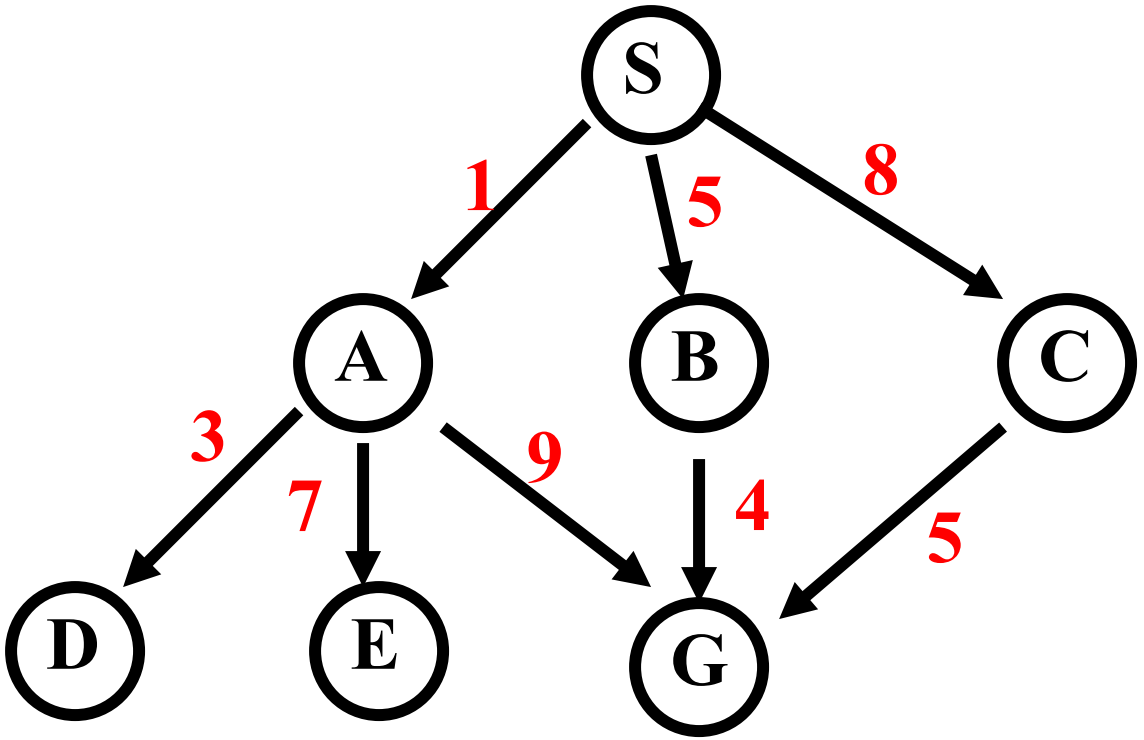
- A complete search tree of depth d where each non-leaf node has b children, has a total of 1 + b + b^2 + ... + b^d = (b^(d+1) - 1)/(b-1) nodes
- Time complexity (# of nodes generated): O(b^d)
- Space complexity (maximum length of OPEN): O(b^d)



  - For a complete search tree of depth 12, where every node at depths 0, ..., 11 has 10 children and every node at depth 12 has 0 children, there are 1 + 10 + 100 + 1000 + ... + 10^12 = (10^13 - 1)/9 = O(10^12) nodes in the complete search tree.

