CMSC 671
Fall 2010

Tue 9/7/10

Lisp
Problem solving as search
Uninformed Search

Prof. Laura Zavala, laura.zavala@umbc.edu, ITE 373, 410-455-8775

LISP

Lisp

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

Graham article

Why Lisp?

- Because it’s the most widely used AI programming language
- Because it’s good for writing production software (Graham article)
- Because it’s got lots of features other languages don’t
- Because you can write new programs and extend old programs really, really quickly in Lisp

Why All Those Parentheses?

- Lisp syntax: parenthesized prefix notation
- Surprisingly readable if you indent properly
- Makes prefix notation manageable
- An expression is an expression is an expression, whether it's inside another one or not
  - `(+ 1 2)`
  - `(* (+ 1 2) 3)`
  - `(list (* 3 5) 'atom '(list inside a list) (list 3 4) '(((very) (very) (very) (nested list))))`

Cool Things About Lisp

- Functions as objects (pass a function as an argument)
- Lambda expressions (construct a function on the fly)
- Lists as first-class objects
- Program as data
- Macros (smart expansion of expressions)
- Symbol manipulation

Functional Programming

- Decomposes a problem into a set of functions
- Functions only take inputs and produce outputs, and don't have any internal state (global and local variables) that affects the output produced for a given input
- Recursion is a natural way of thinking in functional programming languages
- Truly functional programs are highly parallelizable

Basic Lisp Types

- NIL and T
- Symbols
  - `‘a  ‘x  ‘marie`
- Numbers
  - `27  -2  7.519`
- Lists
  - `‘(a b c)  ‘(2 3 marie)`
- Strings
  - `“This is a string!”`
- Characters
  - `#\x  #\-  #\B`

Basic Lisp Functions

- Numeric functions: `+ - * / incf decf`
- List access: `car` (`first`), `second` … `tenth`, `nth`, `cdr` (`rest`), `last`, `length`
- List construction: `cons, append, list`
- Advanced list processing: `assoc, mapcar, mapcan`
- Predicates: `listp, numberp, stringp, atom, null, equal, eql, and, or, not`
- Special forms: `setq/setf, quote, defun, if, cond, case, progn, loop`

MapReduce

- Without understanding functional programming, you can't invent **MapReduce**, the algorithm that makes Google so massively scalable.

- The terms Map and Reduce come from Lisp and functional programming.

Useful help facilities

- `(apropos 'str)` → list of symbols whose name contains 'str
- `(describe 'symbol)` → description of symbol
- `(describe #'fn)` → description of function
- `(trace fn)` → print a trace of fn as it runs
- `(print "string")` → print output
- `(format …)` → formatted output (see Norvig p. 84)
- `:a` → abort one level out of debugger

Great! How Can I Get Started?

- On sunserver or linux server (CS), run /usr/local/bin/clisp
- From http://clisp.cons.org you can download CLISP for your own PC (Windows or Linux)
- Great Lisp resource page: http://www.apl.jhu.edu/~hall/lisp.html

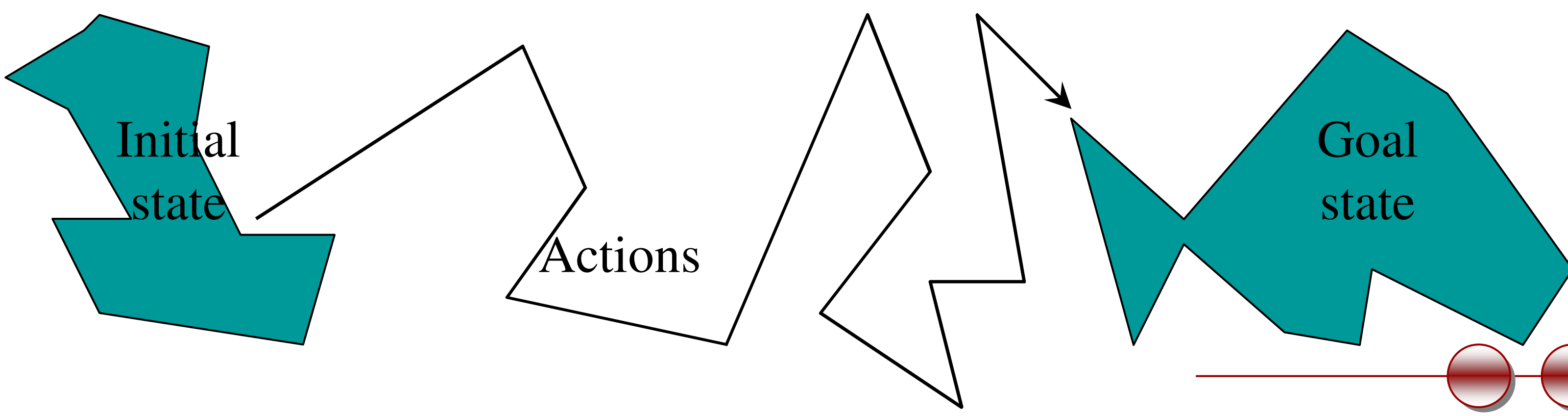
Remember to subscribe!

- Course mailing list: cps-cmsc671@lists.umbc.edu
  - Visit http://lists.umbc.edu
  - Search for cps-cmsc671
  - Click “Subscribe” link

Building goal-based agents

To build a goal-based agent we need to answer the following questions:

- What is the goal to be achieved?
- What are the actions?
- What *relevant* information is necessary to encode in order to describe the state of the world, describe the available transitions, and solve the problem?



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 question that is rarely tackled in AI, usually assuming that the system designer or user will specify the goal to be achieved.
- Certainly psychologists and motivational speakers always stress the importance of people establishing clear goals for themselves as the first step towards solving a problem.
- What are your goals???

What is the goal to be achieved?

2	8	3
1	6	4
7		5

Start State

1	2	3
8		4
7	6	5

Goal State a

	1	2
3	4	5
6	7	8

Goal State b

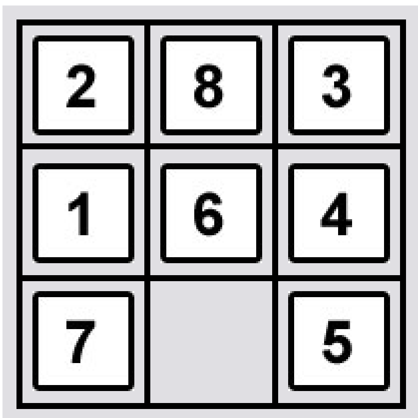
What are the actions?

