Practical AI

Part 1- Intelligent Search

Search

(Intelligent) Search

Constraint Satisfaction

Adversarial Search (Games)

Universität Zürich[UZH]

Dynamic and Distributed Information Systems

Note: Slides based on various slide sets for the Russel and Norvig AI book

Overview – Part 1

- **Problem Definition?**
- Search
  - Breadth first, Depth first, Iterative deepening
- Informed (or Intelligent) Search
  - Best First
  - A*, Hill climbing, Simulated annealing
- Constraint Satisfaction
  - Forward checking, Constraint propagation
- Adversarial Search (Games)

Planning

- Input
  - Description of set of all possible states of the world (in some knowledge representation language)
  - Description of initial state of world
  - Description of goal
  - Description of available actions
    - May include costs for performing actions
- Output
  - Sequence of actions that convert the initial state into one that satisfies the goal
  - May wish to minimize length or cost of plan

Example problem: Traveling in Romania

Situation:

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

Formulate Goal:

- be in Bucharest

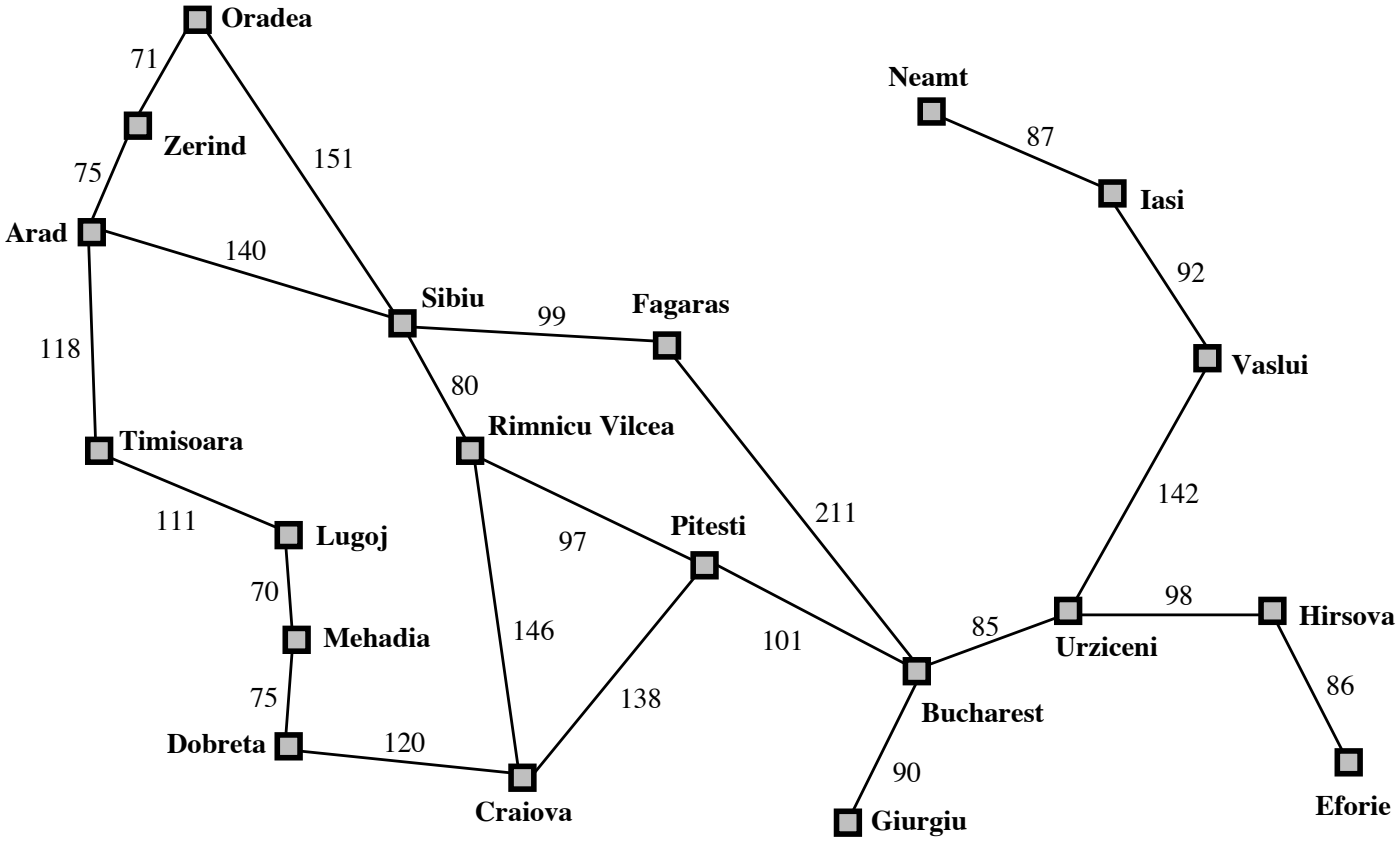
Formulate Problem:

- states: various cities
- actions: drive between cities

Find solution:

- sequence of cities
  e.g., Arad, Sibiu, Fagaras, Bucharest

Romania


Oradea
71
Neamt
87
Zerind
151
Iasi
75
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
97
Pitesti
70
98
Hirsova
Mehadia
146
85
101
Urziceni
75
138
86
Bucharest
Dobreta
120
90
Craiova
Giurgiu
Eforie

Problem Types

Deterministic, fully observable → single-state problem

- Agent knows exactly which state it will be in; solution is a sequence

Non-observable → conformant problem

- Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable → contingency problem

- percepts provide *new* information about current state
- solution is a *tree* or *policy*
- often *interleave* search, execution

Unknown state space → exploration problem (online)

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

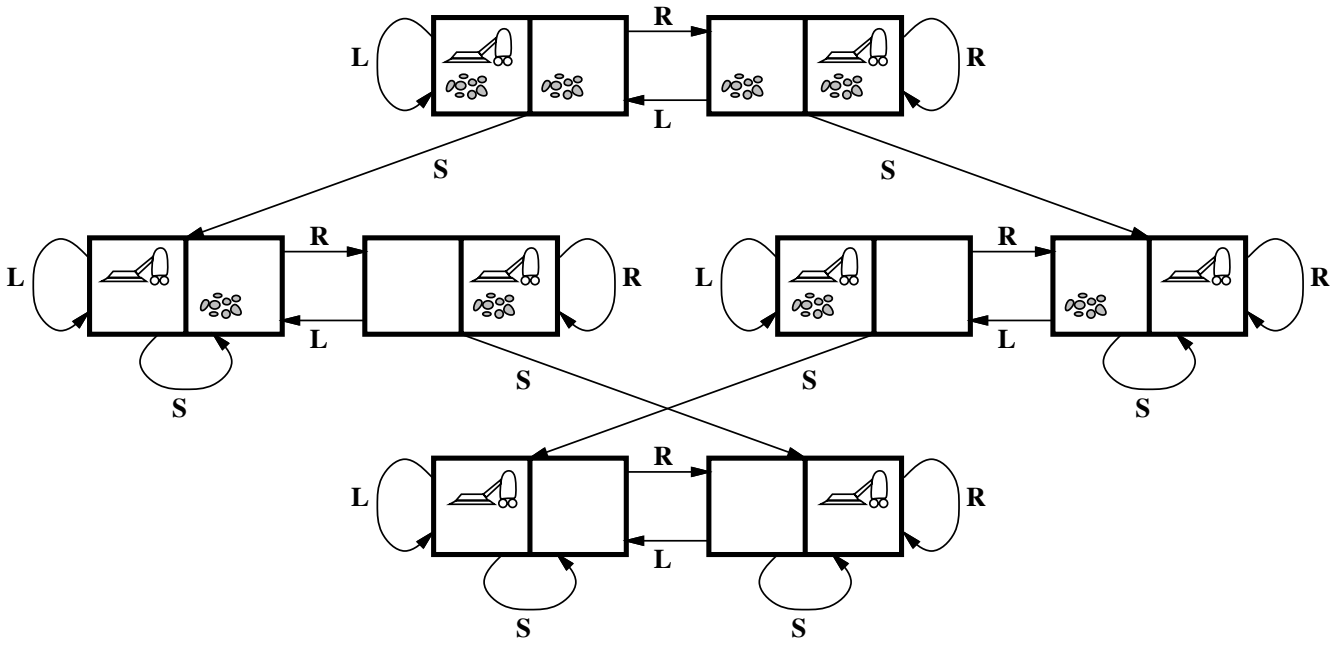
Defining the Problem
Example: Vacuum World

States?:

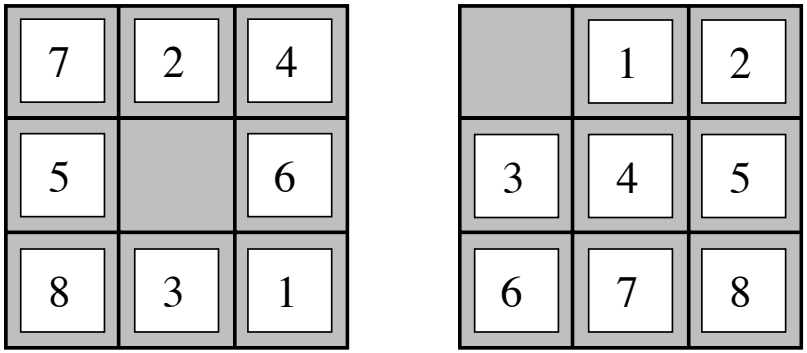
Actions?:

Goal test?:

Path cost?:



Defining the Problem
Example: The 8-Puzzle



Start State Goal State

States?:

Actions?:

