4.1

Functions: A function is a process or a relation that associates each element x of a set X, the domain of the function, to a single element y of another set Y (possibly the same set), the codomain of the function.

https://www.britannica.com/science/function-mathematics

Curves: In mathematics, a curve is an object similar to a line, but that does not have to be straight. It is the image of an interval to a topological space by a continuous function. One way to recognize a curve is that it bends and changes its direction at least once. It could be upward, downward, and so on.

Surfaces: A surface is the outermost or uppermost layer of a physical object or space. It is the portion or region of the object that can first be perceived by an observer using the senses of sight and touch, and is the portion with which other materials first interact.

Computer visualization: Computer visualization is the graphical representation of information and data through computer software. By converting data like constants, functions, matrices into visual elements like charts, graphs, and maps, computer visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data.

4.2

State-space representation.

(In control engineering, a state-space representation is a mathematical model of a physical system as a set of input, output and state variables related by first-order differential equations or difference equations. State variables are variables whose values evolve over time in a way that depends on the values they have at any given time and on the externally imposed values of input variables. Output variables' values depend on the values of the state variables.)

Single-state problem formulation

A **problem** is defined by four items:

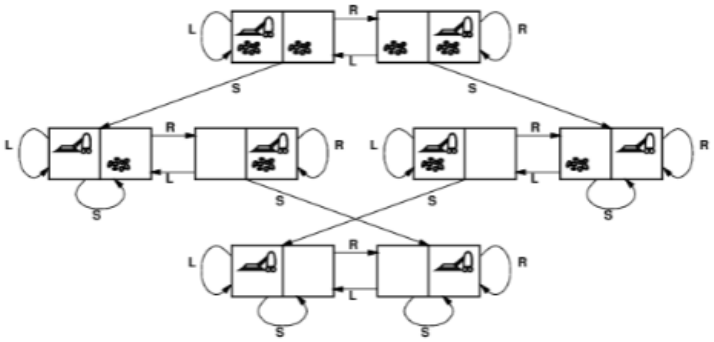
- **initial state** e.g. "at Arad"
- **successor function** S(x) = set of action–state pairs
  - e.g. S(Arad) = {<`Arad-Zerind, Zerind`>, ... }
- **goal test**, can be
  - *explicit*, e.g. x = "at Bucharest"
  - *implicit*, e.g. `NoDirt(x)`
- **path cost** (additive)
  - e.g. sum of distances, number of actions executed, etc.
  - c(x,a,y) is the **step cost**, assumed to be ≥ 0

A **solution** is a sequence of actions leading from the initial state to a goal state

Selecting a state space

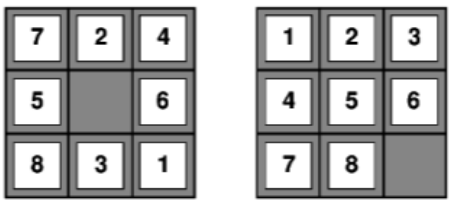
- Real world is absurdly complex
  - $\Rightarrow$ state space must be *abstracted* for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g. "Arad $\rightarrow$ Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - For guaranteed realizability, *any* real state "in Arad" must get to *some* real state "in Zerind"
- (Abstract) solution = set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem!

Example: vacuum world state space graph



- states
  - integer dirt and robot locations (ignore dirt etc.)
- actions
  - Left, Right, Suck, NoOp
- goal test
  - no dirt
- path cost
  - 1 per action (0 for NoOp)

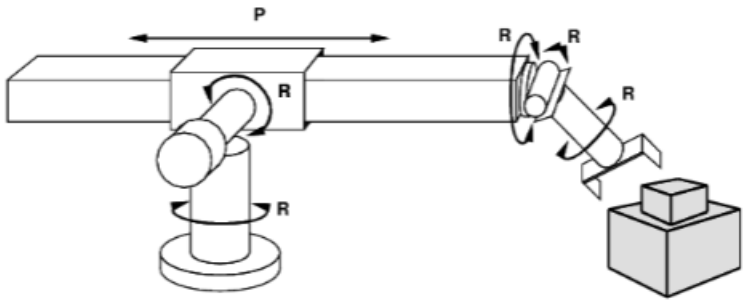
Example: The 8-puzzle



- states
  - integer locations of tiles (ignore intermediate positions)
- actions
  - move blank left, right, up, down (ignore unjamming etc.)
- goal test
  - $=$ goal state (given)
- path cost
  - 1 per move