- Characterize the **primitive actions** or events that are available for making changes in the world in order to achieve a goal.
- **Deterministic** world: no uncertainty in an action's effects. Given an action (a.k.a. **operator** or move) and a description of the **current world state**, the action completely specifies
  - whether that action *can* be applied to the current world (i.e., is it applicable and legal), and
  - 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 compute what the new world looks like).

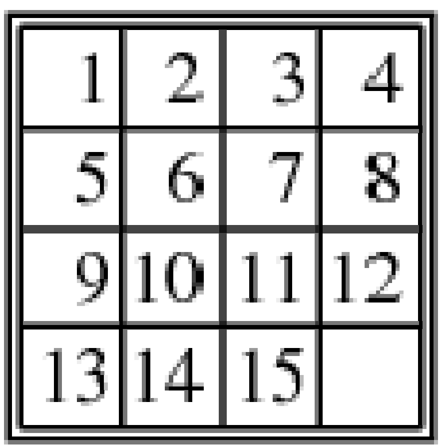
Representing actions

- Note also that actions in this framework can all be considered as **discrete events** that occur at an **instant of time**.
  - 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”).

Representing actions



- The number of actions / operators depends on the **representation** used in describing a state
  - In the 8-puzzle, we could specify 4 possible moves for each of the 8 tiles, resulting in a total of **4*8=32 operators**
  - On the other hand, we could specify four moves for the "blank" square and we would only need **4 operators**
    - BlankUp, BlankDown, BlankLeft, BlankRight

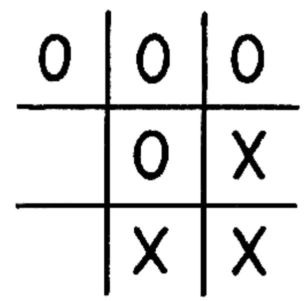


Only **4 operators** also for 15-puzzle

Representational shift can greatly simplify a problem!

Representing states

- What information is necessary to encode about the world to sufficiently describe all relevant aspects to solving the goal? That is, what knowledge needs to be represented in a state description to adequately describe the current state or situation of the world?
- The **size of a problem** is usually described in terms of the **number of states** 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.



Closed World Assumption

- We will generally use the **Closed World Assumption**.
- 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.

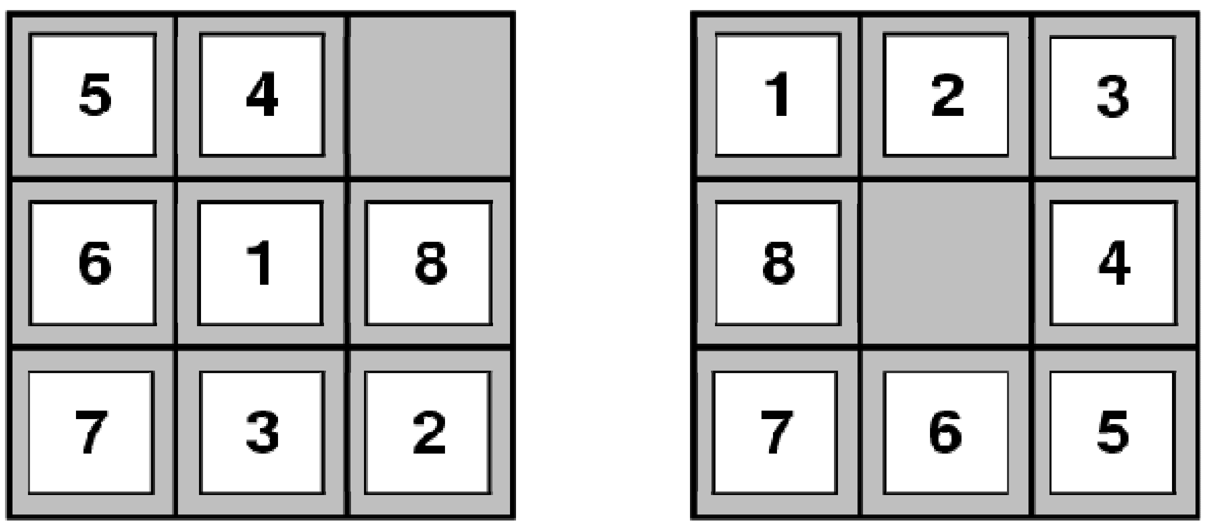
Some example problems

- Toy problems and micro-worlds
  - 8-Puzzle
  - Missionaries and Cannibals
  - Cryptarithmetic
  - Remove 5 Sticks
  - Water Jug Problem
- Real-world problems

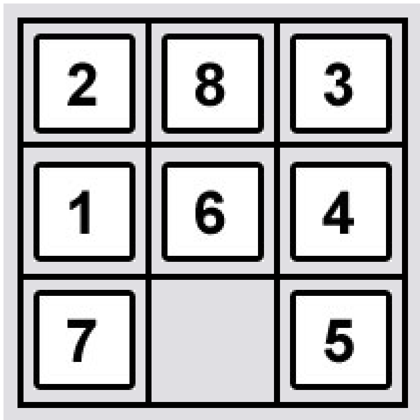
$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

