Artificial Intelligence

Lecture 2：Search

Xiaojin Gong

2022-02-28

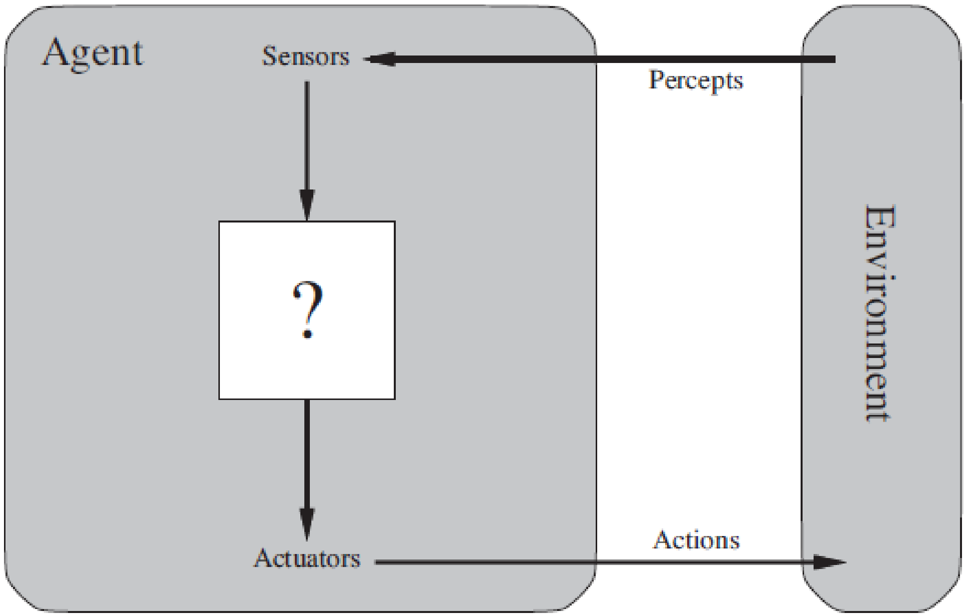
Credits: AI Courses in Berkeley

Review

Thinking Humanly	Thinking Rationally
"The exciting new effort to make computers think ... *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI ...is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Review

- Rational Agents
  - Structure



Task environment
- Performance
- Environment
- Actuators
- Sensors

Agent = Architecture + Program

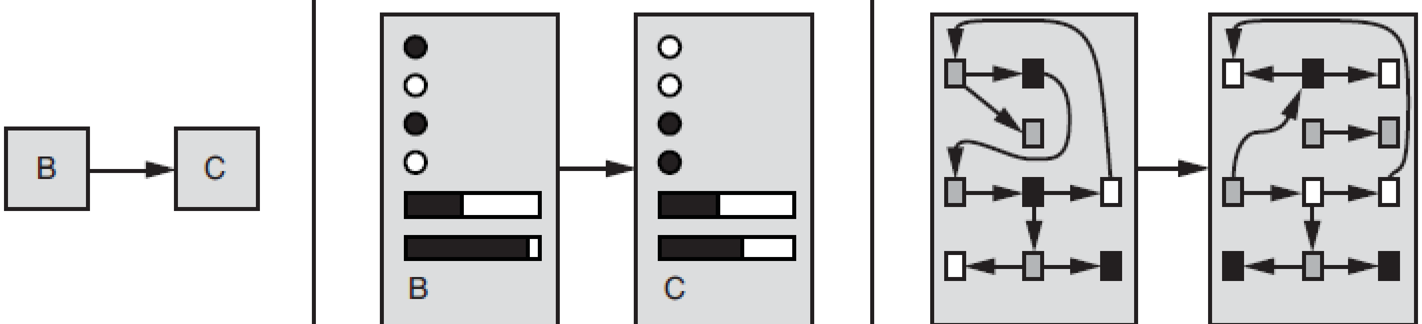
Review

- Rational Agents
  - Representation of Environments

Search
Game-playing
Markov decision processes

Constraint satisfaction
Bayesian network
Machine learning

First-order logic
Knowledge-based learning
Natural language processing



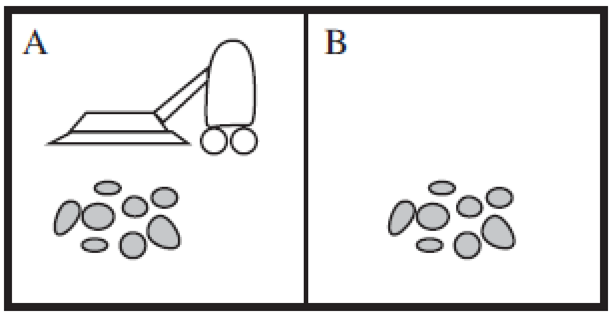
(a) Atomic (b) Factored (b) Structured

Outline

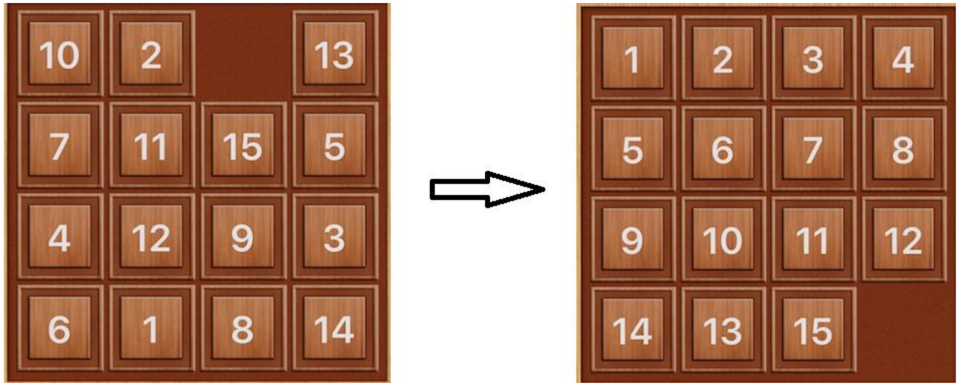
- Problem-Solving Agents
  - Problem Formulation
  - Solving Problems by Searching
    - Uninformed Search
      - Breadth First Search
      - Depth First Search
      - Iterative Deepening Search
      - Cost-sensitive Search
    - Informed Search (Heuristic Search)
      - Greedy Search
      - A* Search

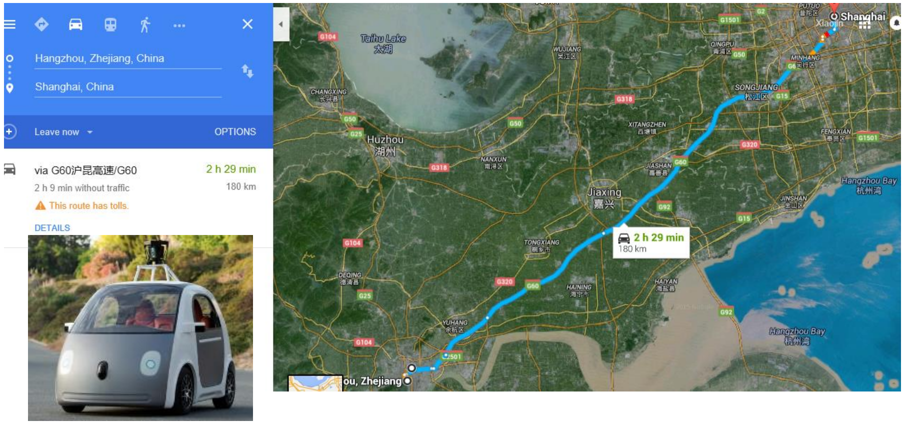
Problem-Solving Agents

- **Observable, discrete, known, deterministic**



A vacuum-cleaner world





Problem-Solving Agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
   persistent: seq, an action sequence, initially empty
               state, some description of the current world state
               goal, a goal, initially null
               problem, a problem formulation