Note: optimal solution of n-Puzzle family is NP-hard

Example: robotic assembly



- states
  - real-valued coordinates of robot joint angles
  - parts of the object to be assembled
- actions
  - continuous motions of robot joints
- goal test
  - complete assembly *with no robot included!*
- path cost
  - time to execute

Backtracking strategies:

CSP + Backtracking search
(On slide05: Page 3-18)

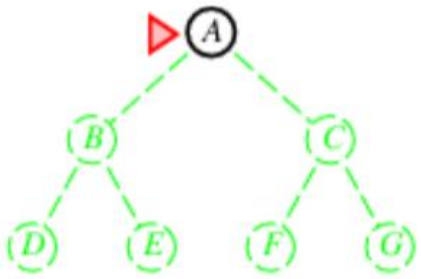
Graph-search procedures:

Graph search

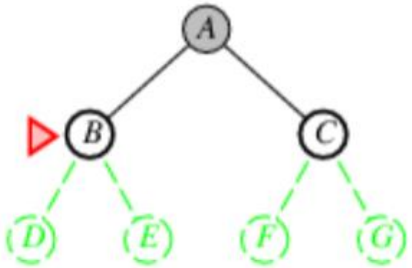
```
function Graph-Search(problem, fringe): a solution, or failur
  closed = an empty set
  fringe = Insert(Make-Node(Initial-State[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node = Remove-Front(fringe)
    if Goal-Test(problem, State[node]) then return node
    if State[node] is not in closed then
       add State[node] to closed
           fringe = InsertAll(Expand(node, problem), fringe)
  end
```

Breadth-First:

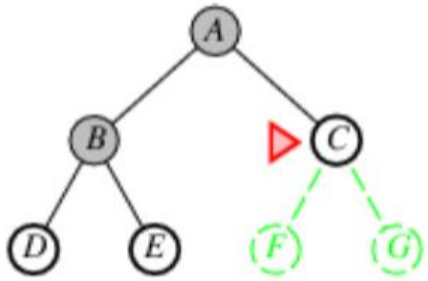
Expand **shallowest** unexpanded node
Implementation: *fringe* is a FIFO queue, i.e. new successors go at end



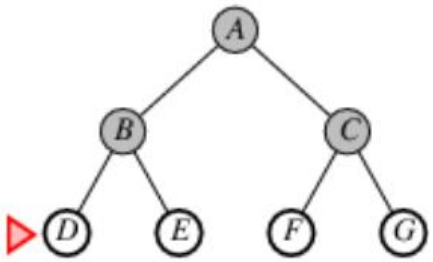
Expand **shallowest** unexpanded node
Implementation: *fringe* is a FIFO queue, i.e. new successors go at end



Expand **shallowest** unexpanded node
Implementation: *fringe* is a FIFO queue, i.e. new successors go at end



Expand **shallowest** unexpanded node
Implementation: *fringe* is a FIFO queue, i.e. new successors go at end