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

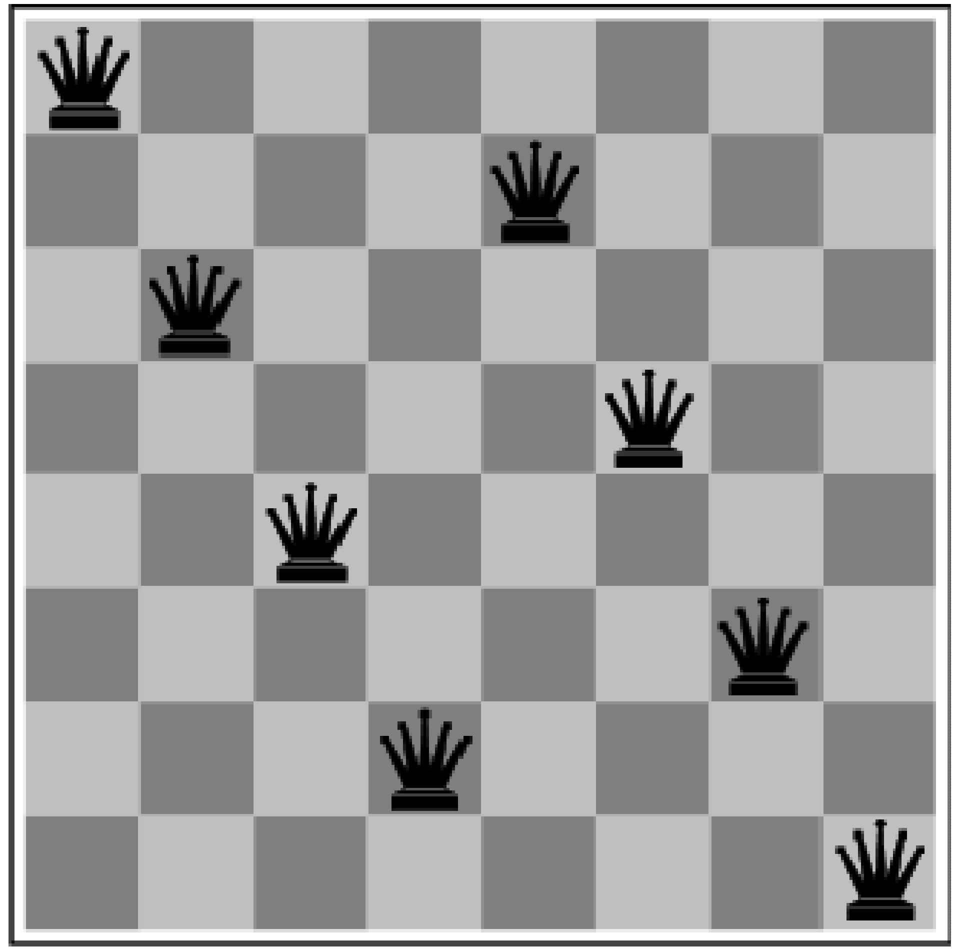


8 puzzle

- **State:** 3 x 3 array configuration of the tiles on the board.
- **Operators:** Blank Left, Blank Right, Blank Up, Blank Down.
- **Initial State:** A particular configuration of the board.
- **Goal:** A particular configuration of the board.

The 8-Queens Problem

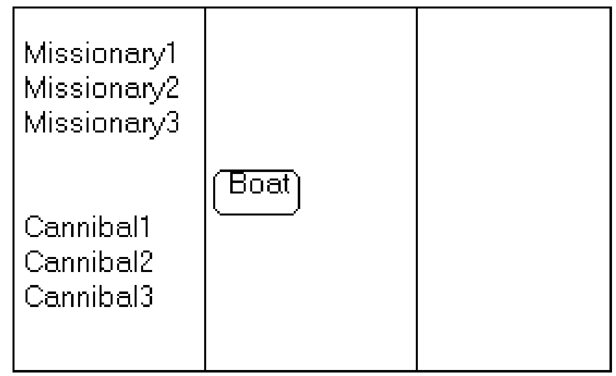
Place eight queens on a chessboard such that no queen attacks any other!



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 some set of occupants across the river (in either direction) to the other side.



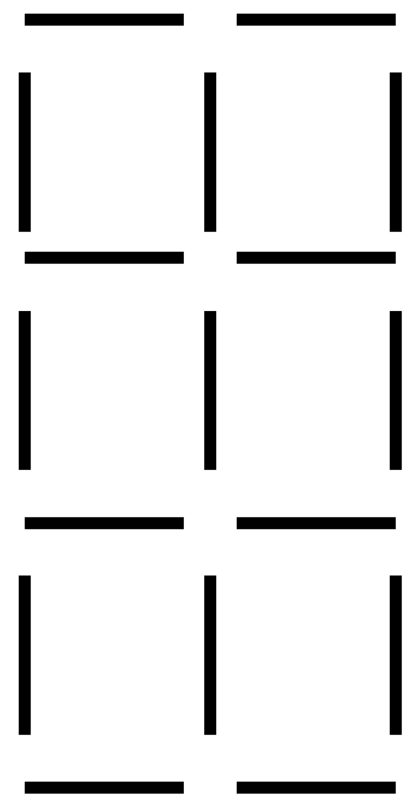
3 Missionaries and 3 Cannibals wish to cross the river. They have a boat that will carry two people. Everyone can navigatethe boat. If at any time the Cannibals outnumber the missionaries on either bank of the river, they will eat the Missionaries. Find the smallest number of crossings that will allow 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.



Water Jug Problem

Given a full 5-gallon jug and an empty 2-gallon jug, the goal is to fill the 2-gallon jug with exactly one gallon of water.

- State = (x,y), where x is the number of gallons of water in the 5-gallon jug and y is # of gallons in the 2-gallon jug
- Initial State = (5,0)
- Goal State = (*,1), where * means any amount

Operator table

Name	Cond.	Transition	Effect
Empty5	–	$(x,y) \rightarrow (0,y)$	Empty 5-gal. jug
Empty2	–	$(x,y) \rightarrow (x,0)$	Empty 2-gal. jug
2to5	$x \leq 3$	$(x,2) \rightarrow (x+2,0)$	Pour 2-gal. into 5-gal.
5to2	$x \geq 2$	$(x,0) \rightarrow (x-2,2)$	Pour 5-gal. into 2-gal.
5to2part	$y < 2$	$(1,y) \rightarrow (0,y+1)$	Pour partial 5-gal. into 2-gal.

Some real-world problems

- Route finding
- Touring (traveling salesman)
- Logistics
- VLSI layout
- Robot navigation