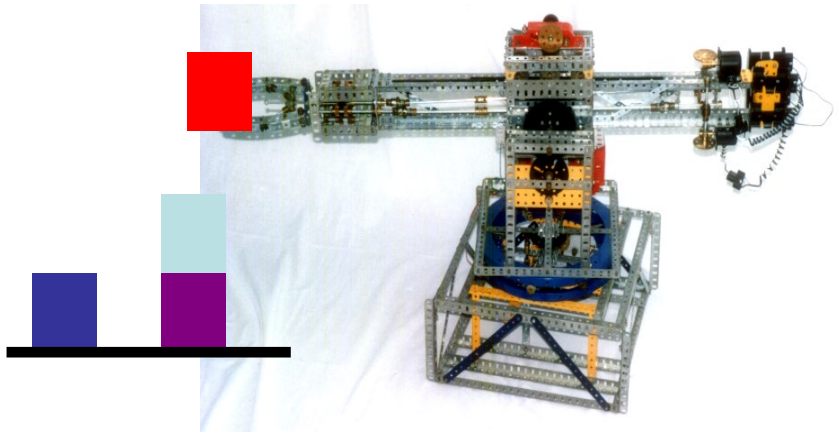
Goal test?:

Path cost?:

Defining the Problem

Example: Robot Control - Blocks World

- Input:
  - ***States?***:
  - ***Actions?***:
  - ***Goal test?***:
  - ***Path cost?***:
- Output:
  - ***Action Plan?***

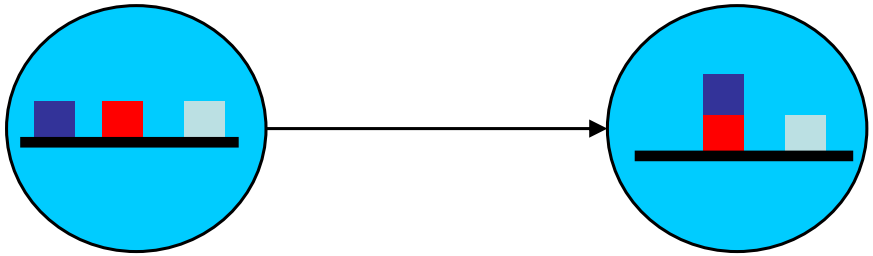


STRIPS Representation

- Description of initial state of world
  - Set of *propositions* that completely describes a world
  - { (block a) (block b) (block c) (on-table a)
    (on-table b) (clear a) (clear b) (clear c)
    (arm-empty) }
- Description of goal (i.e. set of desired worlds)
  - Set of propositions that *partially* describes a world
  - { (on a b) (on b c) }
- Description of available actions

STRIPS Representation
How Represent Actions?

- World = set of propositions true in that world
- Actions:
  - Precondition: conjunction of propositions
  - Effects: propositions made true & propositions made false (deleted from the state description)



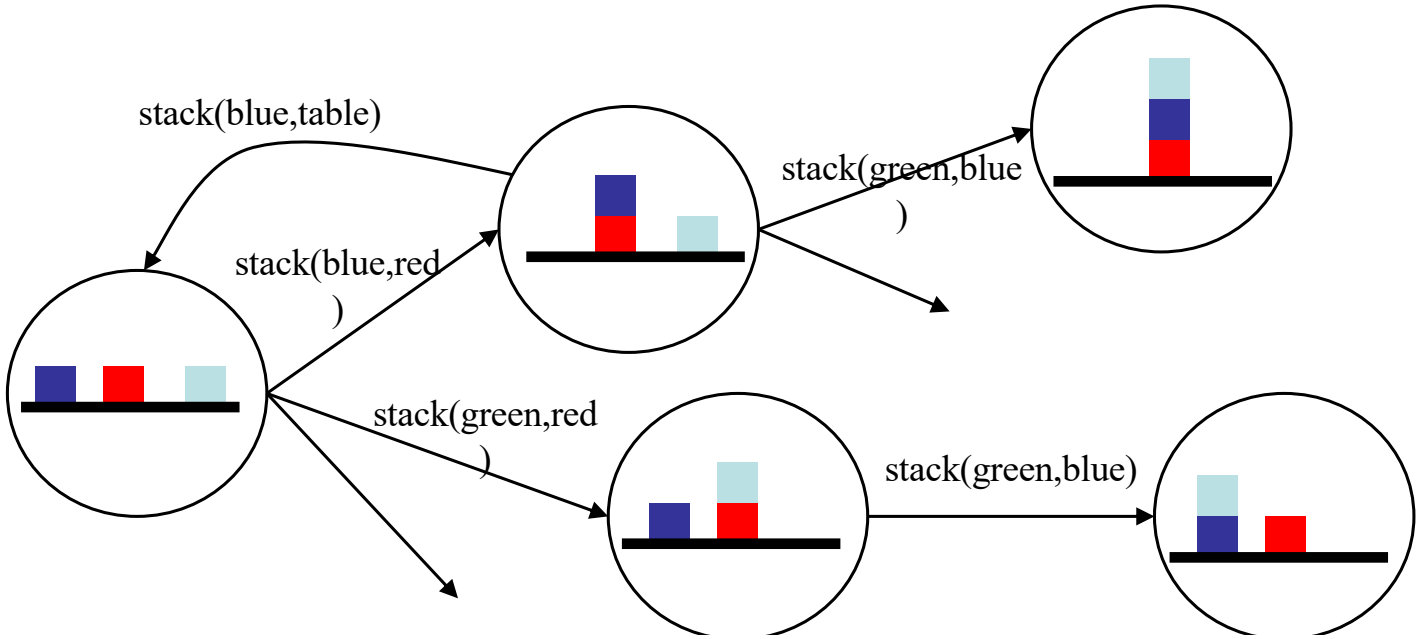
operator: **stack_B_on_R**

precondition: **(on B Table) (clear R)**

effect: **(on B R) (:not (clear R))**

Planning as graph search

- Planning can be viewed as finding paths in a graph, where the graph is implicitly specified by the set of actions
- Blocks world:
  - vertex = relative positions of all blocks
  - edge = robot arm stacks one block



Overview – Part 1

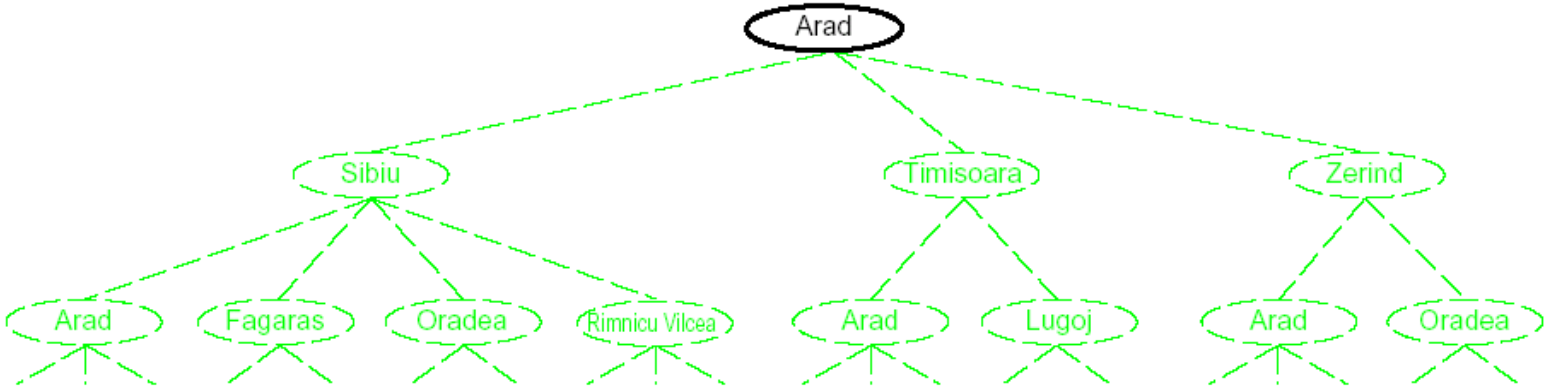
- Problem Definition?
- **Search**
  - Breadth first, Depth first, Iterative deepening
- Informed (or Intelligent) Search
  - Best First
  - A*, Hill climbing, Simulated annealing
- Constraint Satisfaction
  - Forward checking, Constraint propagation
- Adversarial Search (Games)

Searching: Tree Search

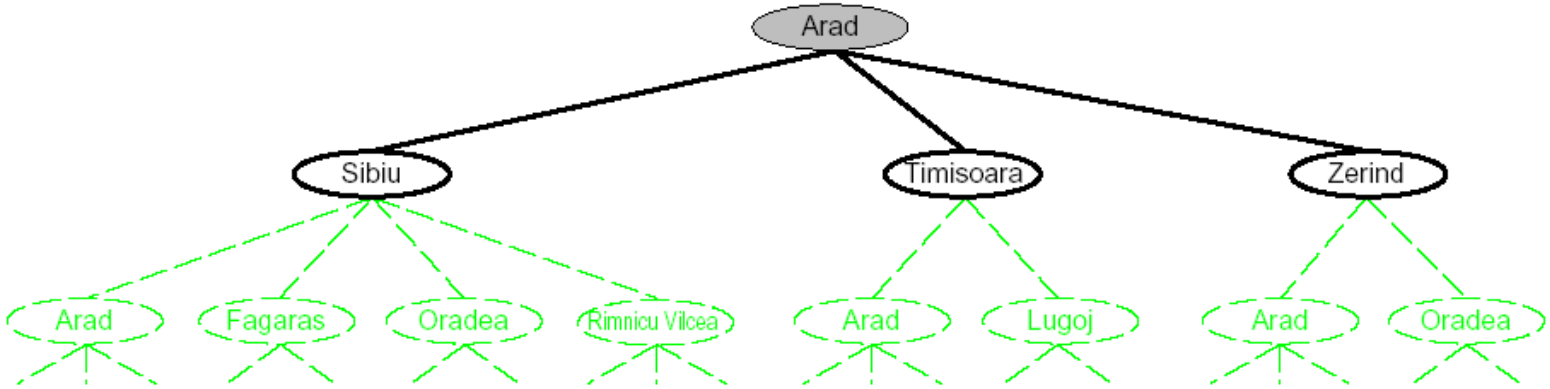
- Basic idea: offline, simulated exploration of state space by generating successors of already–explored states
  (a.k.a. expanding states)