   state ← UPDATE-STATE(state, percept)
   if seq is empty then
       goal ← FORMULATE-GOAL(state)
       problem ← FORMULATE-PROBLEM(state, goal)
       seq ← SEARCH(problem)
       if seq = failure then return a null action
   action ← FIRST(seq)
   seq ← REST(seq)
   return action
```

Formulate

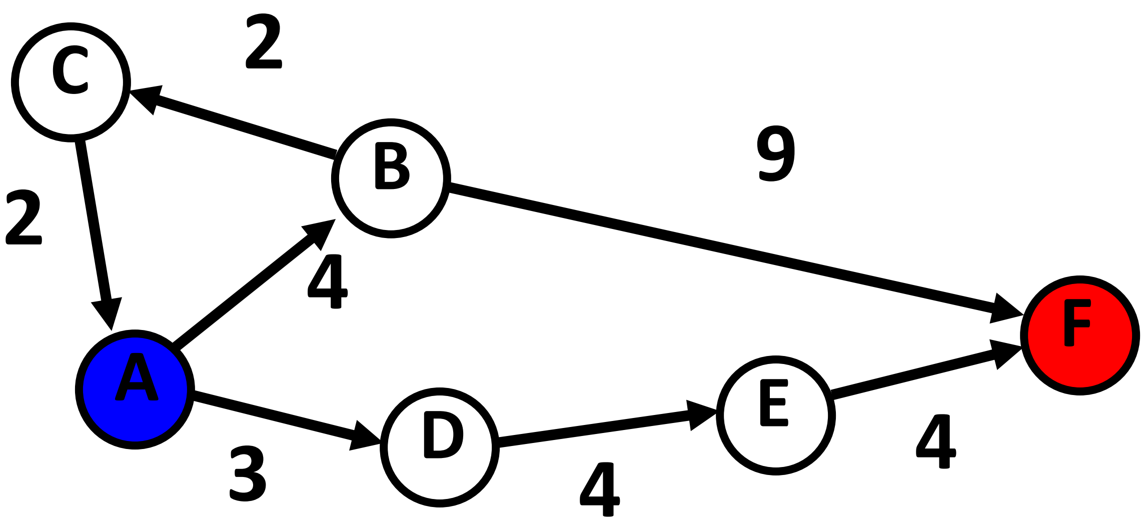
Search

Execute

Problem Formulation

- A search problem consists of:
  - States
  - Initial state
  - Goal test
  - Actions
  - Transition model
  - Path cost
- A solution is a sequence of actions which transforms the initial state to a goal state

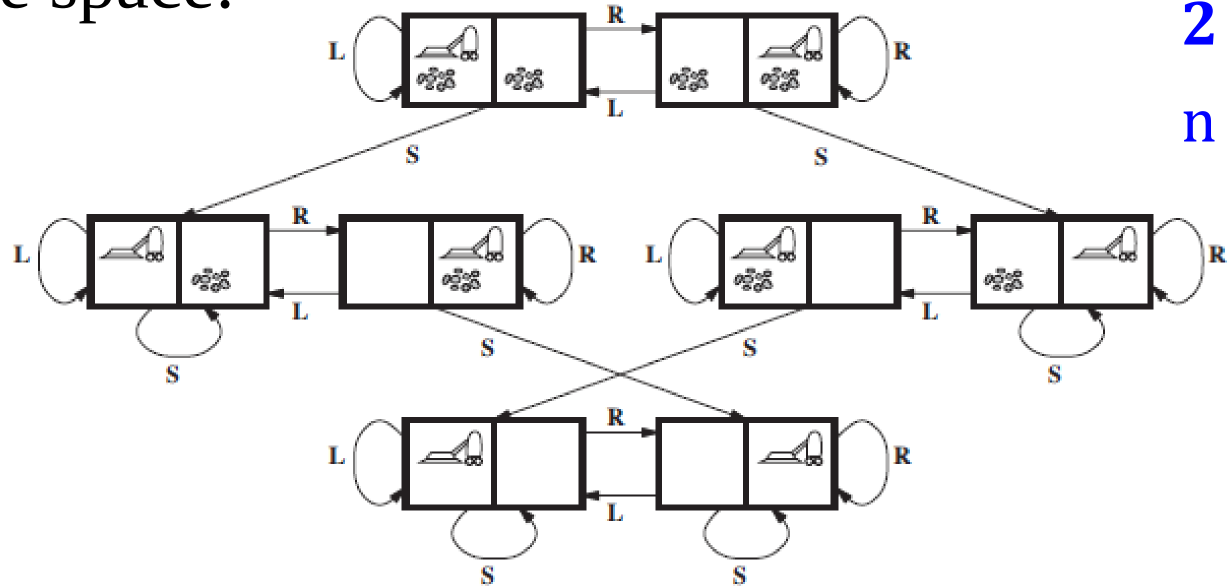
Example: Route-finding Problem



- Initial state:
  - A
- Goal test:
  - Is state == F?
- State space:
  - Vertices
- Actions:
  - Go to adjacent vertex with cost = weight
- Solution ?

Example: The Vacuum World

- The state space:



$2 * 2^2$

$n * 2^n$

- Goal state:

  This checks whether all the squares are clean

- Actions:

  Move Left, Right, Suck

- Cost:

  Each step costs 1

Example: 8-puzzle problem

- Initial state:

1	2	3
4		6
7	5	8

- Goal state:

1	2	3
4	5	6
7	8	

- The state space: **9!/2 = 181,440**

1	2	3
4		6
7	5	8

1		3
4	2	6
7	5	8

1	2	3
	4	6
7	5	8

…

1	2	3
4	5	6
7	8	

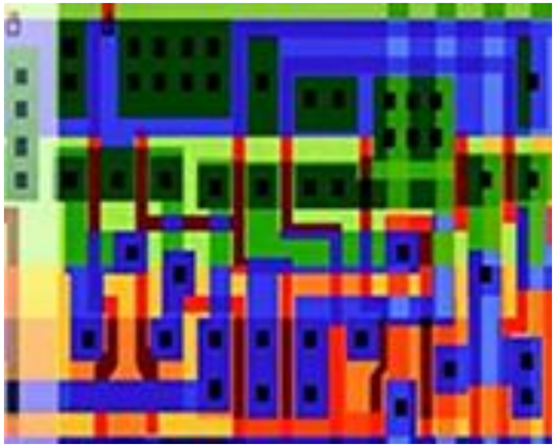
- Actions:

  Move the blank space Left, Right, Up, or Down

- Cost:

  Each step costs 1

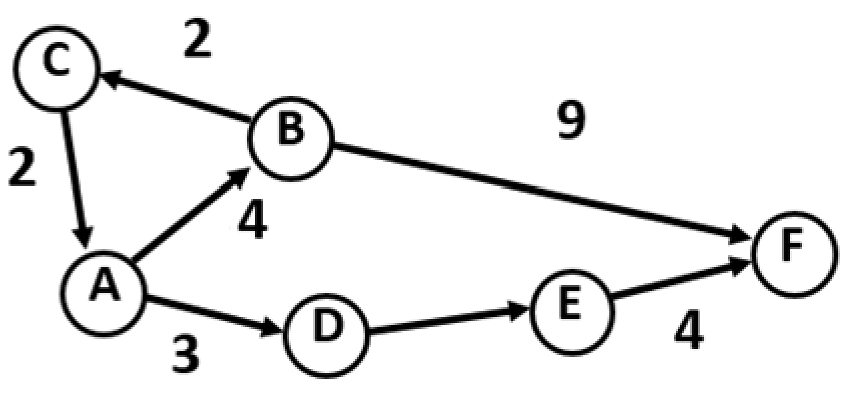
Example: VLSI Layout