Knowledge representation issues

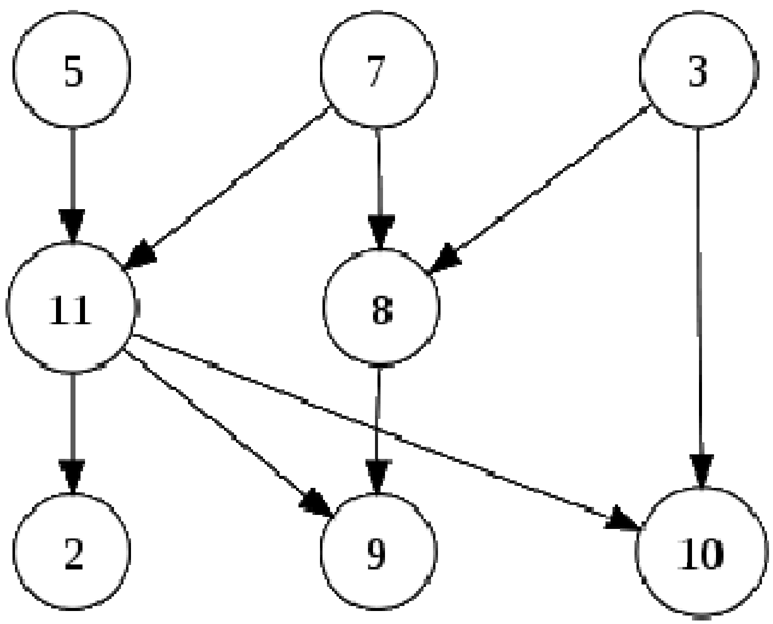
- What's in a state ?
  - Is the color of the boat relevant to solving the Missionaries and Cannibals problem? Is sunspot activity relevant to predicting the stock market? What to represent is a very hard problem that is usually left to the system designer to specify.
- What **level of abstraction** or detail to describe the world.
  - Too fine-grained and we'll "miss the forest for the trees." Too coarse-grained and we'll miss critical details for solving the problem.

Knowledge representation issues

- The number of states depends on the representation and level of abstraction chosen.
  - In the Remove-5-Sticks problem, if we represent the individual sticks, then there are 17-choose-5 possible ways of removing 5 sticks. On the other hand, if we represent the "squares" defined by 4 sticks, then there are 6 squares initially and we must remove 3 squares, so only 6-choose-3 ways of removing 3 squares.

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**, and each arc is directed from a node to another node



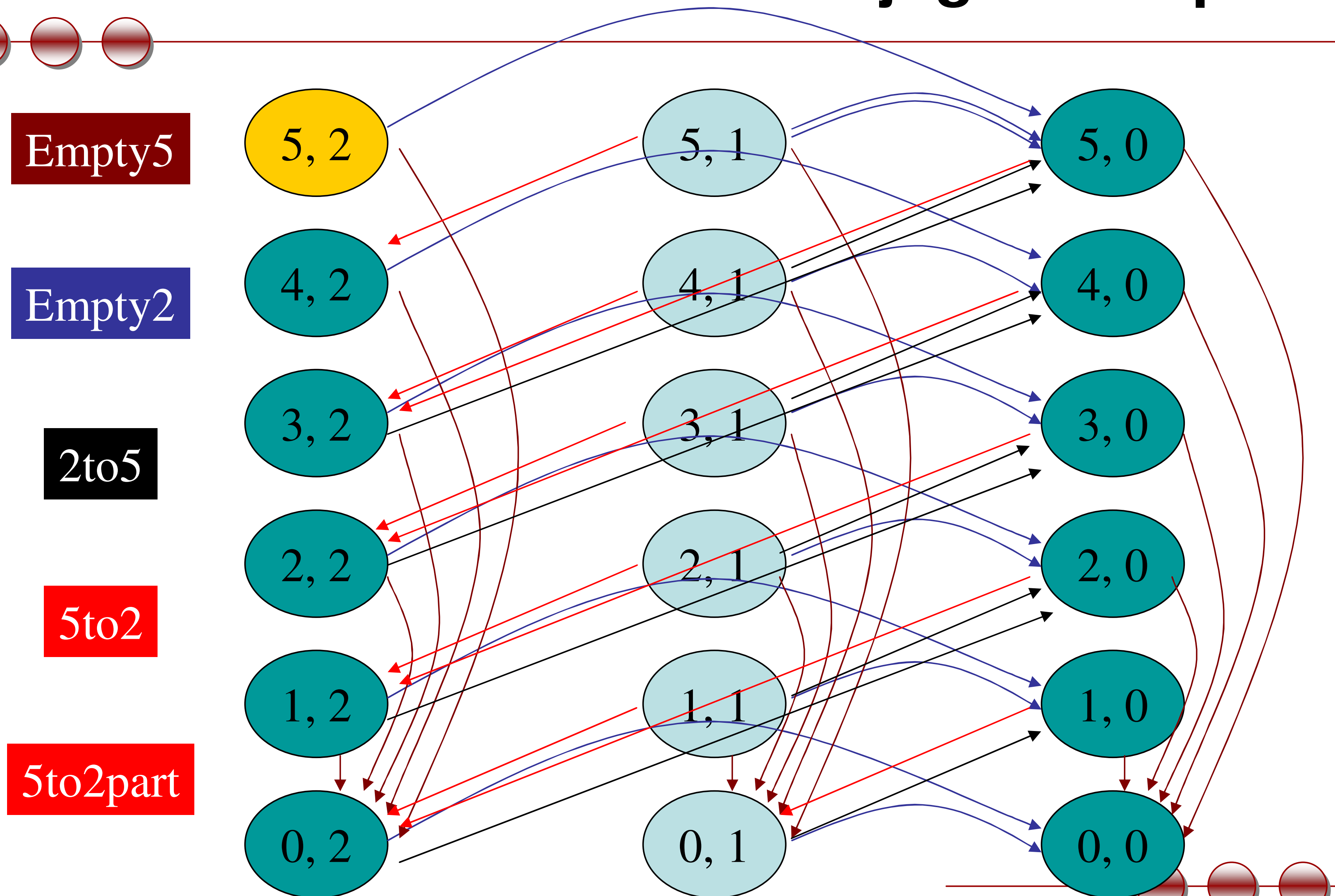
Each **node** is a data structure that contains a state description plus other information such as the parent of the node, the name of the operator that generated the node from that parent, and other bookkeeping data

Each **arc** corresponds to an instance of one of the operators. When the operator is applied to the state associated with the arc's source node, then the resulting state is the state associated with the arc's destination node