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf 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 tree
    end
```

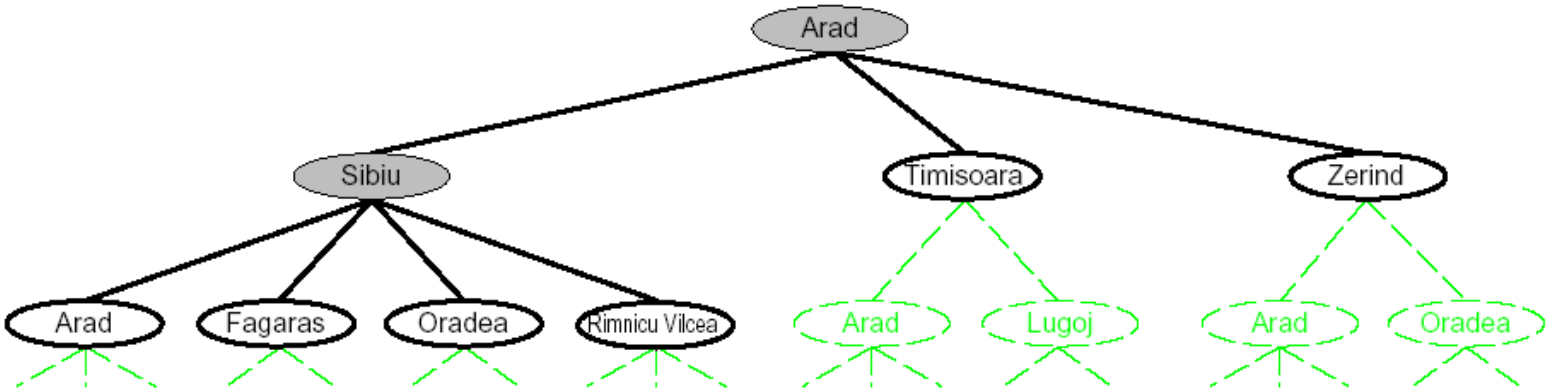
Searching: *Tree Search*



Searching: *Tree Search*



Searching: *Tree Search*

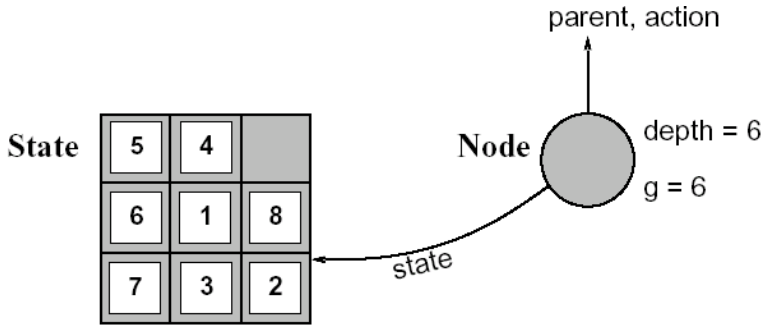


Implementation: States as Nodes

A *state* is a (representation of) a physical configuration

A *node* is a data structure constituting part of a search tree includes *parent, children, depth, path cost* **`g(x)`**

States do not have parents, children, depth, or path cost!



The **`Expand`** function creates new nodes, filling in the various fields and using the **`SuccessorFn`** of the problem to create the corresponding states.

Implementation: General Tree Search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    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] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Search Strategies

- A strategy is defined by picking the *order of node expansion*

- Strategies are evaluated along the following dimensions:
  - ***Completeness*** - does it always find a solution if one exists?
  - ***Complexity*** -number of nodes generated/expanded
  - ***Space complexity*** -maximum number of nodes in memory
  - ***Optimality*** - does it always find a least-cost solution?

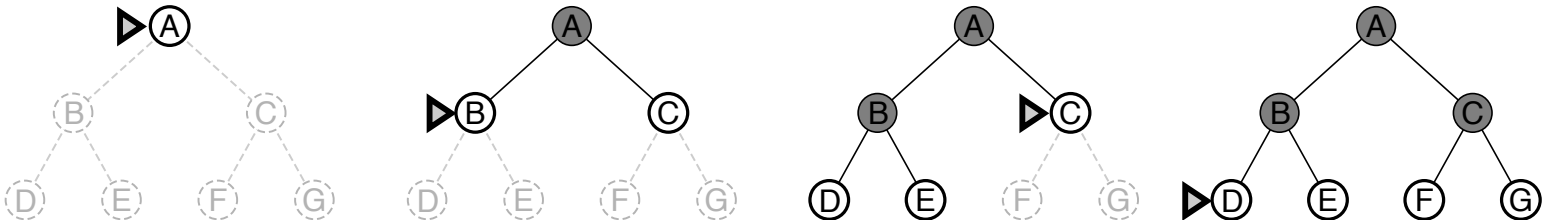
- Time and space complexity are measured in terms of
  - **b**: maximum branching factor of the search tree
  - **d**: depth of the least-cost solution
  - **m**: maximum depth of the state space (may be ∞)

Uninformed Search Strategies

- *Uninformed* strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Breadth-first search

- Expand the *shallowest* expanded Node
- Implementation:
  - **`fringe`** is a FIFO Queue, i.e., new successors go to the end



Breadth-first search: Properties

Complete?

- Yes (if **b** is finite)

Time?

- 1+b+b^2+b^3+ … +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 10MB/secs, 24hrs = 860GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

- `fringe` = queue ordered by path cost