- BFS is suitable for problems with shallow solutions

Example Illustrating Uninformed Search Strategies



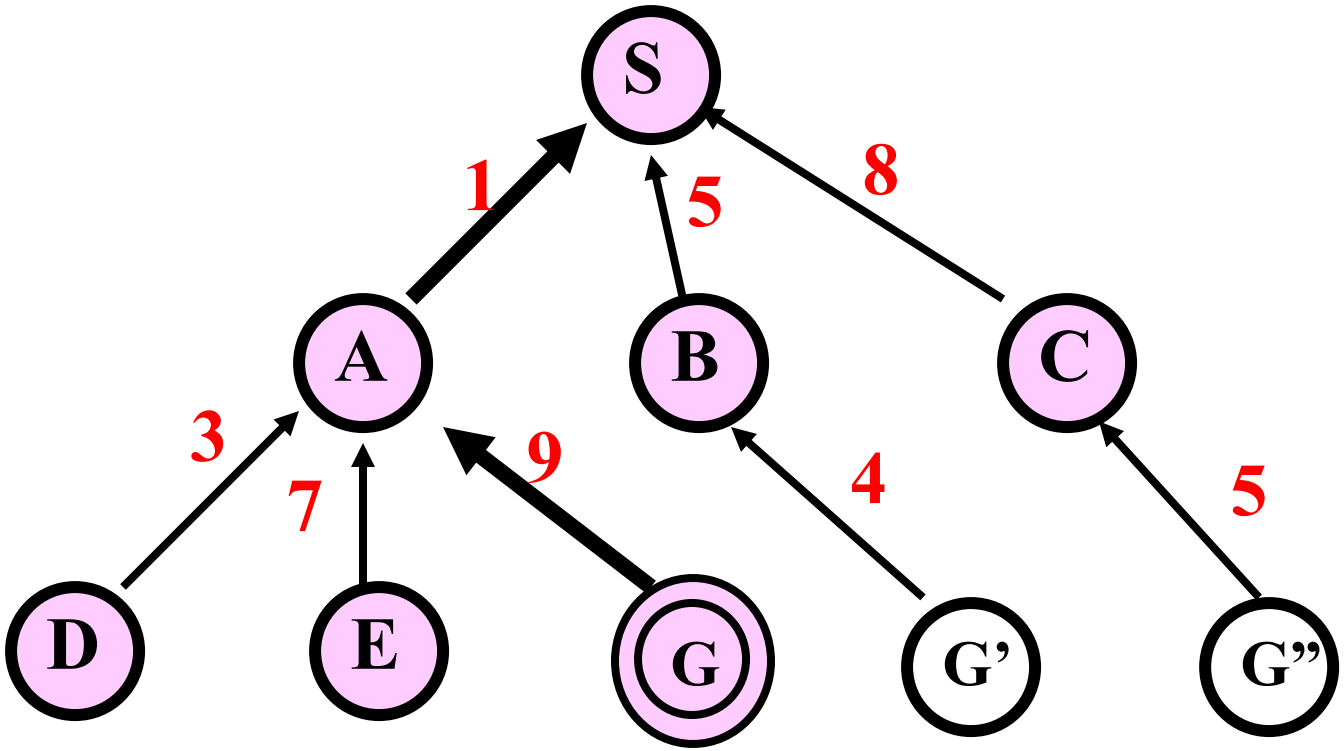
Breadth-First Search

exp. node	OPEN list	CLOSED list
	{ S }	{}
S	{ A B C }	{S}
A	{ B C D E G }	{S A}
B	{ C D E G G' }	{S A B}
C	{ D E G G' G" }	{S A B C}
D	{ E G G' G" }	{S A B C D}
E	{ G G' G" }	{S A B C D E}
G	{ G' G" }	{S A B C D E}

Solution path found is S A G <-- this G also has cost 10

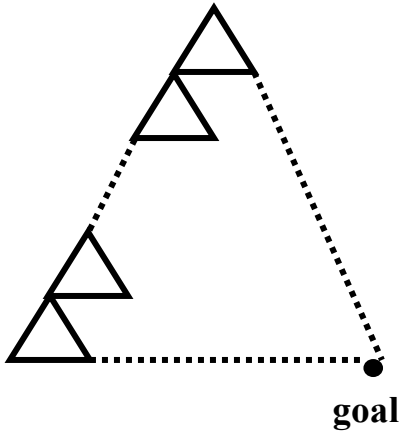
Number of nodes expanded (including goal node) = 7

CLOSED List: the search tree connected by backpointers



Depth-First (DFS)

- Algorithm outline:
  - Always select from the OPEN, the node with the **greatest depth** for expansion, and put all newly generated nodes into OPEN
  - OPEN is organized as **LIFO** (last-in, first-out) list.
  - Terminate if a node selected for expansion is a goal

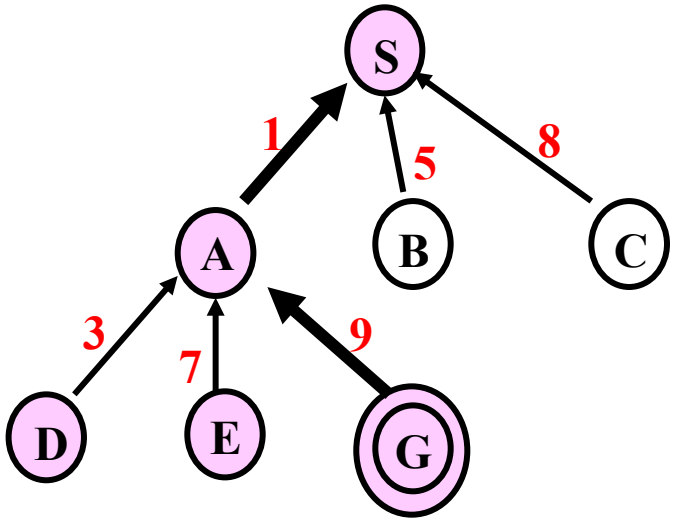


- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D (How to determine the depth bound?)
- **Not complete** (with or without cycle detection, and with or without a cutoff depth)
- **Exponential time**, O(b^d), but only **linear space**, O(bd), required
- Can find deep solutions quickly if lucky
- When search hits a deadend, can only back up one level at a time even if the "problem" occurs because of a bad operator choice near the top of the tree. Hence, only does "chronological backtracking"