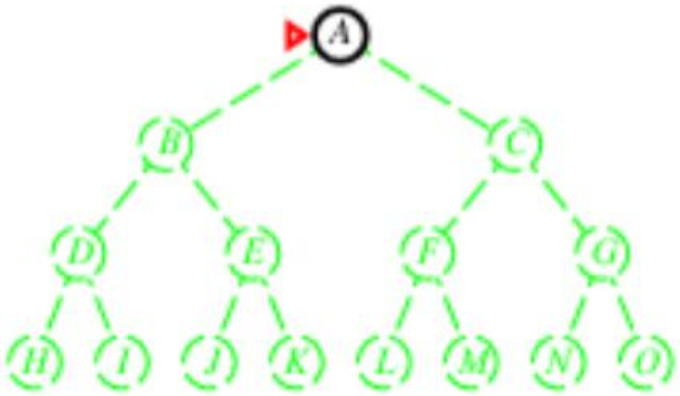


Properties of breadth-first search

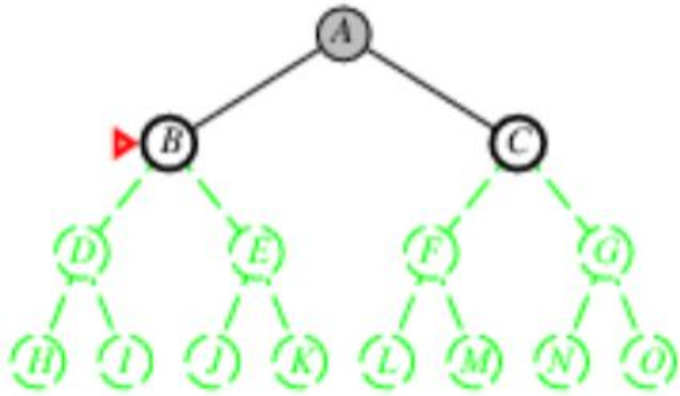
- Complete?
  - Yes (if b is finite)
- Time?
  - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e. exp. in d
- Space?
  - $O(b^{d+1})$ (keeps every node in memory)
- Optimal?
  - Yes (if cost $= 1$ per step); not optimal in general
- *Space* is the big problem; can easily generate nodes at 100MB/sec, so 24hrs $=$ 8640GB.

Depth-First

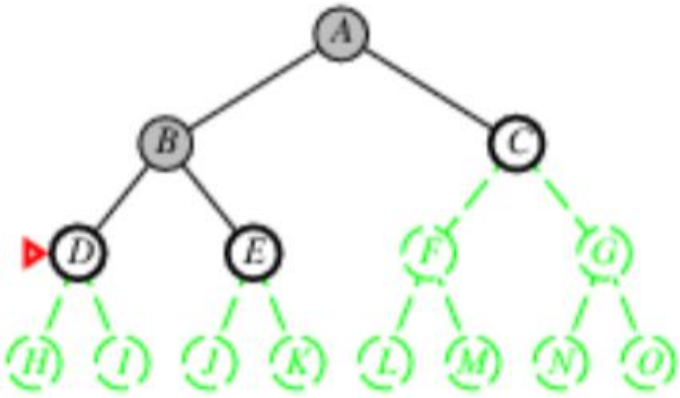
Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front



Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front

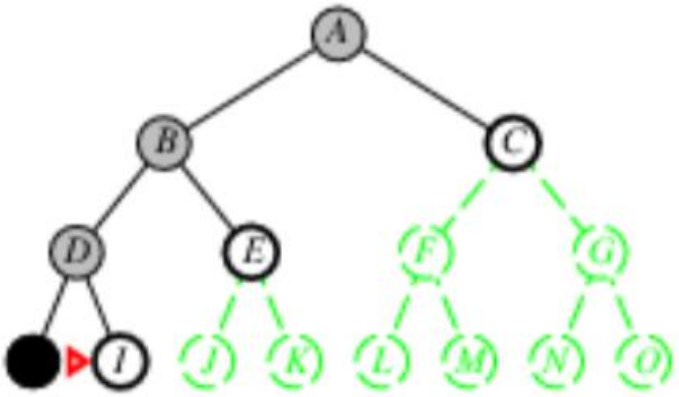


Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front

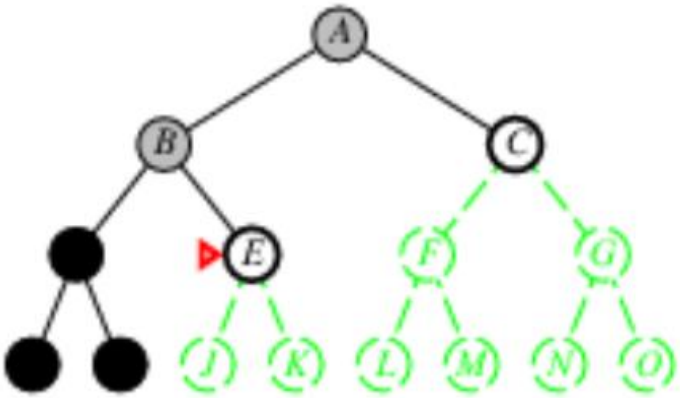


Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front

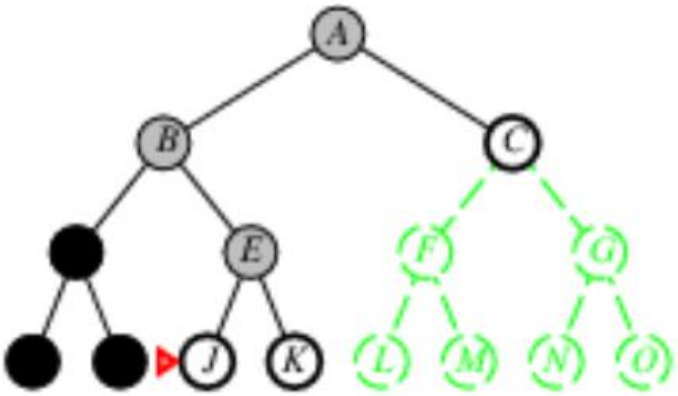
Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front



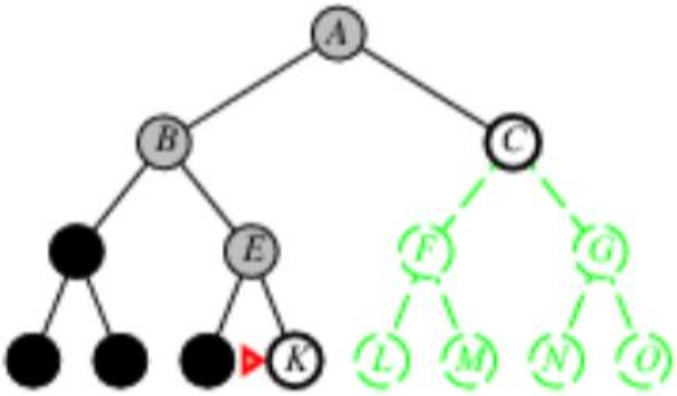
Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front



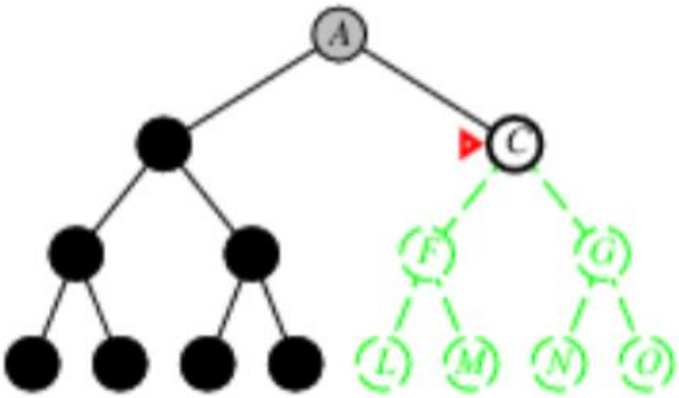
Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front



Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front



Expand **deepest** unexpanded node
Implementation: *fringe* = LIFO queue, i.e. put successors at front



Properties of depth-first search

- Complete?
  - No: fails in infinite-depth spaces, spaces with loops,
  - Modify to avoid repeated states along path,
  - $\Rightarrow$ 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

Summary of algorithms

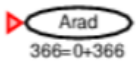
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

The A* algorithms.

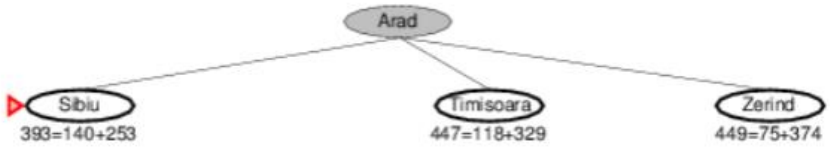
A^* search

- Idea:
  - avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated cost to goal from n
  - $f(n)$ = estimated total cost of path through n to goal
- A^* search uses an **admissible** heuristic
  - i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n.
  - Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.
  - E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance
- **Theorem**: A^* search is optimal