Equivalent to breadth-first if step costs all equal

Complete?

- Yes, if step cost ≥ ε

Time?

- # of nodes with **g** ≤ cost of optimal solution, $O(b^{\lceil C^*/\varepsilon \rceil})$ where C* is the cost of the optimal solution

Space?

- # of nodes with **g** ≤ cost of optimal solution, $O(b^{\lceil C^*/\varepsilon \rceil})$

Optimal?

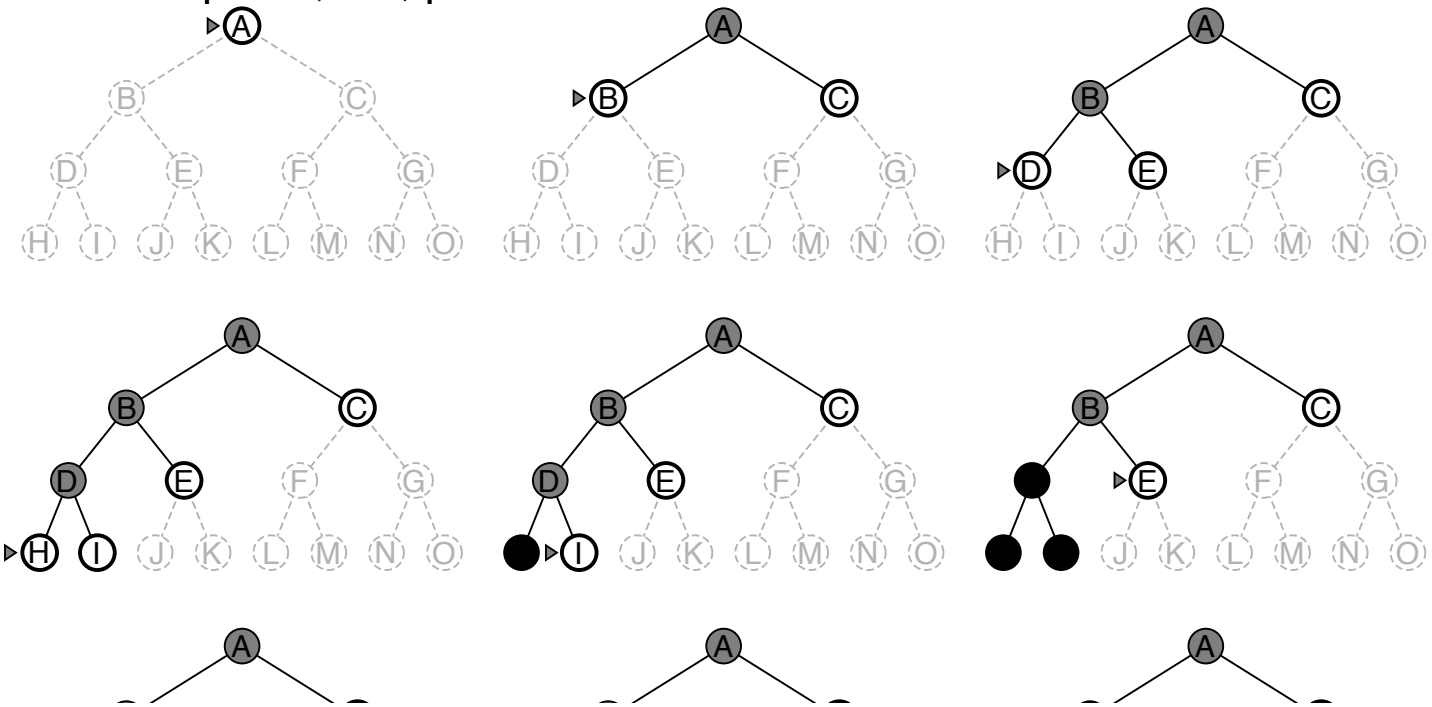
- Yes, nodes expanded in increasing order of g(n)

Depth-first search

Expand deepest unexpanded node

Implementation:

`fringe` = LIFO queue, i.e., put successors at front

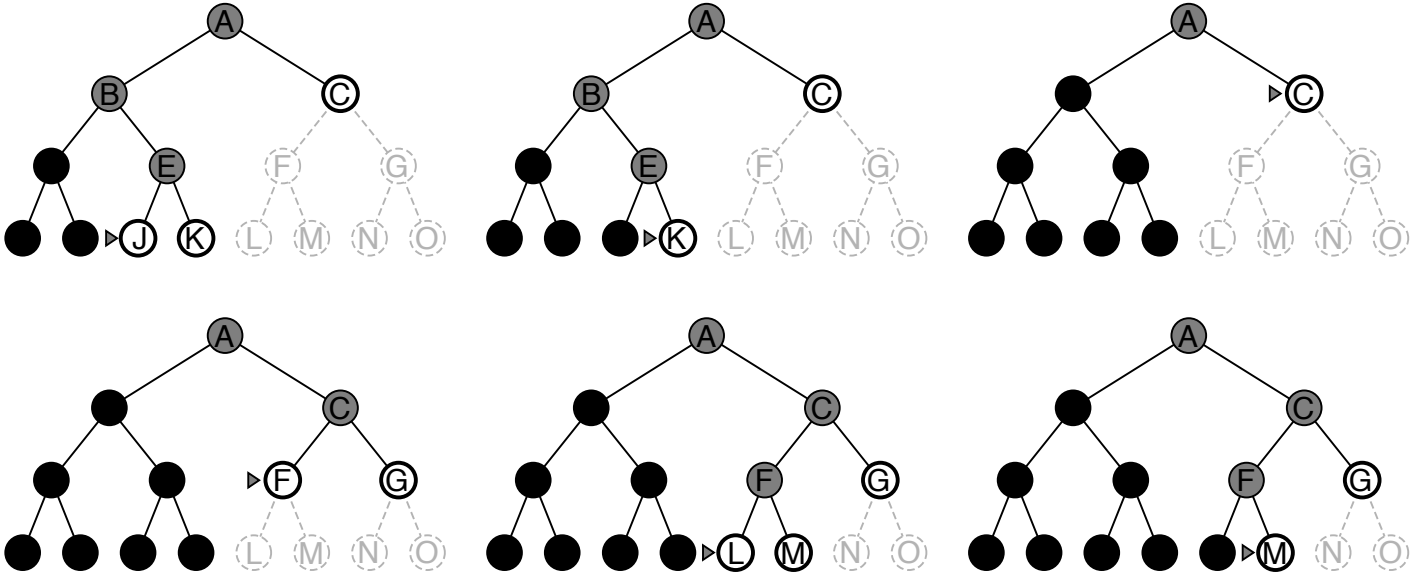


Depth-first search (cont.)

Expand deepest unexpanded node

Implementation:

`fringe` = LIFO queue, i.e., put successors at front



Depth-first search: Properties

Complete?

- No: fails in infinite-depth spaces, spaces with loops
- Modify to avoid repeated states along path
- → 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

Depth-limited search

= depth-first search with depth limit **l**,

i.e., nodes at depth **l** have no successors

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
            result ← RECURSIVE-DLS(successor, problem, limit)
            if result = cutoff then cutoff-occurred? ← true
            else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

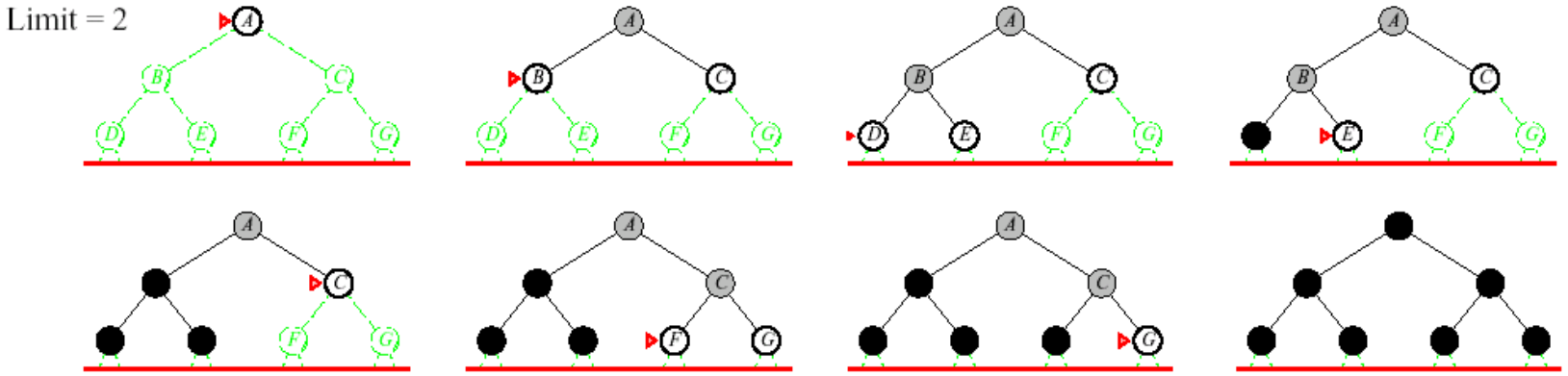
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Iterative deepening search *l* = *0*

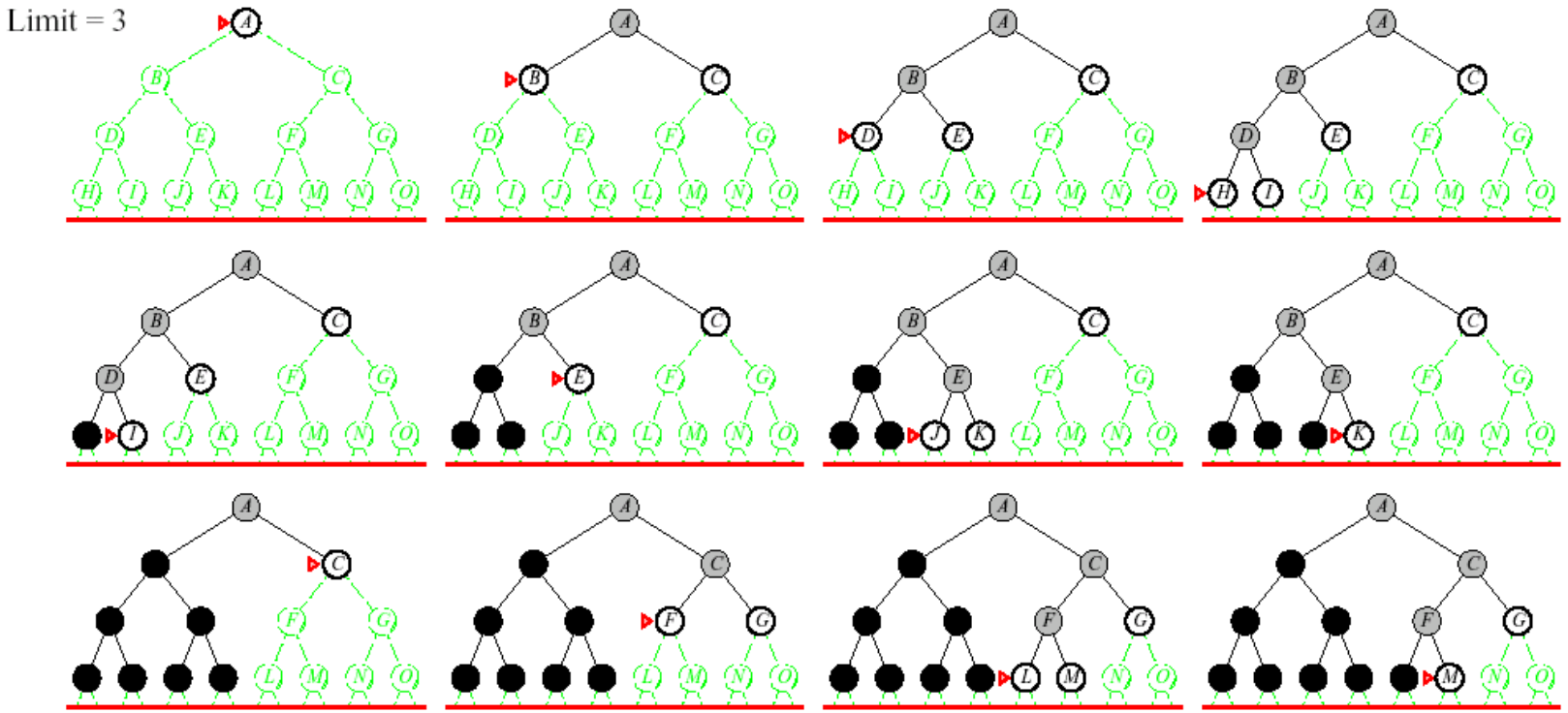
Limit = 0

Iterative deepening search *l* = *1*

Iterative deepening search *l* = 2



Iterative deepening search *l* = *3*



Iterative deepening search: Properties

Complete?

- Yes

Time?

- (d+1)b^0 + d b^1 + (d-1)b^2 + … + b^d = O(b^d)

Space?

- O(bd)

Optimal?

- Yes, if step cost = 1
  Can be modified to explore uniform-cost tree

Numerical comparison for b=10 and d=5, solution at far right:

N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

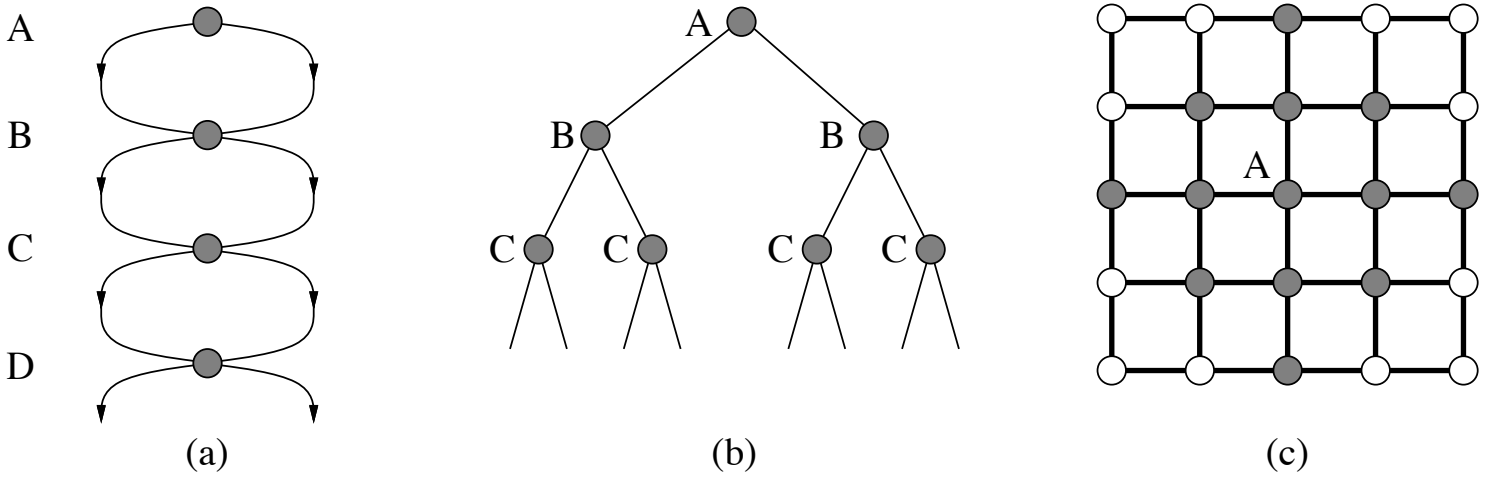
N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100

Comparison 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*