Depth-First Search

return GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)

exp. node	**OPEN list**	**CLOSED list**
	{ S }	
S	{ A B C }	
A	{ D E G B C}	
D	{ E G B C }	
E	{ G B C }	
G	{ B C }	



Solution path found is S A G <-- this G has cost 10

Number of nodes expanded (including goal node) = 5

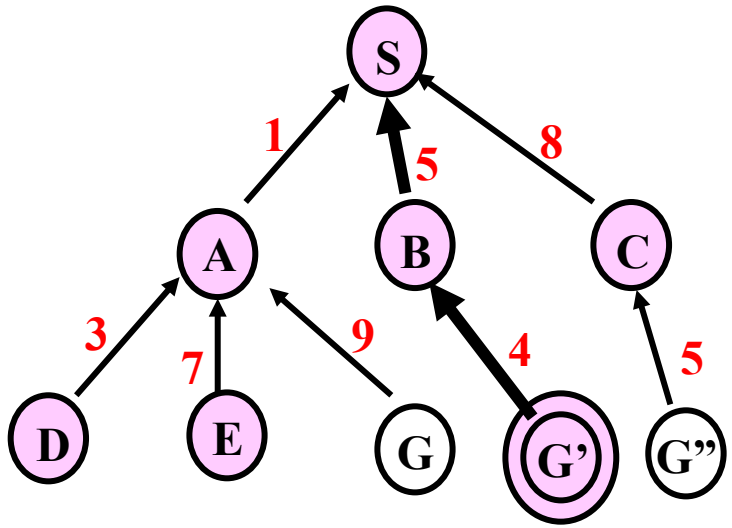
Uniform-Cost (UCS)

- Let g(n) = cost of the path from the start node to an open node n
- Algorithm outline:
  - Always select from the OPEN the node with the **least g(.) value** for expansion, and put all newly generated nodes into OPEN
  - Nodes in OPEN are sorted by their g(.) values (in ascending order)
  - Terminate if a node selected for expansion is a goal
- Called *"Dijkstra's Algorithm"* in the algorithms literature and similar to *"Branch and Bound Algorithm"* in operations research literature

Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

exp. node	nodes list	CLOSED list
	{S(0)}	
S	{A(1) B(5) C(8)}	
A	{D(4) B(5) C(8) E(8) G(10)}	
D	{B(5) C(8) E(8) G(10)}	
B	{C(8) E(8) G'(9) G(10)}	
C	{E(8) G'(9) G(10) G"(13)}	
E	{G'(9) G(10) G"(13) }	
G'	{G(10) G"(13) }	



Solution path found is S B G <-- this G has cost 9, not 10

Number of nodes expanded (including goal node) = 7

Uniform-Cost (UCS)

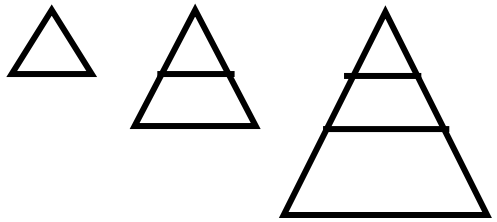
- **Complete** (if cost of each action is not infinitesimal)
  - The total # of nodes n with g(n) <= g(goal) in the state space is finite
  - If n' is a child of n, then g(n') = g(n) + c(n, n') > g(n)
  - Goal node will eventually be generated (put in OPEN) and selected for expansion (and passes the goal test)
- **Optimal/Admissible**
  - Admissibility depends on the goal test being applied when a node is removed from the OPEN list, not when it's parent node is expanded and the node is first generated (delayed goal testing)
  - Multiple solution paths (following different backpointers)
  - Each solution path that can be generated from an open node n will have its path cost >= g(n)
  - When the first goal node is selected for expansion (and passes the goal test), its path cost is less than or equal to g(n) of every OPEN node n (and solutions entailed by n)
- **Exponential time and space complexity**,
  - worst case: becomes BFS when all arcs cost the same

Depth-First Iterative Deepening (DFID)

- BF and DF both have exponential time complexity O(b^d)
  - BF is **complete** but has exponential space complexity (**conservative**)
  - DF has **linear space complexity** but is incomplete (**radical**)
- Space is often a **harder** resource constraint than time
- Can we have an algorithm that
  - Is complete
  - Has linear space complexity, and
  - Has time complexity of O(b^d)
- DFID by Korf in 1985 (17 years after A*)

  First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

  until solution found do
  ***DFS** with depth bound c*
  c = c+1



Depth-First Iterative Deepening (DFID)

- **Complete** (iteratively generate all nodes up to depth d)
- **Optimal/Admissible** if all operators have the same cost. Otherwise, not optimal but does guarantee finding solution of shortest length (like BF).
- **Linear space complexity:** O(bd), (like DF)
- **Time complexity** is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, O(b^d)

Depth-First Iterative Deepening

- If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc., and node at depth 1 is generated d times.