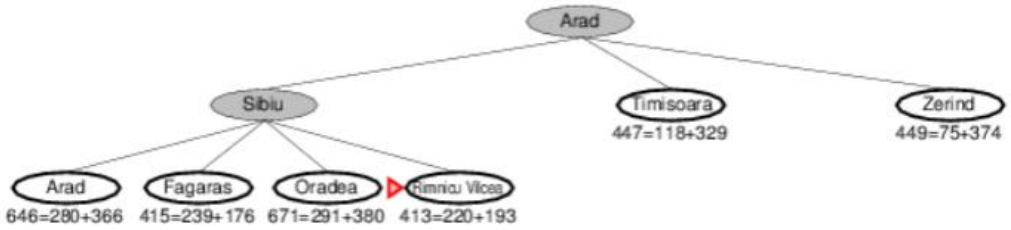
A^* search example



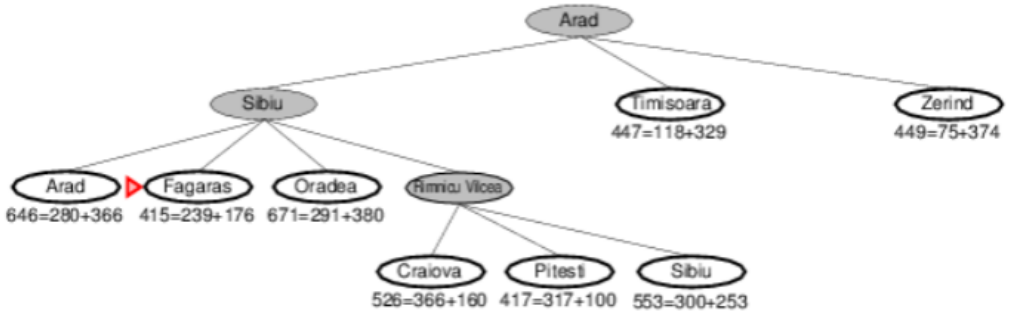
A^* search example



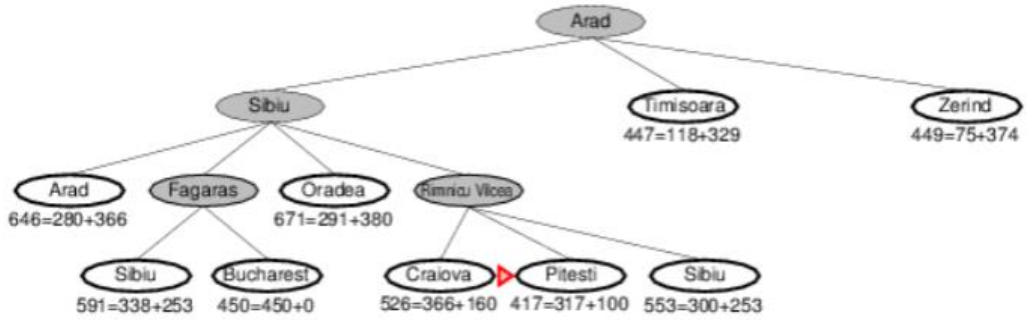
A^* search example



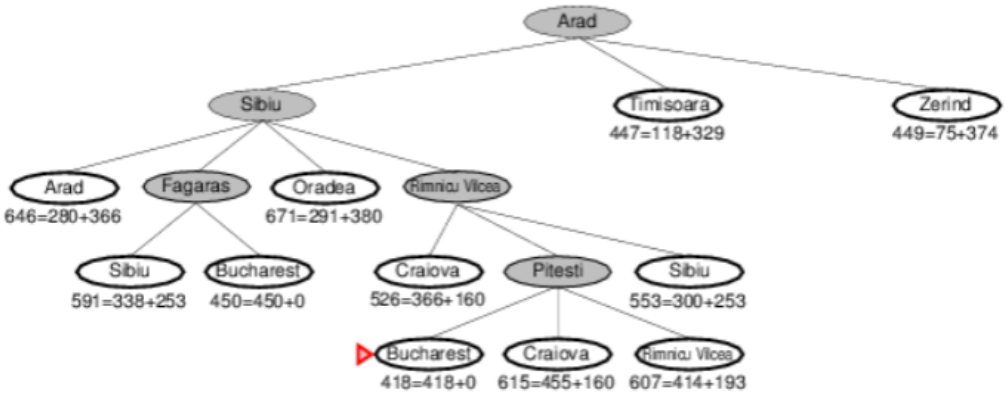
A^* search example



A^* search example

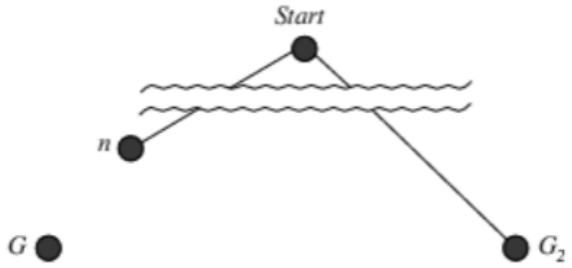


A^* search example



Optimality of A^* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G.