Repeated States

- Failure to detect repeated states can turn a linear problem into an exponential one!



Graph Search

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

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

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Overview – Part 1

- Problem Definition?
- Search
  - Breadth first, Depth first, Iterative deepening
- **Informed (or Intelligent) Search**
  - Best First
  - A*, Hill climbing, Simulated annealing
- Constraint Satisfaction
  - Forward checking, Constraint propagation
- Adversarial Search (Games)

Informed Search Strategies

- *Informed* strategies use some information about the problem (a heuristic) to improve performance
  - Best-first search
  - A* search
  - Heuristics
  - Hill-climbing
  - Simulated annealing

Review: Tree Search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    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] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

- A strategy is defined by picking the order of node expansion

➔ How can we choose a “smart order”?

Best First Search

Idea:

- use an ***evaluation*** function for each node as an estimate of ***"desirability"***

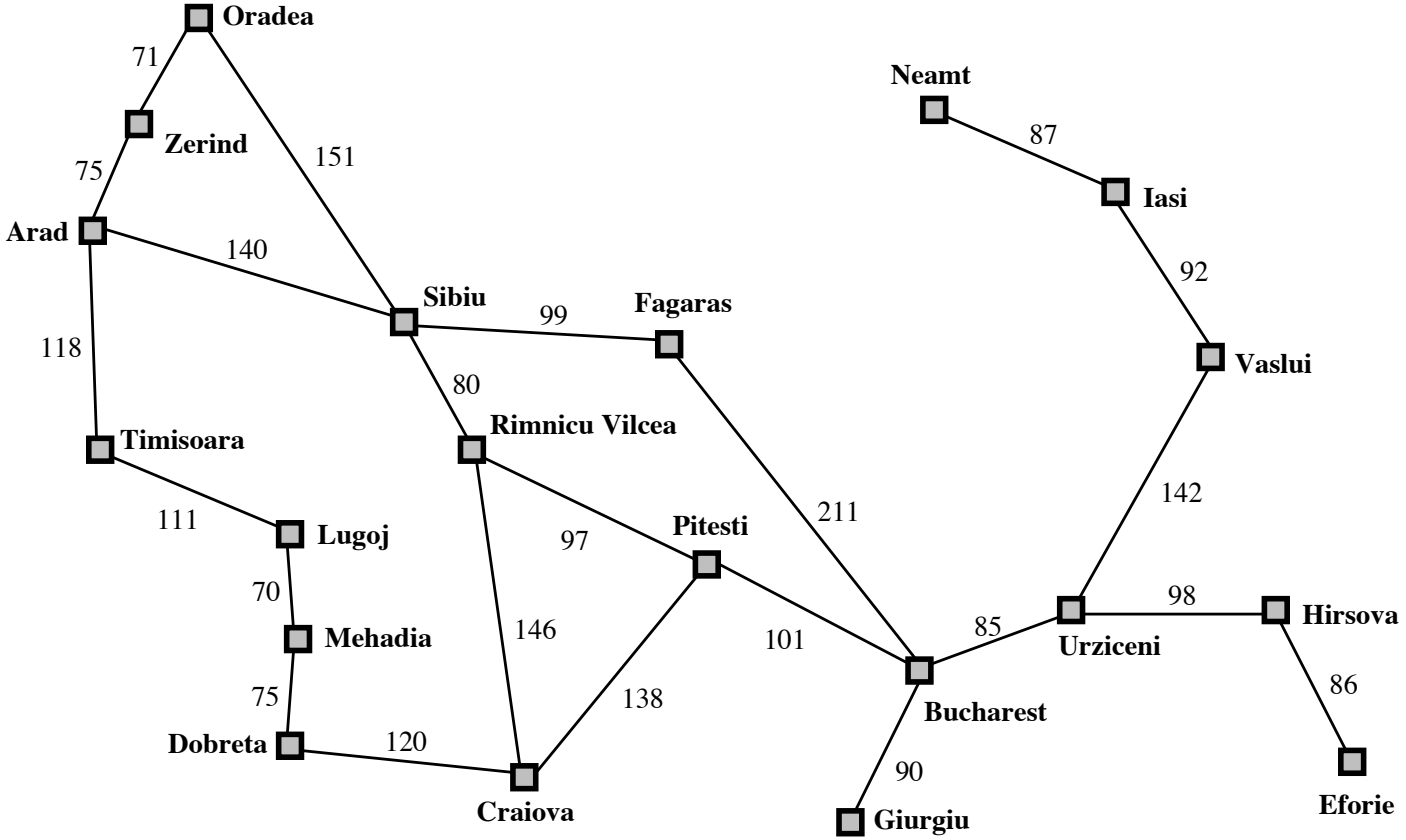
➔Expand most desirable unexpanded node

Implementation:

- **`fringe`** is a queue sorted in decreasing order of desirability

- Special cases:
  - greedy search
  - A* search

Romania with Step Cost in km



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

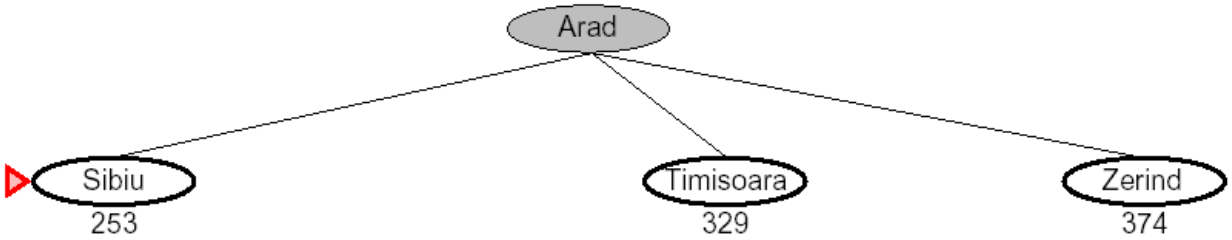
- Evaluation function $h(n)$ (heuristic) = estimate of cost from n to the closest goal

- E.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

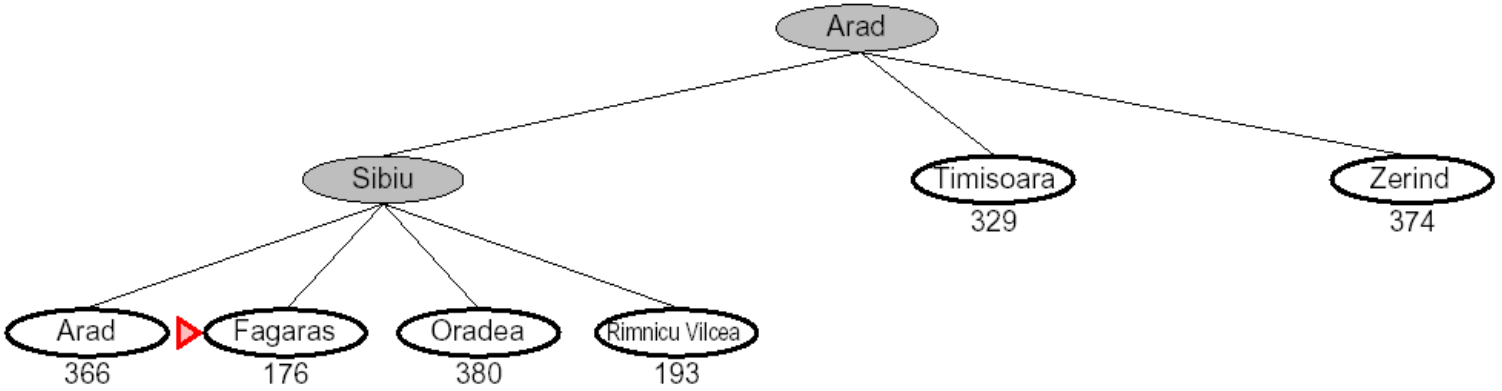
- Greedy search expands the node that *appears* to be closest to goal

Greedy Search

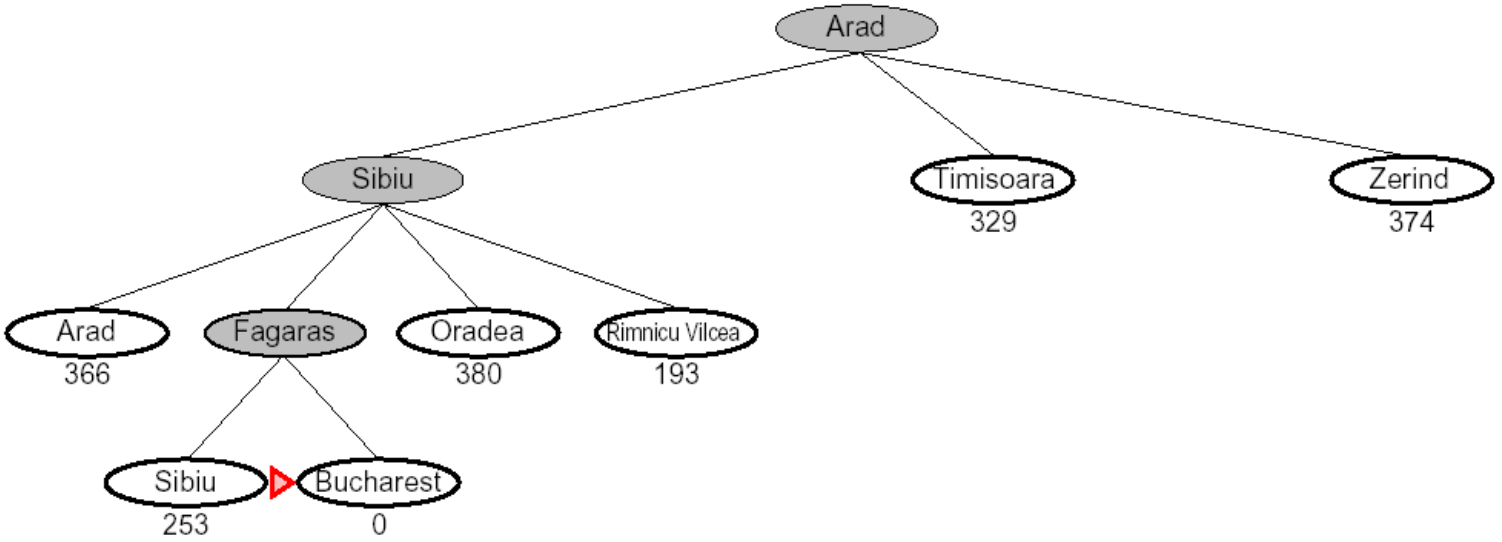
Greedy Search



Greedy Search



Greedy Search



Greedy Search: Properties

Complete?

- No – can get stuck in loops,
  e.g., Iasi → Neamt → Iasi → Neamt →
- Complete in finite space with repeated-state checking