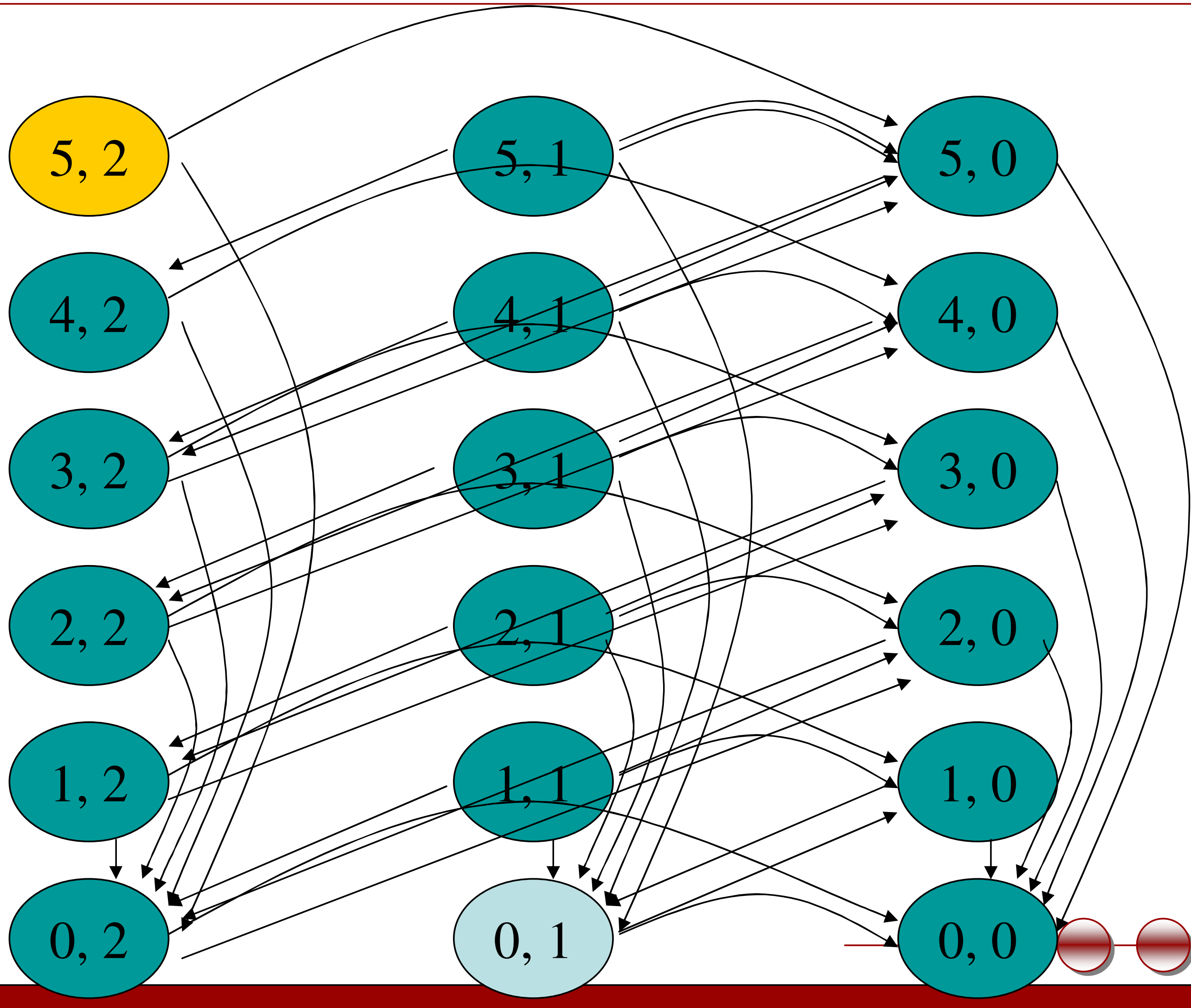
Formalizing search II

- Each arc has a fixed, positive **cost** associated with it corresponding to the cost of the operator.
- Each node has a set of **successor nodes** corresponding to all of the legal operators that can be applied at the source node's state.
  - The process of expanding a node means to generate all of the successor nodes and add them and their associated arcs to the state-space graph
- One or more nodes are designated as **start nodes.**
- A **goal test** predicate is applied to a state to determine if its associated node is a goal node.