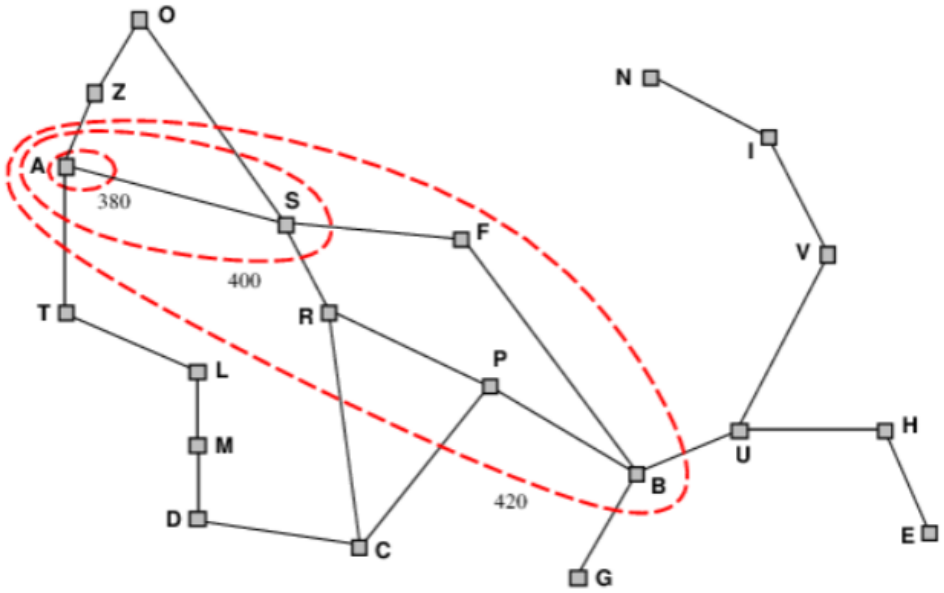


$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G) && \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

Since $f(G_2) > f(n)$, A^* will never select G_2 for expansion

Optimality of A^* (more useful)

Lemma: A^* expands nodes in order of increasing f value
Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A^*

- Complete
  - Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time
  - Exponential in [relative error in h $\times$ length of soln.]
- Space
  - Keeps all nodes in memory
- Optimal
  - Yes—cannot expand f_{i+1} until f_i is finished
    - A^* expands all nodes with $f(n) < C^*$
    - A^* expands some nodes with $f(n) = C^*$
    - A^* expands no nodes with $f(n) > C^*$

If C* is the cost of the optimal solution path, then we can say the following:

Computation time is not, however, A*'s main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A* usually runs

out of space long before it runs out of time. For this reason, A* is not practical for many large-scale problems.

Optimal if h(n) is consistent.