- Initial state:
  - No components placed
- Goal test:
  - All components placed
- States:
  - Positions of components, wires on a chip
- Action:
  - Place components, route wire
- Path cost:
  - Distance, capacity, number of connections per component

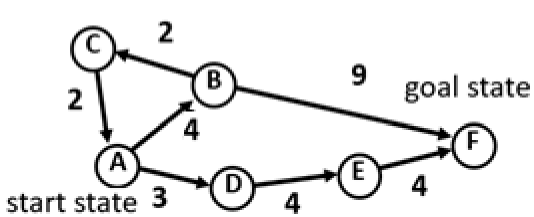
State Graphs and Search Trees

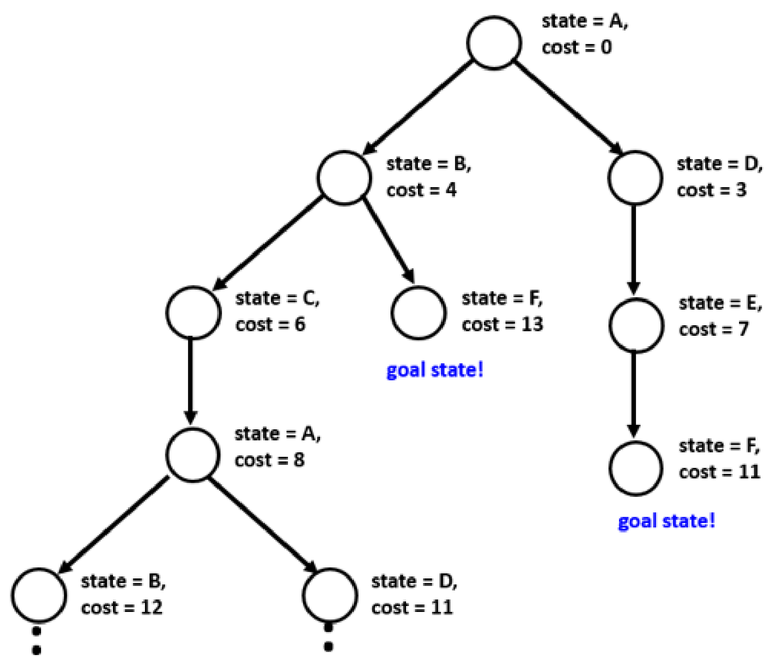
- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state graph, each state occurs only once
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



State Graphs and Search Trees

- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those stat
- For most problems, we can never actually build the whole tree

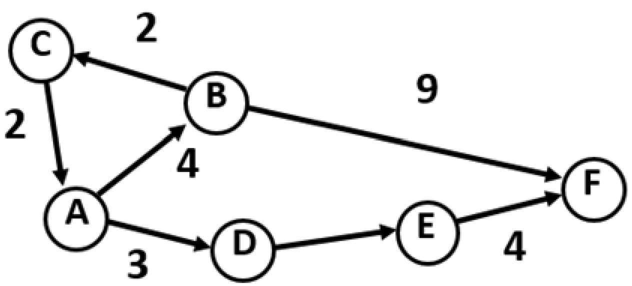




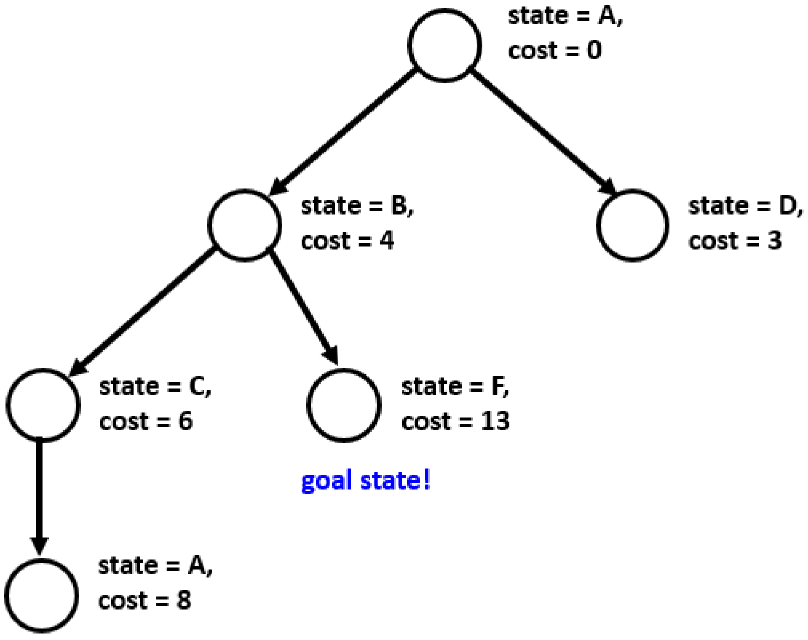
State Graphs and Search Trees

- Each NODE in the search tree is an entire PATH in the problem graph.
- We construct both on demand – and we construct as little as possible.

State graph

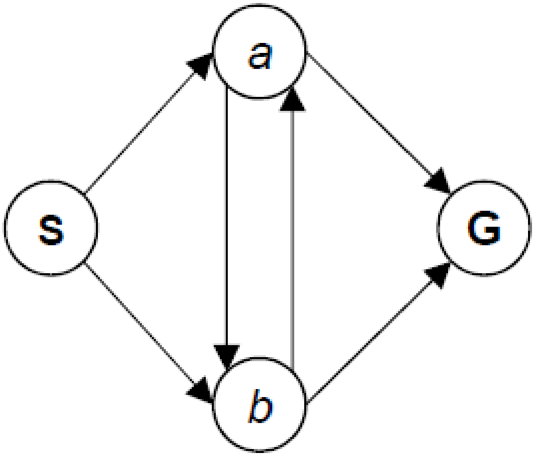


Search tree

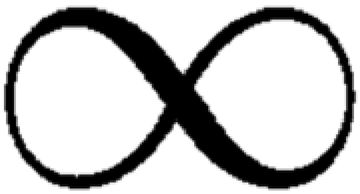


State Graphs and Search Trees

Consider this 4-state graph:



How big is its search tree (from S)?



Important: Lots of repeated structure in the search tree!

Generic Search Algorithm

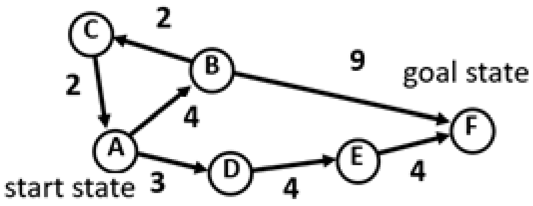
- Searching with a search tree – tree search

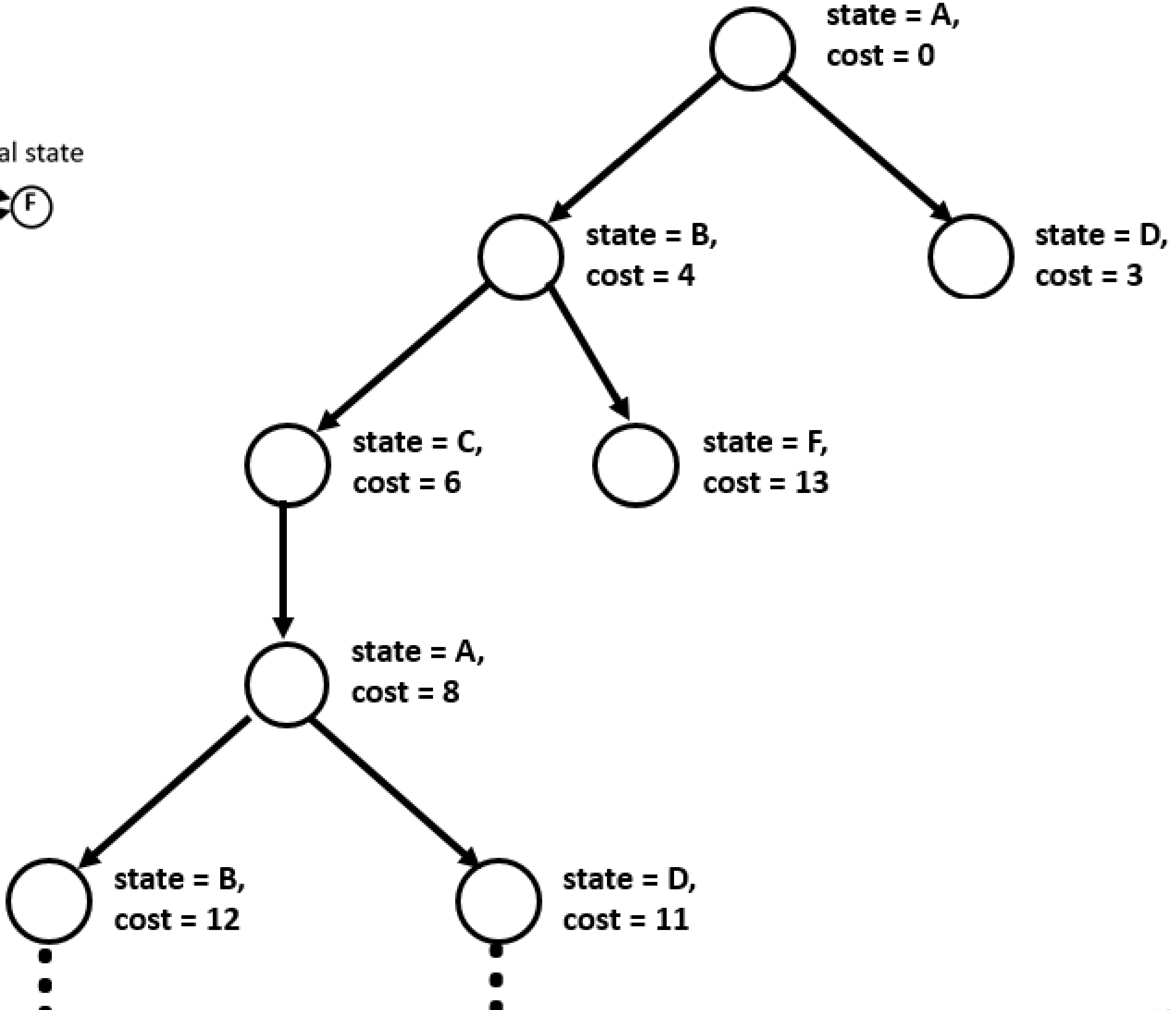
```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