Time?

- *O(b^m)*, but a good heuristic can give dramatic improvement

Space?

- O(b^m), keeps all nodes in memory

Optimal

- No

A* Search

Idea: avoid expanding paths that are already expensive

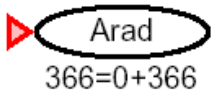
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost to goal from n
- $f(n)$ = estimated total cost of path through n to goal

- 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)

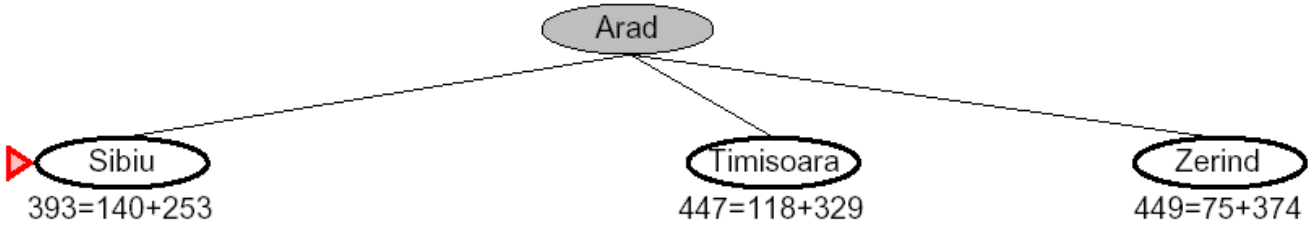
- E.g., $h_{SLD}(n)$ never overestimates the actual road distance

Theorem: A* search is optimal

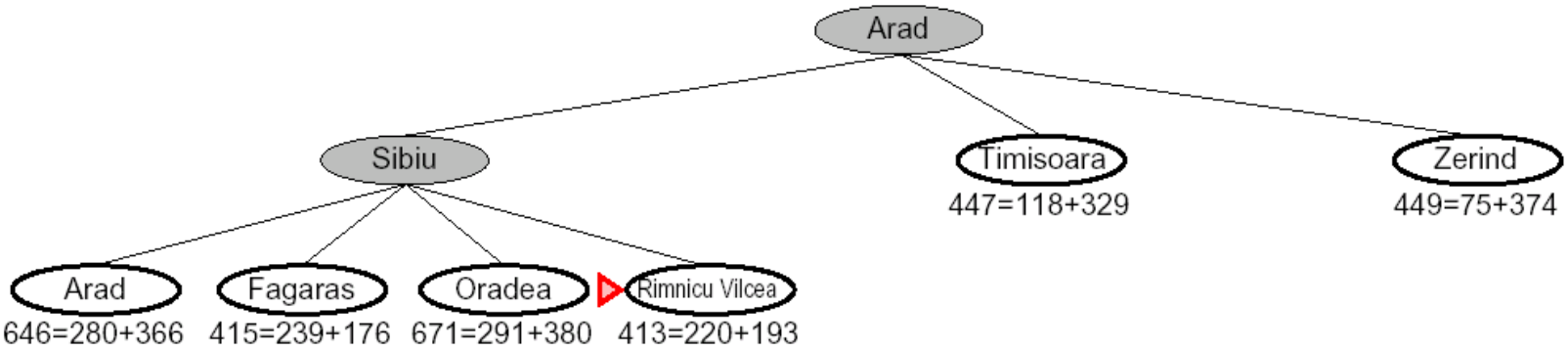
A* Search



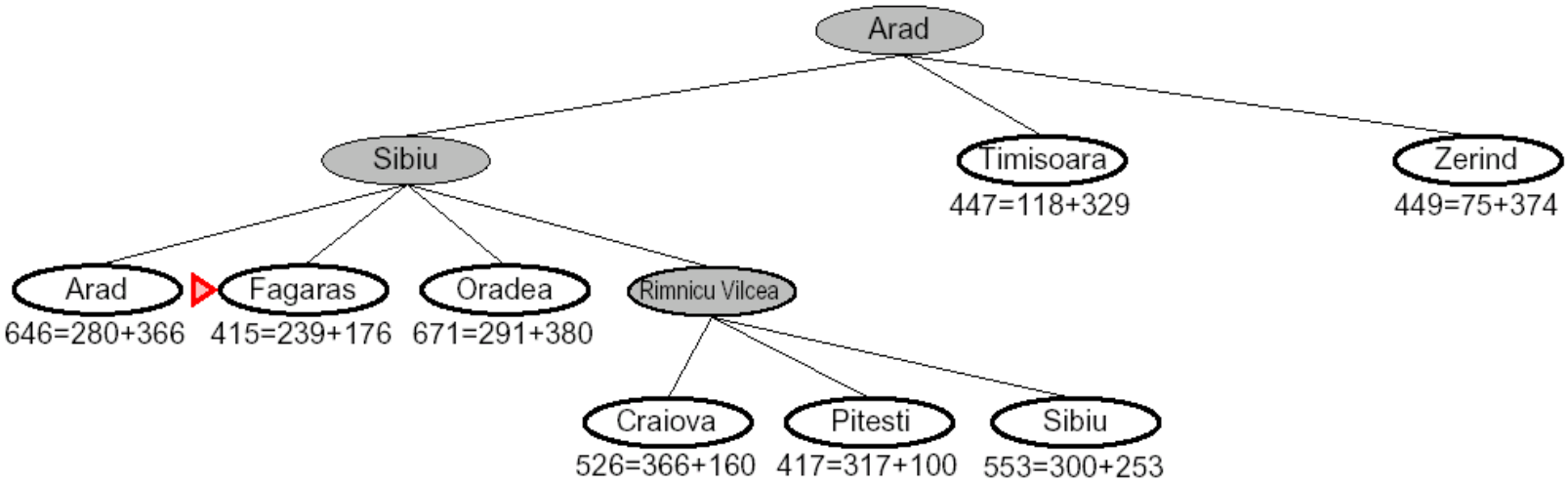
A* Search


Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

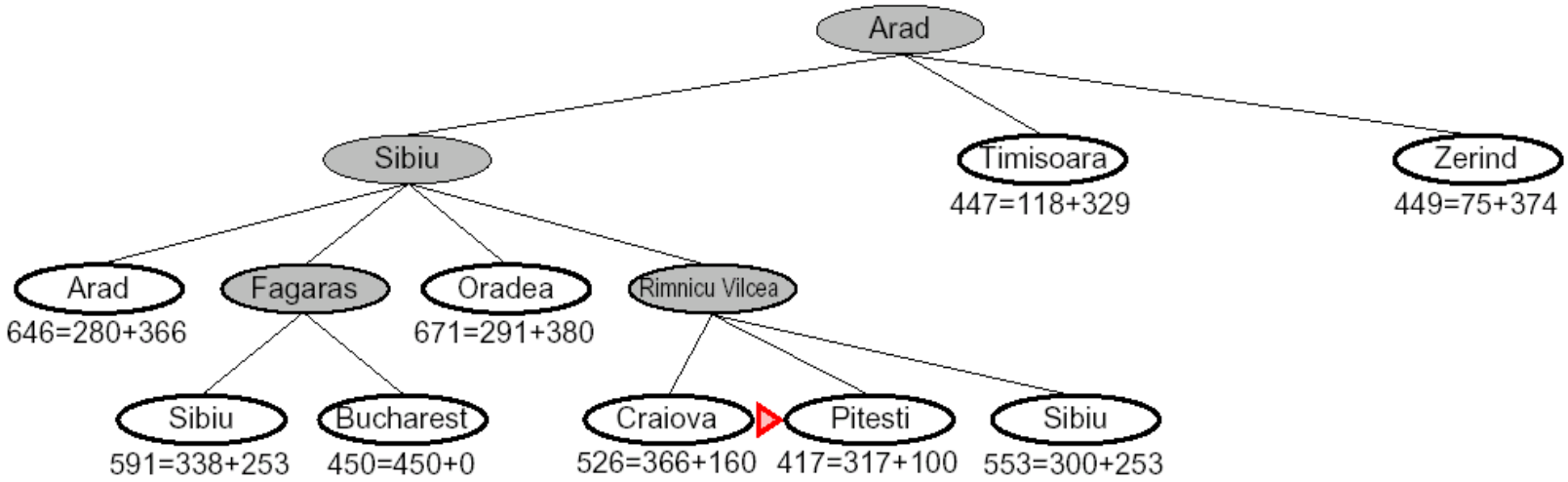
A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

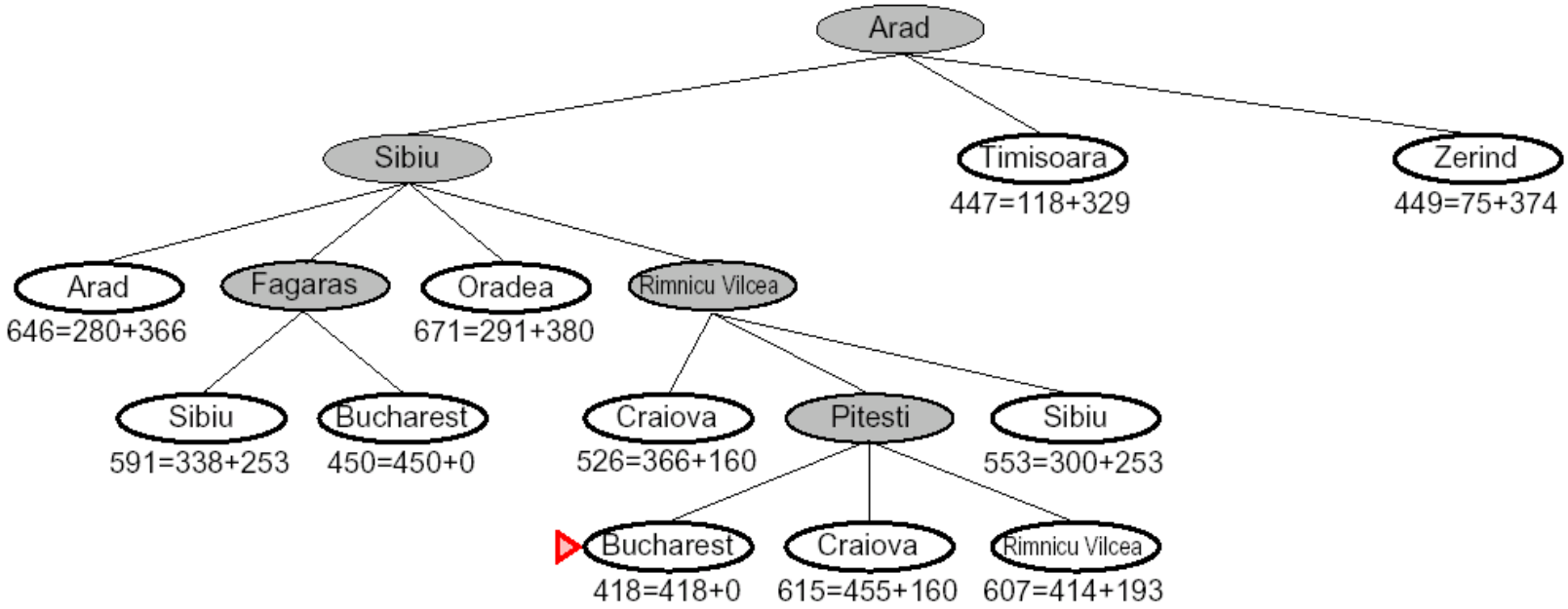
A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

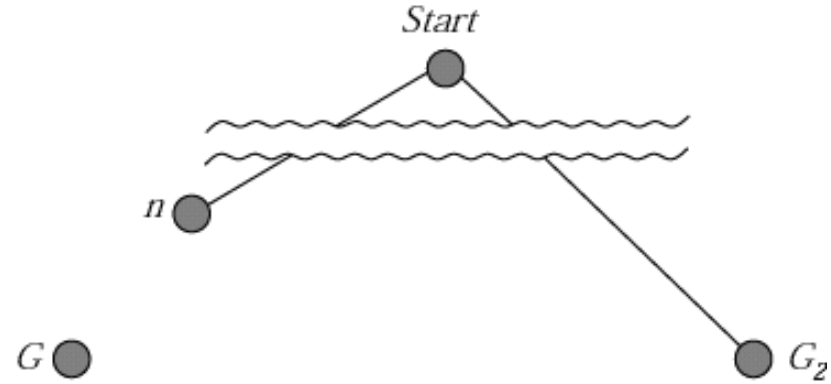
Optimality of A* (standard proof)

- 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_1.

$f(G_2) = g(G_2)$ since $h(G_2) = 0$

$> g(G_1)$ since G_2 is suboptimal

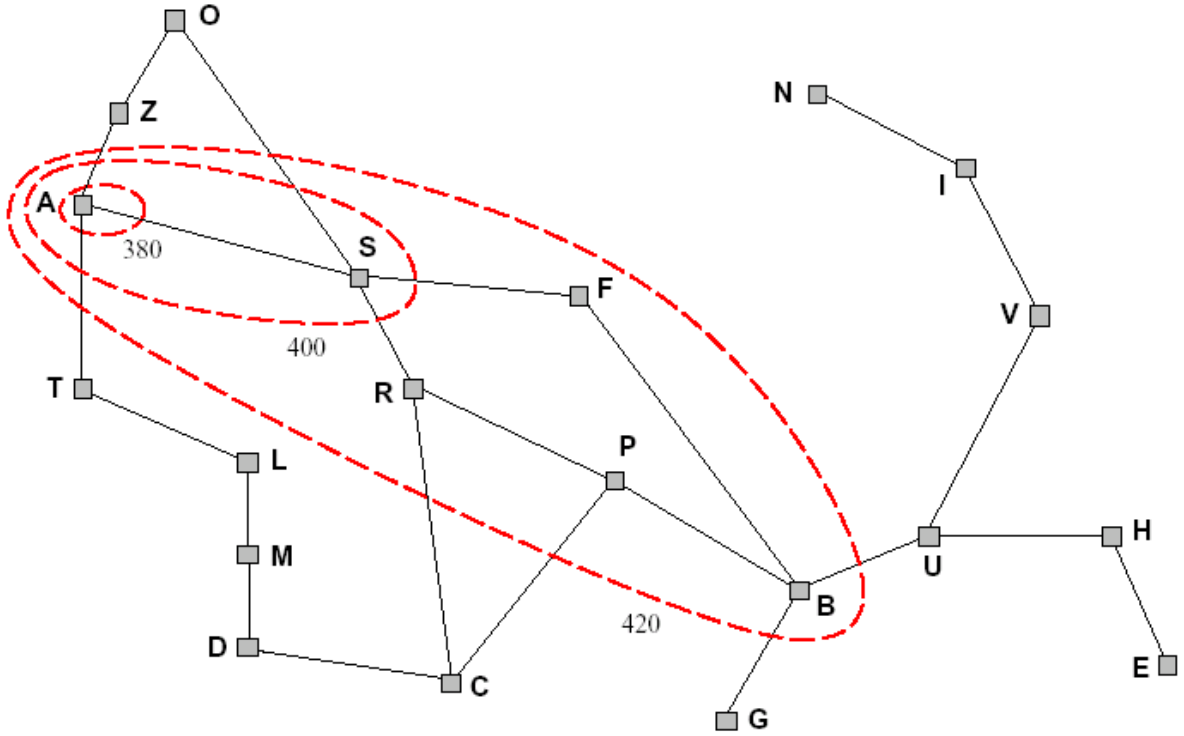
$\geq f(n)$ since h is admissible



- 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 $f_i < f_{i+1}$



A* search: Properties

Complete?

- Yes,
  unless there are infinitely many nodes with $f \leq f(G)$

Time?

- Exponential in [relative error in h * length of soln.]

Space?

- Keeps all nodes in memory

Optimal?

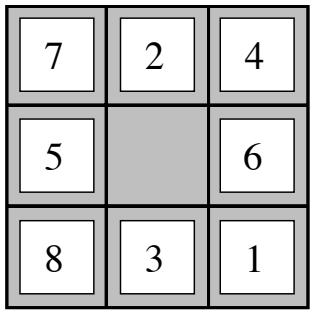
- Yes, cannot expand f_{i+1} until f_i is finished

- A* expands all nodes with $f(n) < C^*$
- A* expands some nodes with $f(n) = C^*$
- A* expands no nodes with $f(n) > C^*$

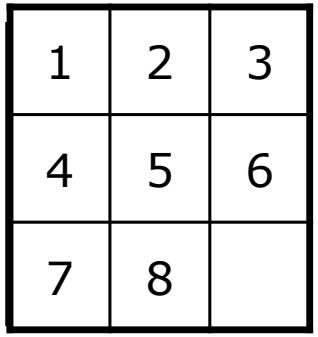
Admissible Heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total *Manhattan* distance (i.e., no. of squares from desired location of each tile)



Start State



Goal State

- $h_1(S)$ = 7
- $h_2(S)$ = 4+0+3+3+1+0+2+1 = 14

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible) then $h_2(n)$ ***dominates*** $h_1(n)$ and is better for search