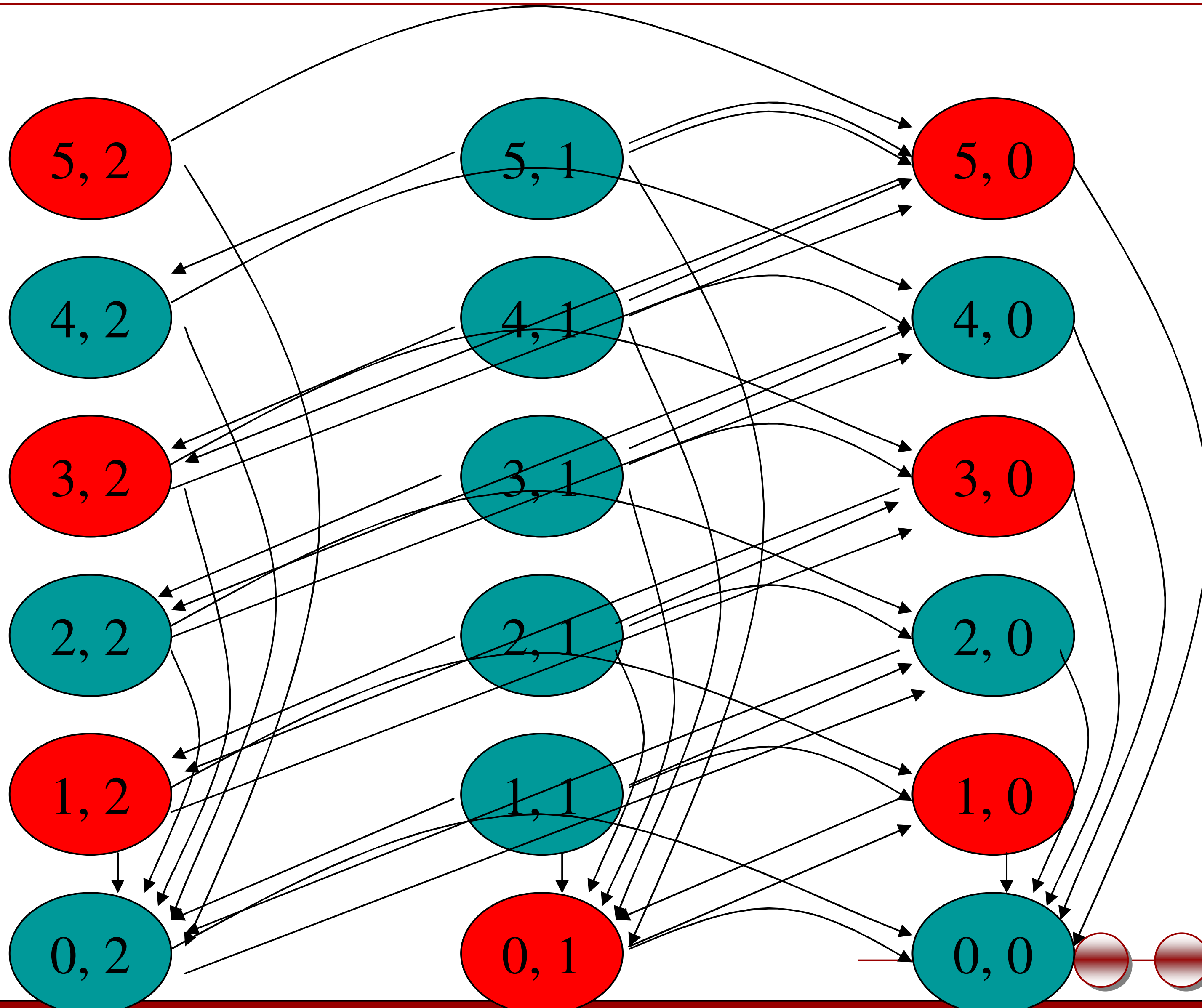
Water jug state space



Water jug solution



Water jug solution



Formalizing search III

- A **solution** is a sequence of operators 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.
  - If all arcs have the same (unit) cost, then the solution cost is just the length of the solution (number of steps / state transitions)

Formalizing search IV

- **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 find a goal node.
  - For large state spaces, it isn't practical to represent the whole space.
  - 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 found.
- Each node implicitly or explicitly represents a partial solution path (and cost of the partial solution path) from the start node to the given node.
  - In general, from this node there are many possible paths (and therefore solutions) that have this partial path as a prefix.

Search Process

- Search process constructs a search tree, where
  - **root** is the initial state and
  - **leaf nodes** are nodes
    - not yet expanded (i.e., they are in the list “nodes”) or
    - having no successors (i.e., they’re “deadends” because no operators were applicable and yet they are not goals)

Some issues

- Search tree may be infinite because of loops even if state space is small
- Return a path or a node depending on problem.
  - E.g., in cryptarithmetic return a node; in 8-puzzle return a path
- Changing the way we choose which state to expand next leads to different **search strategies**

Evaluating search strategies

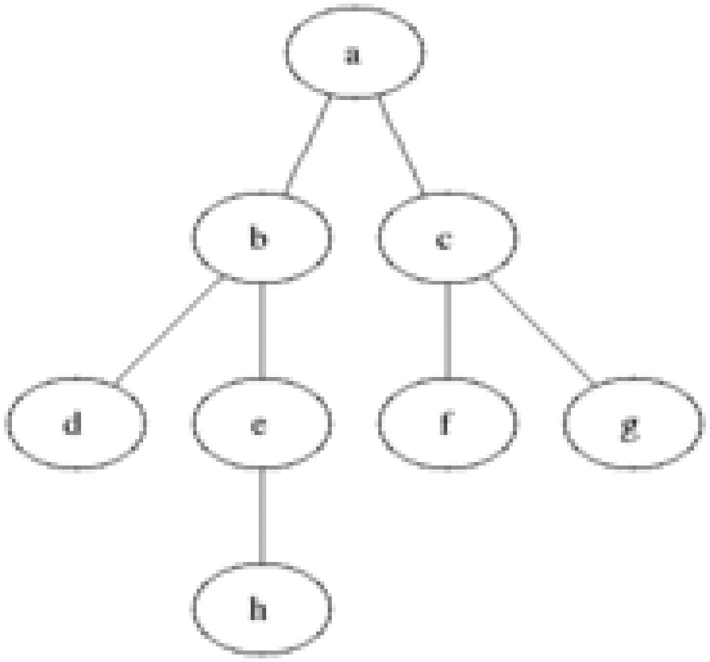
- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded
- **Space complexity**
  - How much space (memory) is used by the algorithm? Usually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

Uninformed vs. informed search

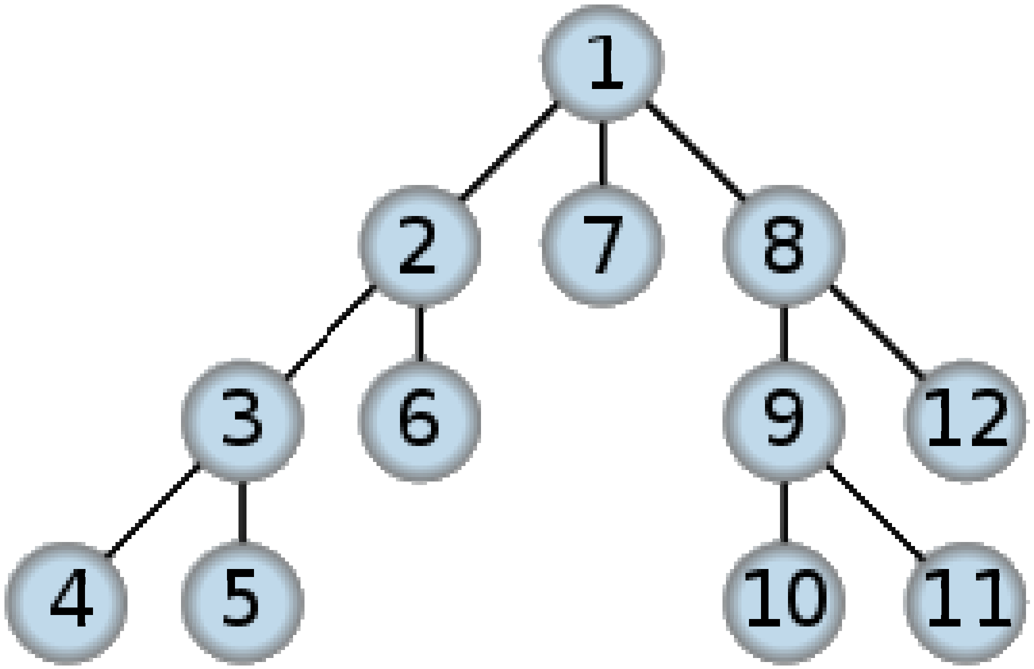
- **Uninformed search strategies**
  - Also known as "blind search," uninformed search strategies use no information about the likely "direction" of the goal node(s)
    - generate successors
    - distinguish a goal state from a non-goal state
  - Uninformed search methods: Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional
- **Informed search strategies**
  - Also known as "heuristic search," informed search strategies use information about the domain to (try to) (usually) head in the general direction of the goal node(s)
  - Informed search methods: Hill climbing, best-first, greedy search, beam search, A, A*