- Important ideas:
  - Frontier
  - Expansion
  - Exploration strategy
- Main question: which leaf nodes to explore?

Generic Search Algorithm

- Tree search





Generic Search Algorithm

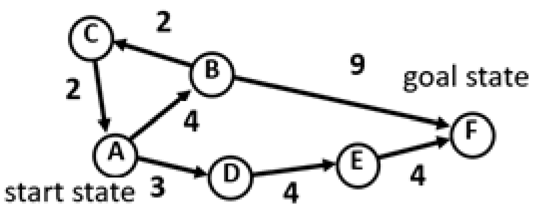
- Searching with a search tree – graph search

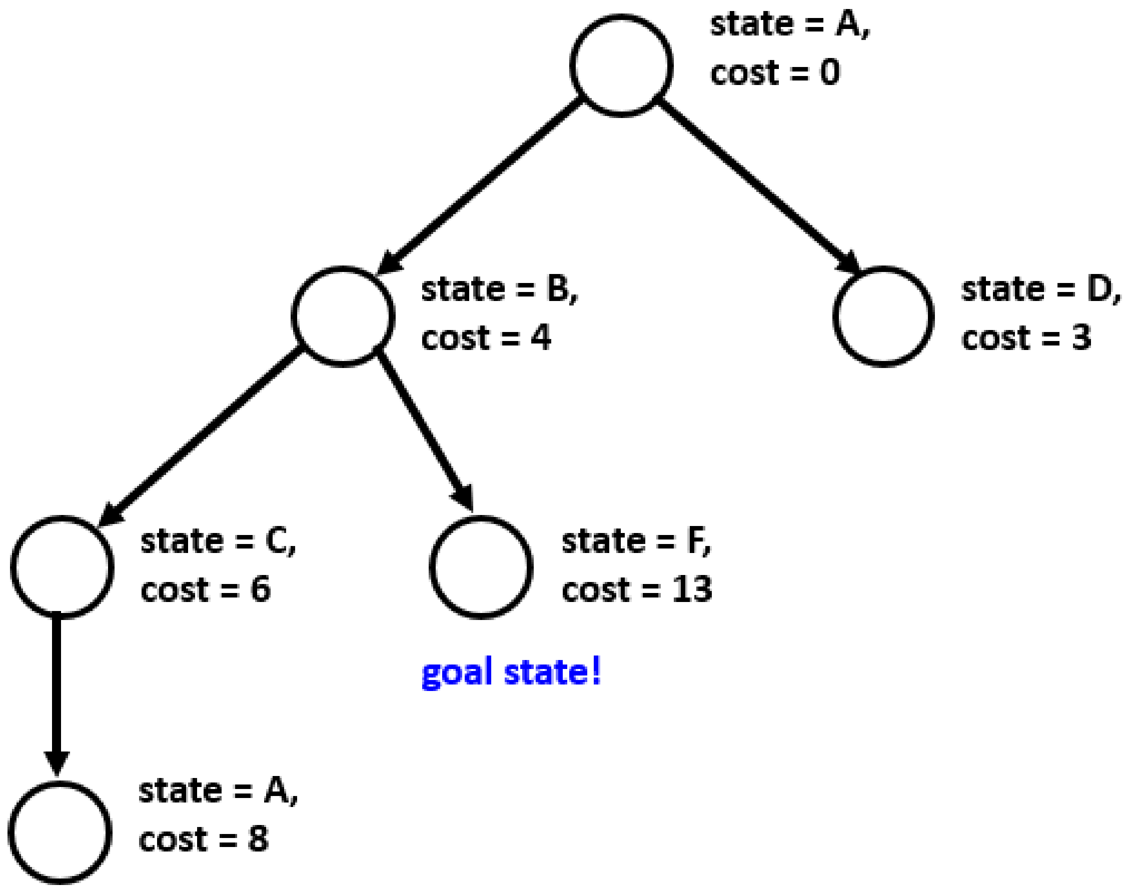
```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

- Important ideas:
  - Frontier
  - Explored set
  - Expansion
  - Exploration strategy
- Main question: which leaf nodes to explore?

Generic Search Algorithm

- Graph search





Search Strategies

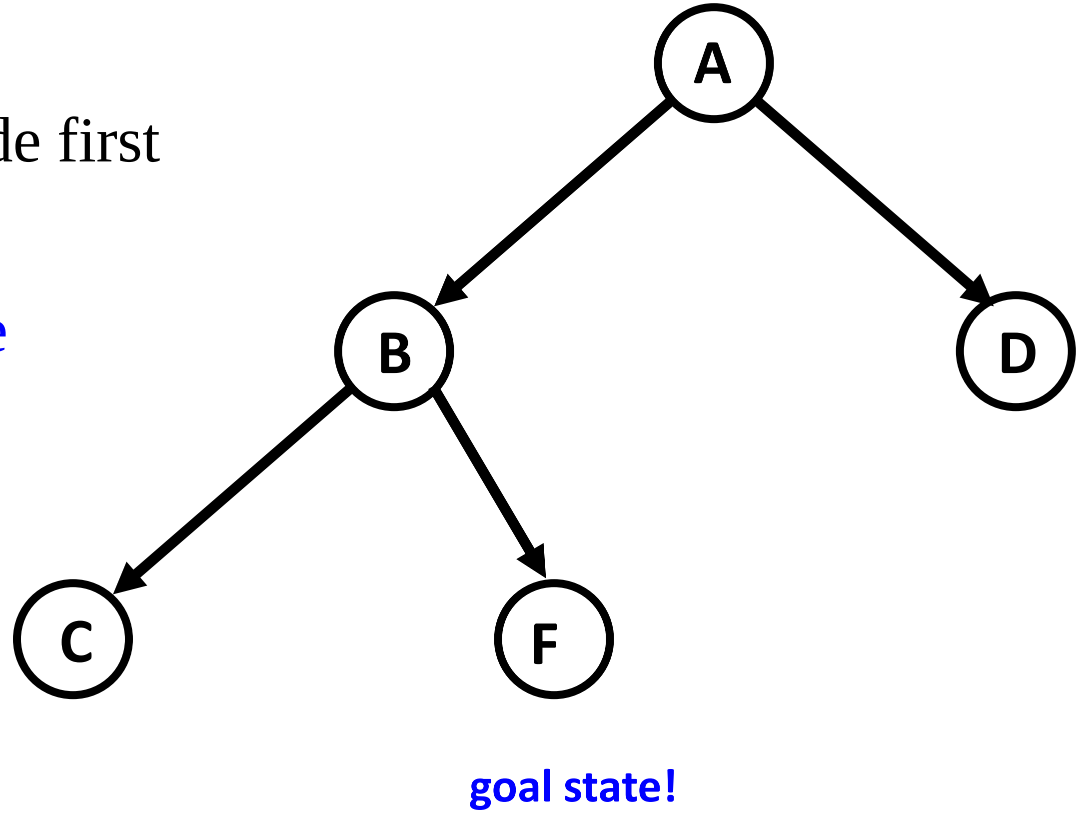
- Performance measurements:
  - Completeness: Guaranteed to find a solution if one exists?
  - Optimality: Guaranteed to find the optimal solution?
  - Time complexity:
  - Space complexity:
- Strategies:
  - Uninformed search: (blind search)
  - Informed search: (heuristic search)

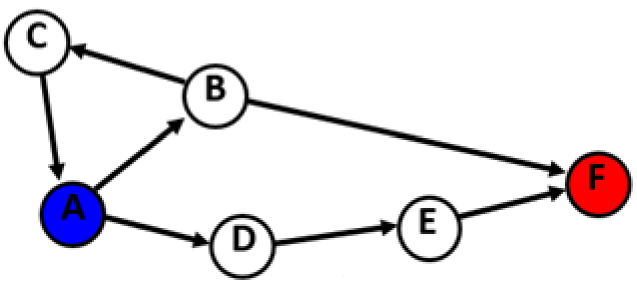
Uninformed Search Strategies

- Uninformed Search:
  - Can generate successors and distinguish a goal state
  - No additional information about states
- Uninformed Search Strategies:
  - Breadth first search
  - Depth first search
  - Iterative deepening depth-first search
  - Uniform-cost search

Breadth First Search

- Strategy:
  - Expand a shallowest node first
- Implementation:
  - Frontier is a FIFO queue

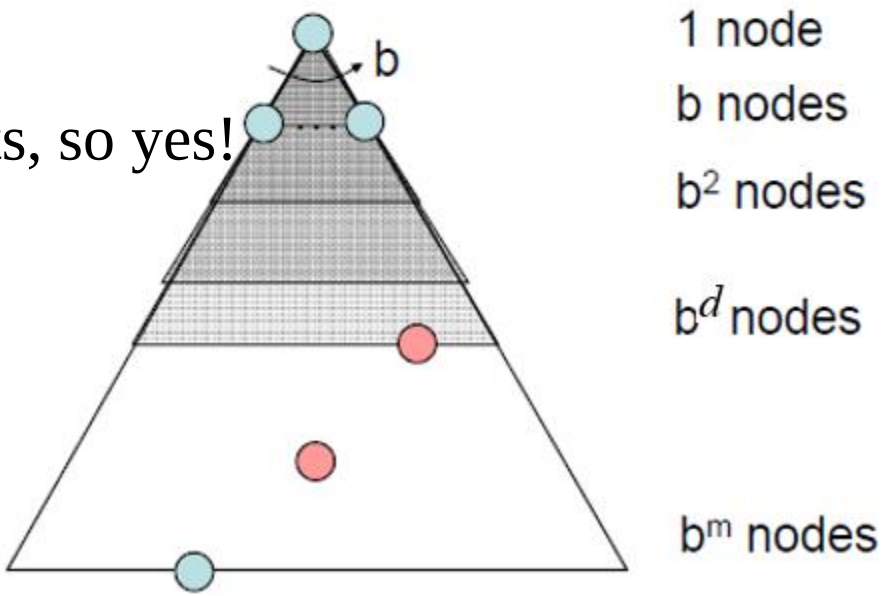




Goal test is applied to a node when it is generated

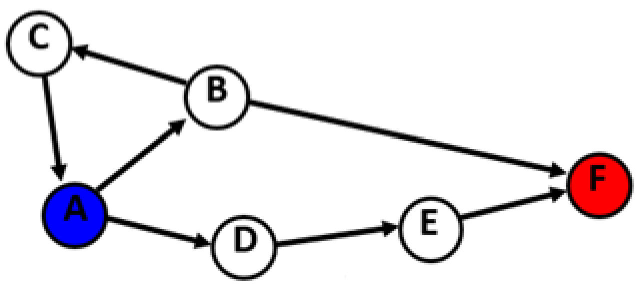
Breadth First Search

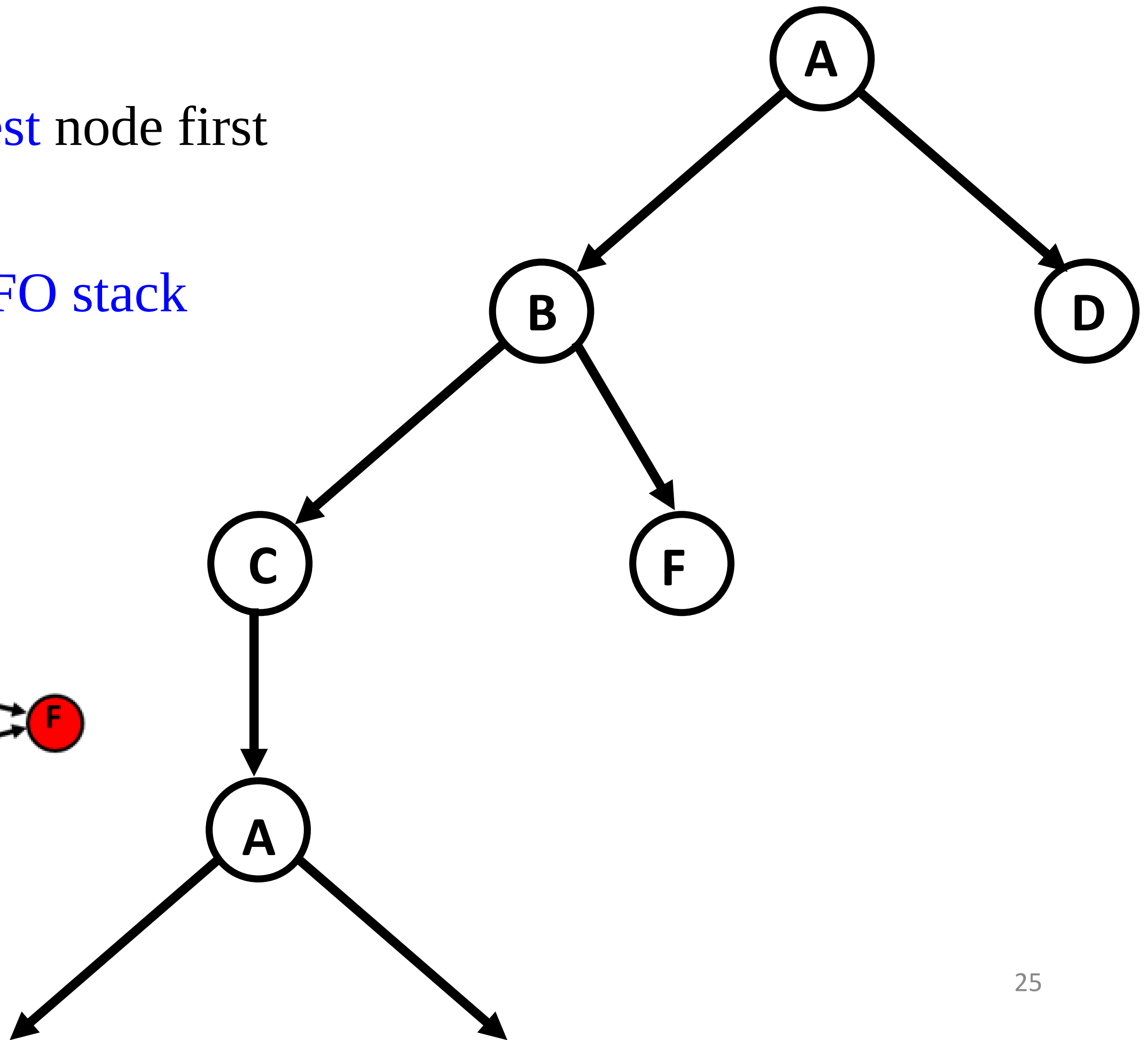
- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be *d*
  - Search takes time $O(b^d)=b+b^2+b^3+\ldots+b^d$