Typical search costs:

d=14	IDS	= 3,473,941 nodes
	A*(h_1)	= 539 nodes
	A*(h_2)	= 113 nodes
d=24	IDS	approx 54,000,000,000 nodes
	A*(h_1)	= 39,135 nodes
	A*(h_2)	= 1,641 nodes

Relaxed Problems

- Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem
- If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to *any adjacent square*, then $h_2(n)$ gives the shortest solution
- Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

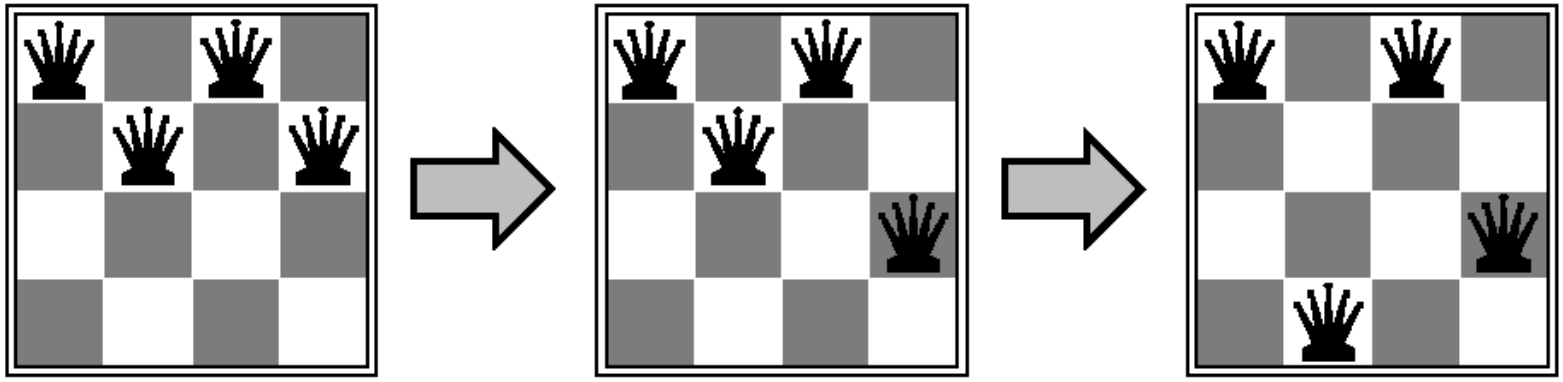
Iterative Improvement Algorithms

- In many optimization problems, *path* is irrelevant; the goal state itself is the solution
- Then state space = set of "complete"' configurations;
  - find *optimal* configuration, e.g., TSP
  - or, find configuration satisfying constraints, e.g., timetable
- In such cases, can use *iterative improvement* algorithms;
- keep a single "current"' state, try to improve it
- Constant space, suitable for online as well as offline search

Example: n-queens

- Put *n* queens on an *n* * *n* board with no two queens on the same row, column, or diagonal

- Move a queen to reduce number of conflicts



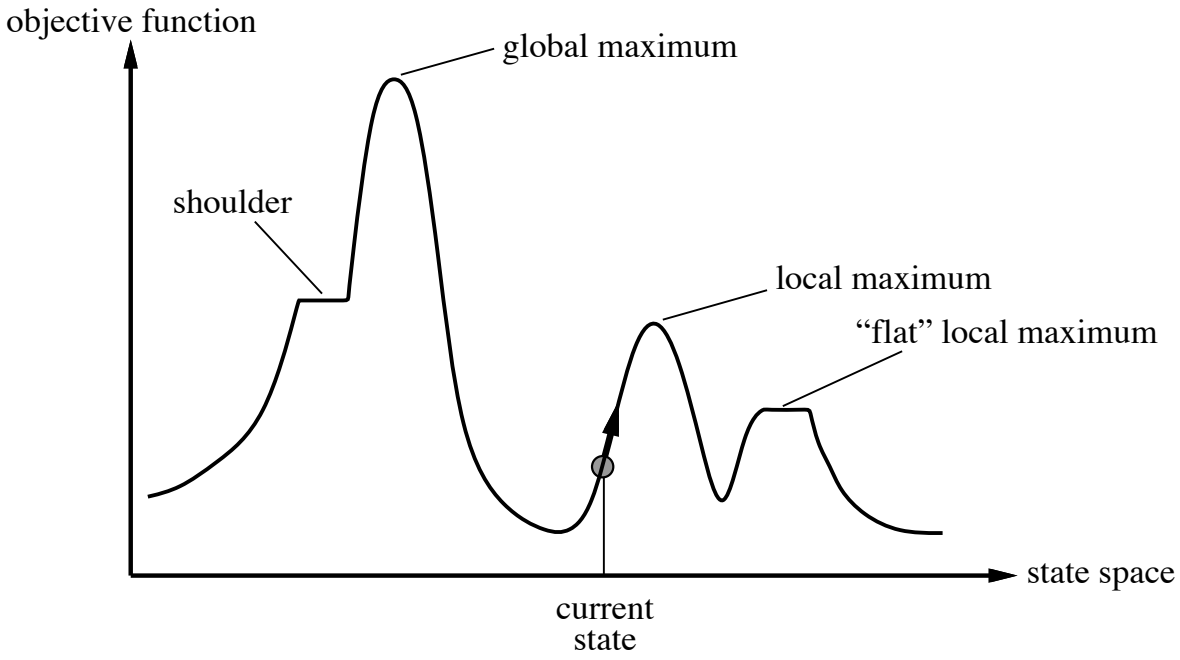
Hill-climbing (or gradient ascent/descent)

- “Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] < VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

Hill-climbing (cont.)
(or gradient ascent/descent)

Problem: depending on initial state, can get stuck on local maxima



Simulated annealing

Idea: escape local maxima by allowing some "bad"' moves *but gradually decrease their size and frequency*

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state

inputs: *problem*, a problem
schedule, a mapping from time to "temperature"

local variables: *current*, a node
next, a node
T, a "temperature" controlling prob. of downward steps

current $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
for $t \leftarrow 1$ **to** ∞ **do**
  $T \leftarrow$ *schedule*[t]
  if $T = 0$ **then return** *current*
  next $\leftarrow$ a randomly selected successor of *current*