Uninformed Search Methods

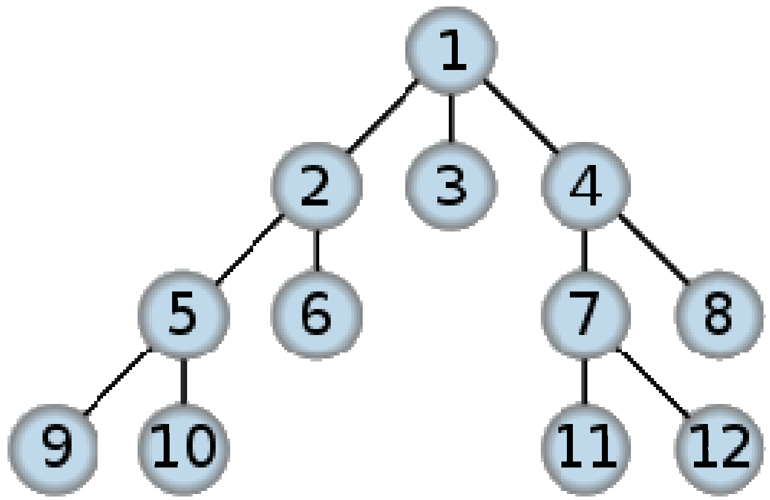
Breadth-First



Depth-First (DFS)

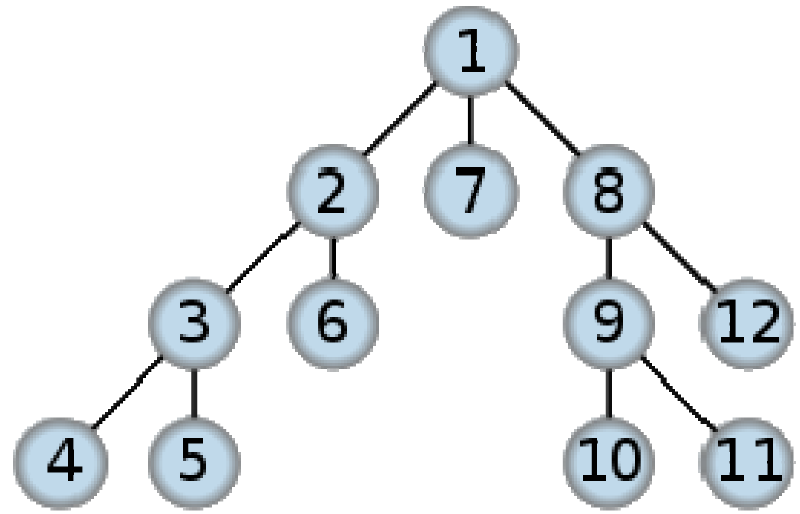


Breadth-First vs Depth-First (DFS)



Breadth-First

Exponential time and space complexity $O(b^d)$



Depth-First

Exponential time $O(b^d)$

Linear space $O(bd)$

Breadth-First

- **Complete**
- **Optimal** (i.e., admissible) if all operators have the same cost[(1)] Otherwise, not optimal but finds solution with shortest path length.
- **Exponential time and space complexity**, $O(b^d)$, where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node
- Will take a **long time to find solutions** with a large number of steps because must look at all shorter length possibilities first
  - 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 $1 + 10 + 100 + 1000 + \ldots + 10^{12} = (10^{13} - 1)/9 = O(10^{12})$ nodes in the complete search tree. If BFS expands 1000 nodes/sec and each node uses 100 bytes of storage, then BFS will take 35 years to run in the worst case, and it will use 111 terabytes of memory!

(1) more strictly if the path cost is a non decreasing function of the depth of the node

Depth-First (DFS)

- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D ("depth-limited search")
- **Complete in Finite State Spaces, otherwise Not complete** (with or without cycle detection, and with or without a cutoff depth)
- **Exponential time**, $O(b^d)$, but only **linear space**, $O(bd)$
- Can find **long solutions quickly** if lucky (and **short solutions slowly** if unlucky!)

Uniform-Cost (UCS)

- Breadth-First search not optimal when steps have different costs
- Store nodes in the frontier by **path cost**.
  - Called *"Dijkstra's Algorithm"* in the algorithms literature and similar to *"Branch and Bound Algorithm"* in operations research literature
- **Complete (*), Optimal/Admissible** (*)
- **Exponential time and space complexity**, $O(b^d)$

Depth-First Iterative Deepening (DFID)

- 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.
- **Complete**[1] (if depth limit >= shallowest goal).
- **Optimal/Admissible** if all operators have the same cost. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- Time complexity is still exponential, $O(b^d)$
- **Linear space complexity**, $O(bd)$, like DFS