- How much space does the frontier take?
  - Dominated by the size of frontier, $O(b^d)$
- Is it complete?
  - *d* must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1



Depth First Search

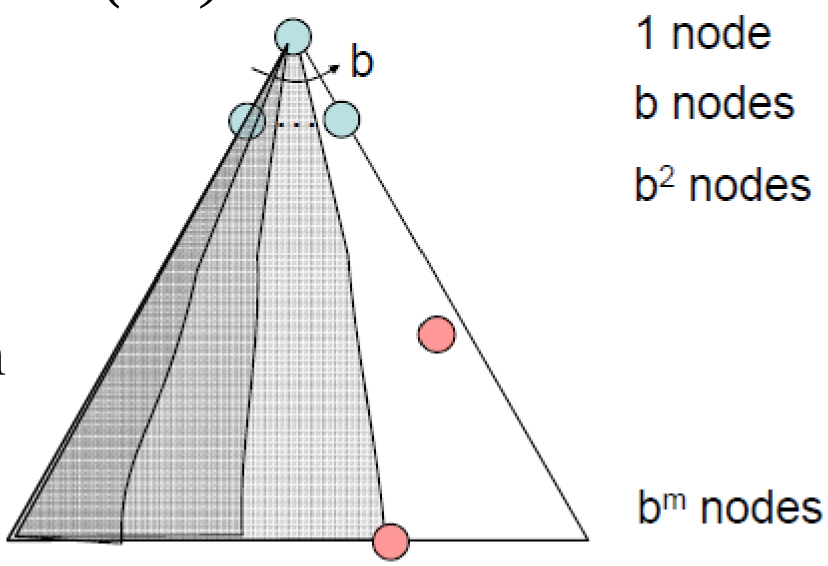
- Strategy:
  - Expand a deepest node first
- Implementation:
  - Frontier is a LIFO stack





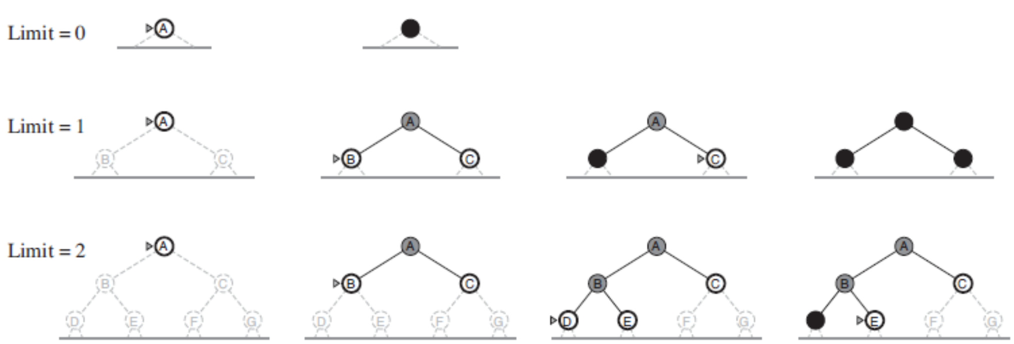
Depth First Search

- What nodes does DFS expand?
  - Some left prefix of the tree
  - Could process the whole tree
  - If m is finite, takes time $O(b^m)$
- How much space does the frontier take?
  - Only has siblings on path to root, so $O(bm)$
- Is it complete?
  - m could be infinite
- Is it optimal?
  - No, it finds the "leftmost" solution



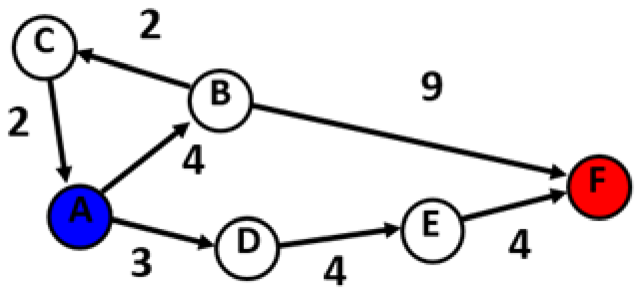
Iterative Deepening Depth-First Search

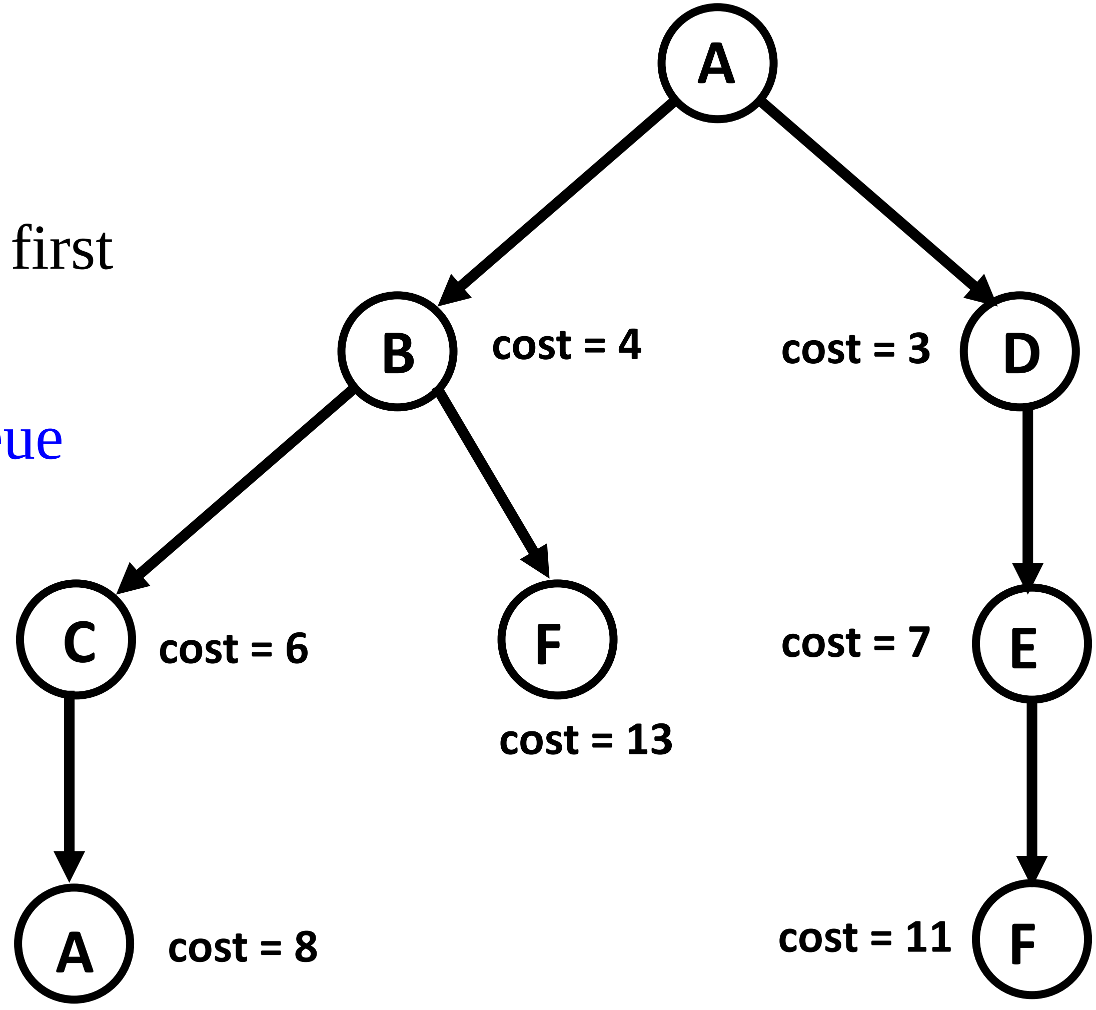
- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - …
  - Run a DFS with depth limit *d*. …
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad.



Uniform-cost Search

- Cost-Sensitive Search
- Strategy:
  - Expand a cheapest node first
- Implementation:
  - Frontier is a priority queue

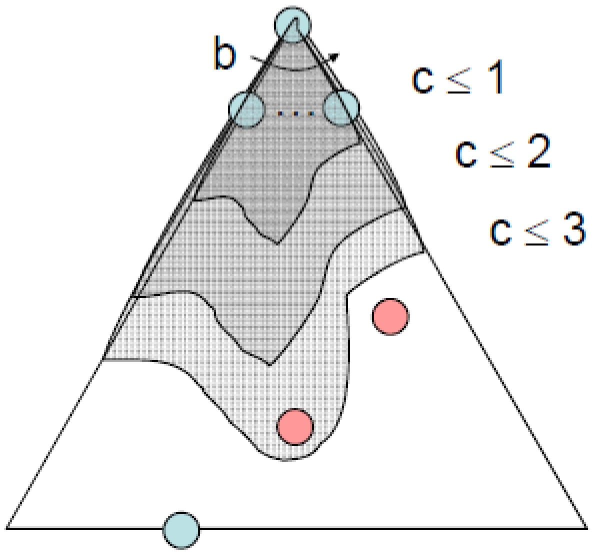




Goal test is applied to a node when it is selected for expansion

Uniform-cost Search

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution
  - If that solution costs C^* and arcs cost at least ε , then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{1+C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the frontier take?
  - $O(b^{1+C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes!



Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^{a}	Yes^{a,b}	No	No	Yes^{a}	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^{c}	Yes	No	No	Yes^{c}	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

Informed Search Strategies

An informed search strategy uses problem-specific knowledge beyond the definition of the problem itself.

- Search heuristics
- Best-first search (Greedy search)
- A* Search

Search Heuristics

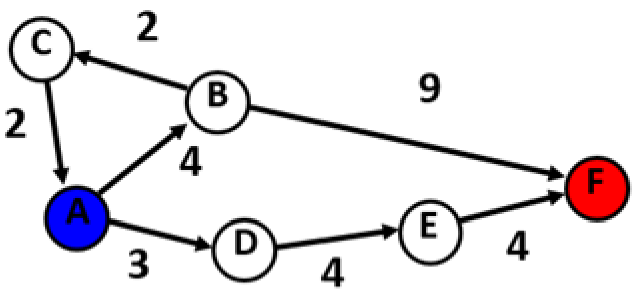
- A heuristic function is:
  - A function that estimates how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing
- The heuristic function is denoted by $h(n)$, which is
  - Non-negative, problem-specific
  - $h(n)=0$ for goal nodes

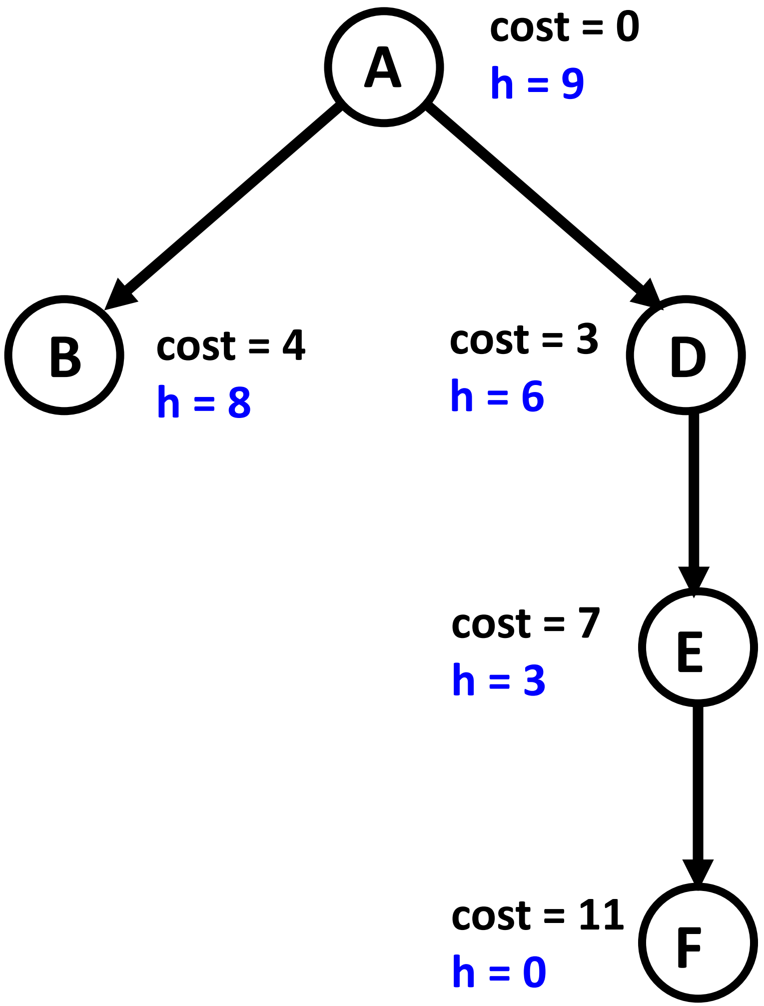
Greedy Best-First Search

- Strategy:
  - Expand a node with lowest $h(n)$ first

  - $h(n)$: straight-line distance

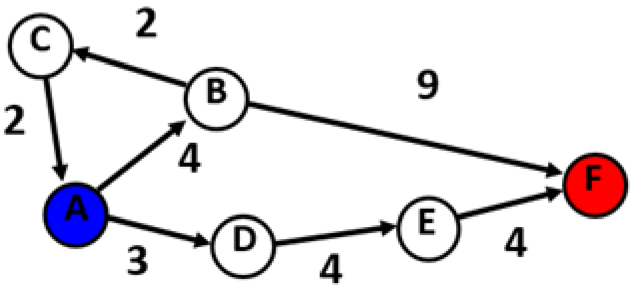
    $h(A) = 9, h(B) = 8, h(C) = 9,$
    $h(D) = 6, h(E) = 3, h(F) = 0$





Greedy Best-First Search

- Greedy search can rapidly finds the optimal solution
- What can go wrong?
  - greedy evaluates the promise of a node only by how far is left to go, does not take cost occurred already into account



h(A) = 9, h(B) = 5, h(C) = 9,
h(D) = 6, h(E) = 3, h(F) = 0

- Not optimal
- Incomplete (tree search)
- The worst-case time and space complexity is $O(b^m)$

A* Search

- Strategy:
  - Let $g(n)$ be cost incurred already on path to n
  - Expand nodes with lowest $g(n)+h(n)$ first

- Uniform-cost orders by path cost, or backward cost $g(n)$
- Greedy orders by goal proximity, or forward cost $h(n)$
- A* orders by the sum $f(n) = g(n) + h(n)$

A* Search

- Is A* optimal?