  Hence

  total(d) = b^d + 2b^(d-1) + ... + db

  <= b^d / (1 - 1/b)^2 = O(b^d).

  – If b=4, then worst case is 1.78 * 4^d, i.e., 78% more nodes searched than exist at depth d (in the worst case).

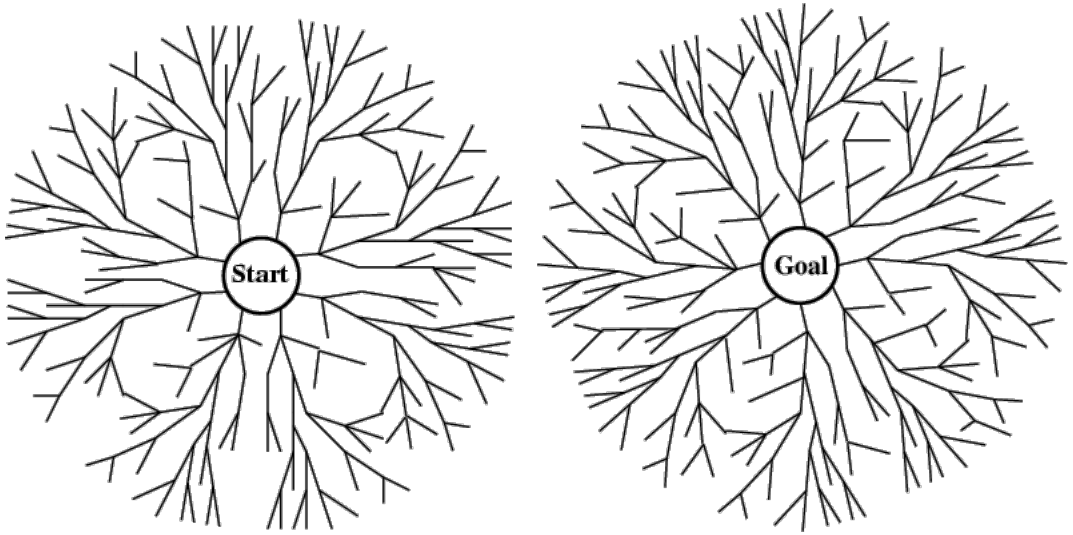
$$tota(d) = 1 \cdot b^d + 2 \cdot b^{d-1} + \text{L} + (d-1) \cdot b^2 + d \cdot b$$
$$= b^d (1 + 2 \cdot b^{-1} + \text{L} + (d-1) \cdot b^{2-d} + d \cdot b^{1-d})$$

Let $x = b^{-1}$, then

$$tota(d) = b^d (1 + 2 \cdot x^1 + \text{L} + (d-1) \cdot x^{d-2} + d \cdot x^{d-1})$$
$$= b^d \frac{d}{dx}(x + x^2 + \text{L} + x^{d-1} + x^d)$$
$$= b^d \frac{d}{dx} \frac{(x - x^{d+1})}{1-x}$$
$$\le b^d \frac{d}{dx} \frac{x}{1-x} \qquad /* x^{d+1} << 1 \text{ when } d \text{ is large since } 1/b < 1 */$$
$$= b^d \frac{1 \cdot (1-x) - x \cdot (-1)}{(1-x)^2}. \text{ Therefore}$$

$$tota(d) \le b^d /(1-x)^2 = b^d /(1-b^{-1})^2$$

Bi-directional search



- Alternate searching from the start state toward the goal and from the goal state toward the start.
- Stop when the frontiers intersect.
- Works well only when there are unique start and goal states and when actions are reversible
- Can lead to finding a solution more quickly (but watch out for pathological situations).

Comparing Search Strategies

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

When to use what

- **Depth-First Search:**
  - Many solutions exist
  - Know (or have a good estimate of) the depth of solution
- **Breadth-First Search**:
  - Some solutions are known to be shallow
- **Uniform-Cost Search**:
  - Actions have varying costs
  - Least cost solution is required

  This is the only uninformed search that worries about costs.
- **Iterative-Deepening Search**:
  - Space is limited and the shortest solution path is required