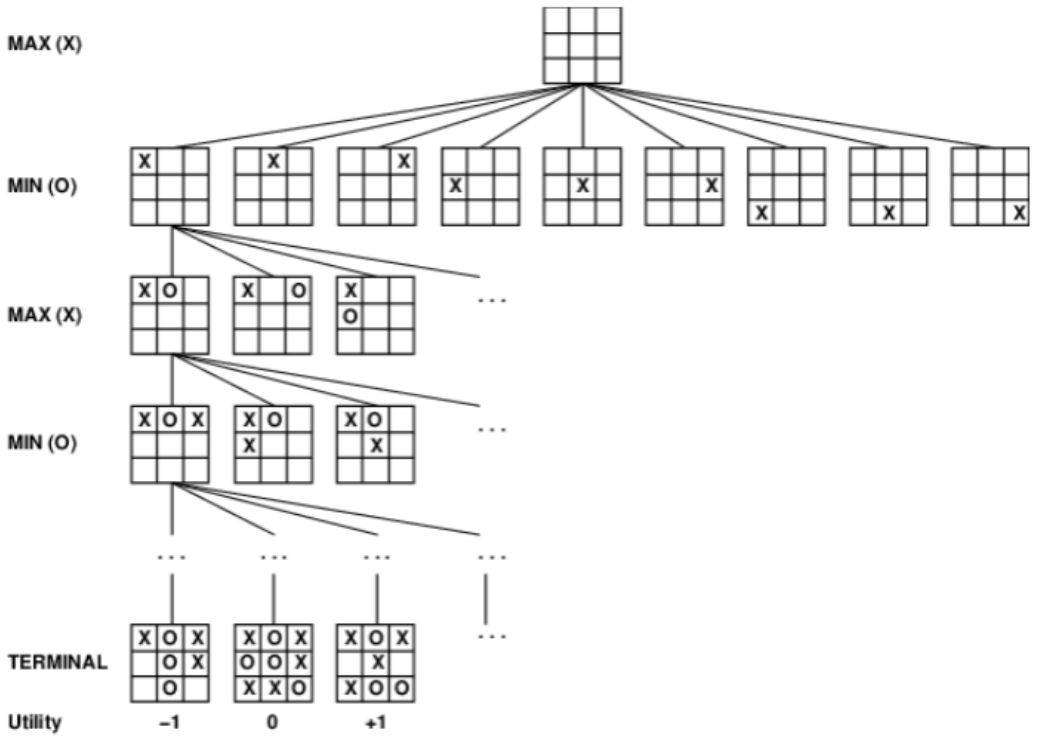
Two-player games, representation of the game.

（Recommend to review all slide6 in AI）

Types of games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Game tree (2-player, deterministic, turns)

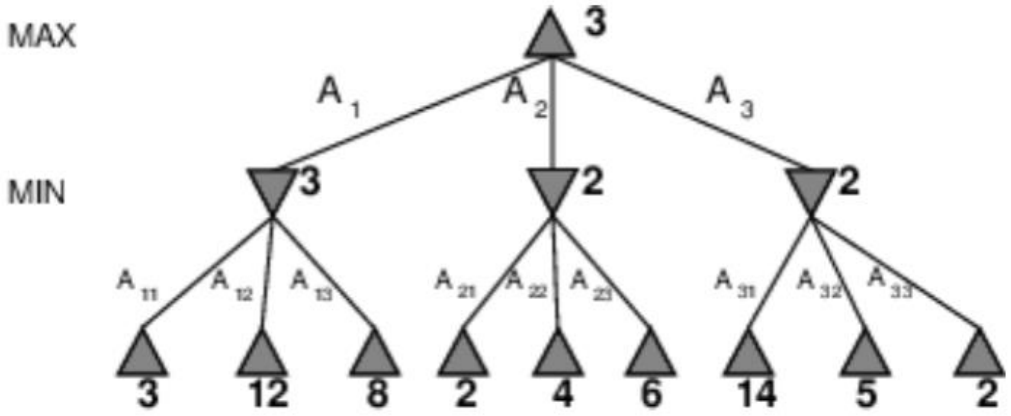


Winning strategy.

Min-max procedure.

Minimax

- Perfect play for deterministic, perfect-information games
- **Idea**: choose move to position with highest **minimax value**
  - = best achievable payoff against best play
- E.g., 2-ply game:



Minimax algorithm

```
function Minimax-Decision(state) returns an action
  state: current state in game

  return a in Actions(state) maximizing Min-Value(Result(a, state))

function Max-Value(state) returns a utility value

  if Terminal-Test(state) then return Utility(state)
  v := -infinity
  for (a, s) in Successors(state) do
    v := Max(v, Min-Value(s))
  return v

function Min-Value(state) returns a utility value

  if Terminal-Test(state) then return Utility(state)
  v := infinity
  for (a, s) in Successors(state) do
    v := Min(v, Max-Value(s))
  return v
```

Properties of minimax

- Complete
  - Yes, if tree is finite (chess has specific rules for this)
- Optimal
  - Yes, against an optimal opponent. Otherwise??
- Time complexity
  - $O(b^m)$
- Space complexity
  - $O(bm)$ (depth-first exploration)
- For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
  - $\Rightarrow$ exact solution completely infeasible
- But do we need to explore every path?