  - If the heuristic is admissible, the tree-search version of A* is optimal
    - A heuristic $h(n)$ is admissible if

$$0 \leq h(n) \leq h^*(\mathrm{n})$$

where $h^*(\mathrm{n})$ is the true cost to a nearest goal

    - Example: straight-line distance is admissible
    - Admissible heuristic means that A* is always optimistic

A* Search

- Is A* optimal?
  - If the heuristic is admissible, the tree-search version of A* is optimal

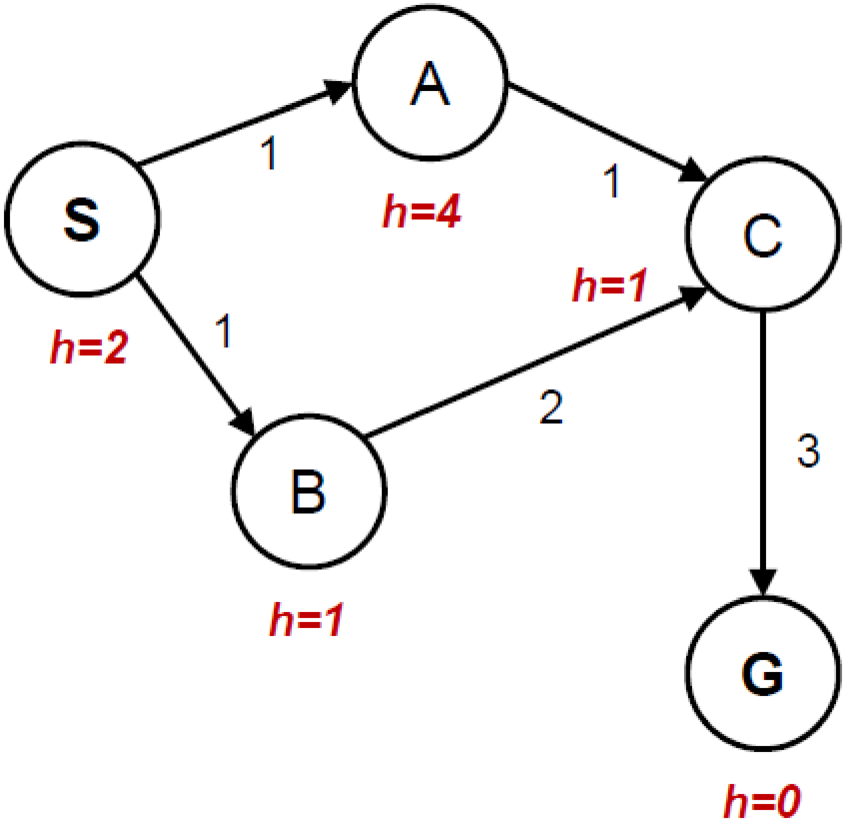
Proof:

- Suppose a suboptimal solution node n with solution value $C > C^*$ is about to be expanded (where C^* is optimal);
- Let n^* be an optimal solution node (perhaps not yet discovered);
- There must be some node n' that is currently in the frontier and on the path to n^*;
- We have $g(n) = C > C^* = g(n^*) \geq g(n') + h(n')$;
- But then, n' should be expanded first (contradiction).

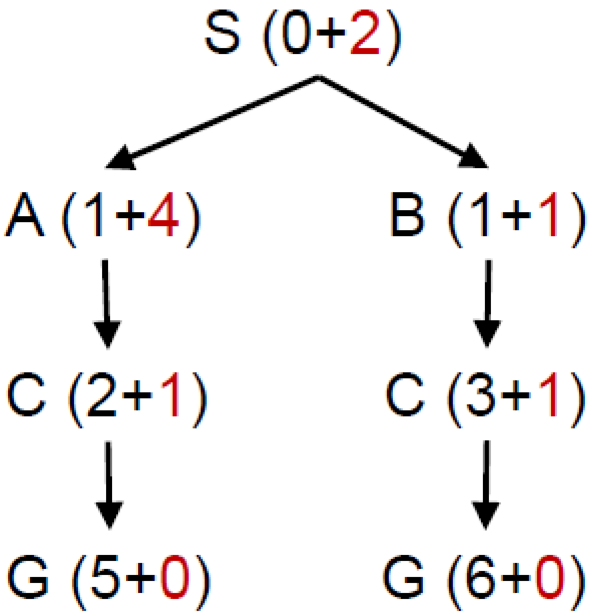
A* Search

- A* graph search goes wrong?

State space graph



Search tree

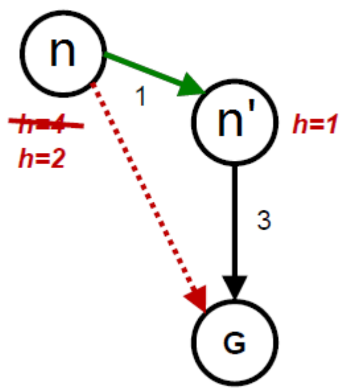


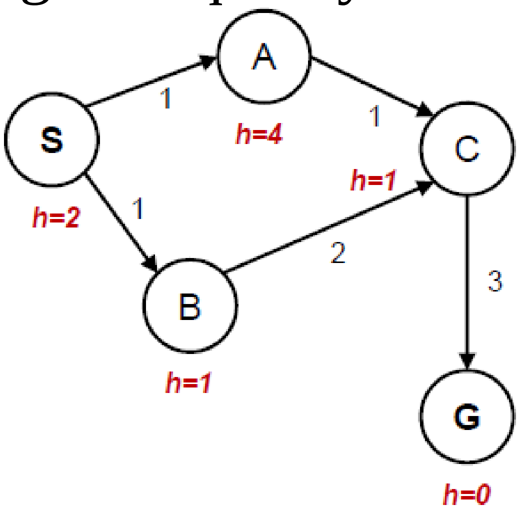
A* Search

- Is A* optimal?
  - If the heuristic is consistent, the graph-search version of A* is optima
    - A heuristic $h(n)$ is consistent (monotonicity) if for every node n and every successor n' of n generated by any action a,

$$h(n) \leq c(n, a, n') + h(\mathrm{n}')$$

It is a form of the general triangle inequality.

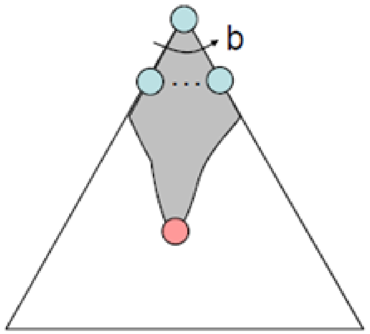




- Every consistent heuristic is also admissible

A* Search

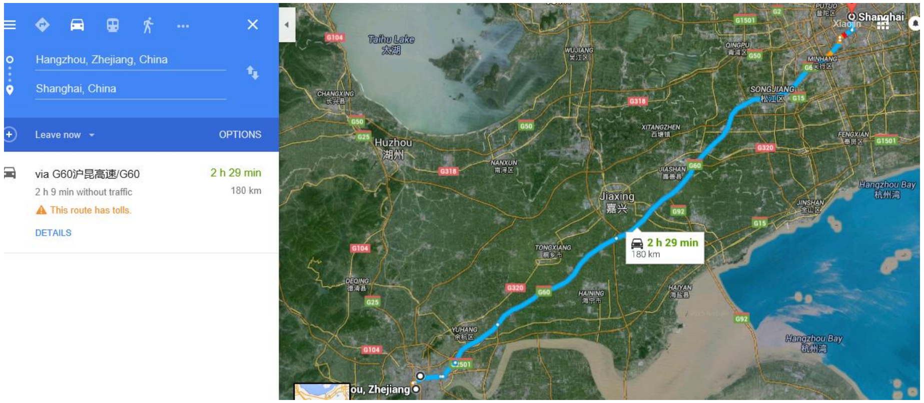
- Is A* optimal?
  - If the heuristic is admissible, the tree-search version of A* is optimal
  - If the heuristic is consistent, the graph-search version A* is optimal