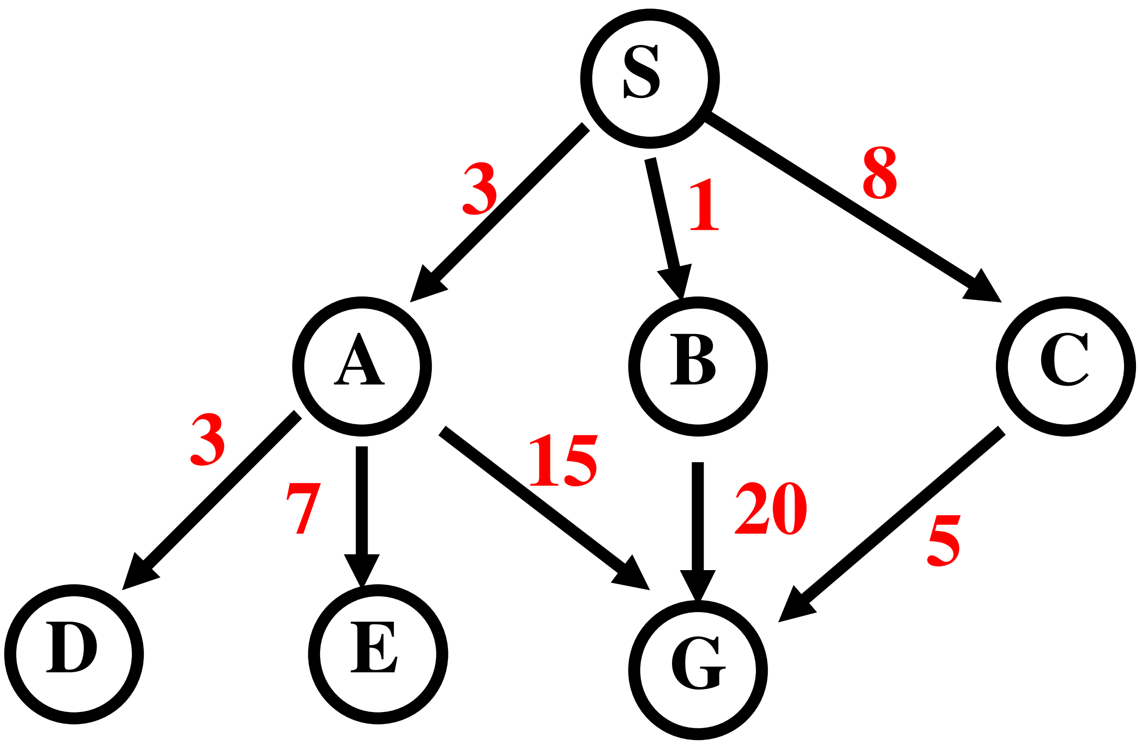
(1) solves the infinite-path problem

Depth-First Iterative Deepening

- Has advantage of BFS (i.e., completeness) and also advantages of DFS (i.e., limited space and finds longer paths more quickly)
- Generally preferred for **large state spaces** where **solution depth is unknown**

Uninformed Search Results

Example for illustrating uninformed search strategies



Depth-First Search

Expanded node	**Nodes list**
	$\{ S^0 \}$
S^0	$\{ A^3\ B^1\ C^8 \}$
A^3	$\{ D^6\ E^{10}\ G^{18}\ B^1\ C^8 \}$
D^6	$\{ E^{10}\ G^{18}\ B^1\ C^8 \}$
E^{10}	$\{ G^{18}\ B^1\ C^8 \}$
G^{18}	$\{ B^1\ C^8 \}$

Solution path found is S A G, cost 18

Number of nodes expanded (including goal node) = 5

Breadth-First Search

Expanded node	Nodes list
	{ S^0 }
S^0	{ A^3 B^1 C^8 }
A^3	{ B^1 C^8 D^6 E^{10} G^{18} }
B^1	{ C^8 D^6 E^{10} G^{18} G^{21} }
C^8	{ D^6 E^{10} G^{18} G^{21} G^{13} }
D^6	{ E^{10} G^{18} G^{21} G^{13} }
E^{10}	{ G^{18} G^{21} G^{13} }
G^{18}	{ G^{21} G^{13} }

Solution path found is S A G , cost 18

Number of nodes expanded (including goal node) = 7

Uniform-Cost Search

Expanded node	**Nodes list**
	$\{ S^0 \}$
S^0	$\{ B^1\ A^3\ C^8 \}$
B^1	$\{ A^3\ C^8\ G^{21} \}$
A^3	$\{ D^6\ C^8\ E^{10}\ G^{18}\ G^{21} \}$
D^6	$\{ C^8\ E^{10}\ G^{18}\ G^1 \}$
C^8	$\{ E^{10}\ G^{13}\ G^{18}\ G^{21} \}$
E^{10}	$\{ G^{13}\ G^{18}\ G^{21} \}$
G^{13}	$\{ G^{18}\ G^{21} \}$

Solution path found is S B G, cost 13

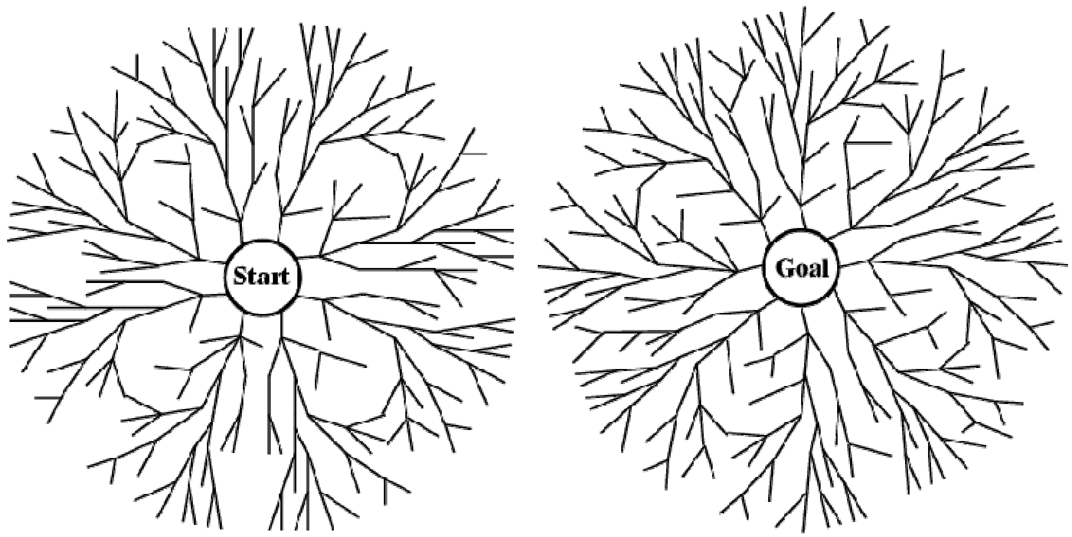
Number of nodes expanded (including goal node) = 7

How they perform

- **Depth-First Search:**
  - Expanded nodes: S A D E G
  - Solution found: S A G (cost 18)
- **Breadth-First Search**:
  - Expanded nodes: S A B C D E G
  - Solution found: S A G (cost 18)
- **Uniform-Cost Search**:
  - Expanded nodes: S A D B C E G
  - Solution found: S B G (cost 13)

  This is the only uninformed search that worries about costs.
- **Iterative-Deepening Search**:
  - nodes expanded: S S A B C S A D E G
  - Solution found: S A G (cost 18)

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.
- Requires the ability to generate "predecessor" states.
- Can (sometimes) lead to finding a solution more quickly.

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

Avoiding Repeated States

- In increasing order of effectiveness in reducing size of state space and with increasing computational costs:
  1. Do not return to the state you just came from.
  2. Do not create paths with cycles in them.
  3. Do not generate any state that was ever created before.
- Net effect depends on frequency of "loops" in state space.

Thanks for coming -- see you next Thursday!