- Is A* complete?
  - Yes!
- A* is optimally efficient for any consistent heuristic.
  - That is, no other optimal algorithm is guaranteed to expand fewer nodes than A*.

A* Search

- A* applications:
  - Video games
  - Pathing / routing problems
  - Resource planning problems
  - Robot motion planning
  - Language analysis
  - Machine translation
  - Speech recognition
  - …



How to Design a Heuristic Function

- Example: 8-puzzle problem

  - Initial state:

1	2	3
4		6
7	5	8

  - Goal state:

1	2	3
4	5	6
7	8	

  - The state space: **9!/2**

1	2	3
4		6
7	5	8

1		3
4	2	6
7	5	8

1	2	3
	4	6
7	5	8

…

1	2	3
4	5	6
7	8	

  - Actions:
    Move the blank space Left, Right, Up, or Down

  - Cost:
    Each step costs 1

Example: 8-puzzle problem

- How to design a heuristic function?
  1. *h1* = the number of tiles misplaced
  2. *h2* = the sum of distances of the tiles from their goal positions

- Initial state:

7	2	4
5		6
8	3	1

- Goal state:

	1	2
3	4	5
6	7	8

h1 = 8

h2 =3+1+2+2+2+3+3+2=18

Example: 8-puzzle problem

- How to evaluate the quality of a heuristic function?
  - The effective branching factor b*

  If the total number of nodes generated by A* is N, and the solution depth is d, then

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Example: 8-puzzle problem

- How to design a heuristic function?
  1. Generating from relaxed problems
     - The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.
  2. Generating from sub-problems
     - Initial state:

*	2	3
	*	4
*	6	*

     - Goal state:

*	2	3
4	*	6
*	*	

  3. Learning from experience

Assignments

- Reading assignment:
  - Ch. 3
-