  $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
  if $\Delta E > 0$ **then** *current* $\leftarrow$ *next*
  else *current* $\leftarrow$ *next* only with probability $e^{\Delta\ E/T}$

Simulated annealing: Properties

- At fixed "temperature"' T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha^{\frac{E(x)}{kT}}$$

- T decreased slowly enough → always reach best state
- Is this necessarily an interesting guarantee?
- Devised by Metropolis et al., 1953, for physical process modelling
- Widely used in VLSI layout, airline scheduling, etc.

Practical AI

(a.k.a. Business Intelligence)

Part 1- Intelligent Search

Search

(Intelligent) Search

Constraint Satisfaction

Adversarial Search (Games)

Universität Zürich[UZH]

Dynamic and Distributed Information Systems

Overview – Part 1

- Problem Definition?
- Search
  - Breadth first, Depth first, Iterative deepening
  - Best First
- Informed (or Intelligent) Search
  - A*, Hill climbing, Simulated annealing
- **Constraint Satisfaction**
  - Forward checking, Constraint propagation
- Adversarial Search (Games)

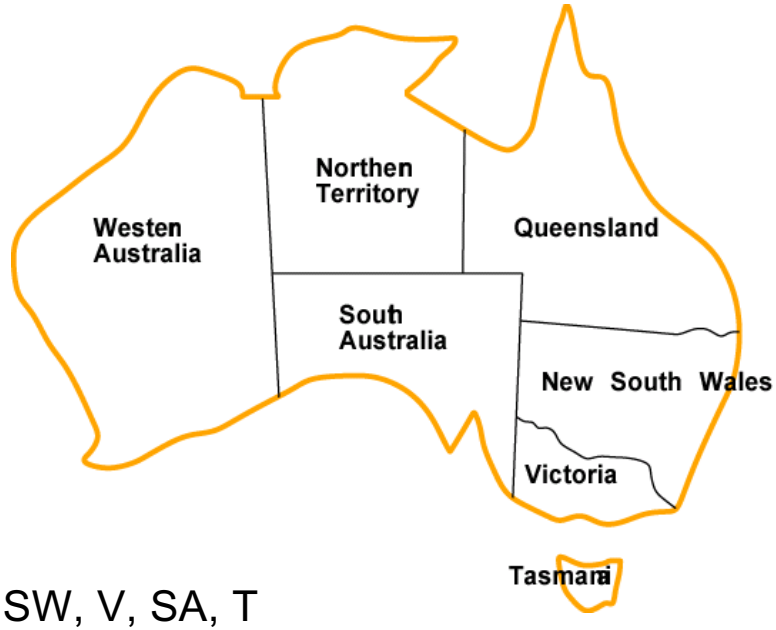
CSP: Outline

- CSP examples
- Backtracking search for CSPs
- Problem structure and problem decomposition
- Local search for CSPs

Constraint satisfaction problems (CSPs)

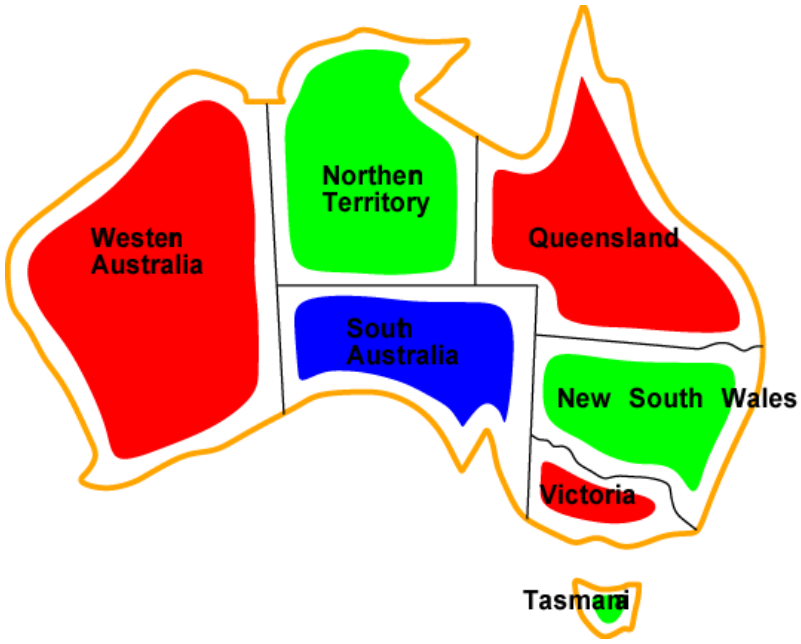
- Standard search problem:
  - ***state*** is a "black box" – any old data structure that supports goal test, eval, successor
- CSP:
  - ***state*** is defined by *variables* X_i with *values* from domain D_i
  - ***goal test*** is a set of *constraints* specifying allowable combinations of values for subsets of variables
- Simple example of a *formal representation language*
- Allows useful *general-purpose* algorithms with more power than standard search algorithms

Example: Map-Coloring



- ***Variables***: WA, NT, Q, NSW, V, SA, T
- ***Domains***: D_i = {red, green, blue}
- ***Constraints***: adjacent regions must have different colors
  - e.g., WA ≠ NT (if the language allows this), or
    (WA,NT) ∈ {(red, green), (red, blue), (green, red), (green, blue) ,…}

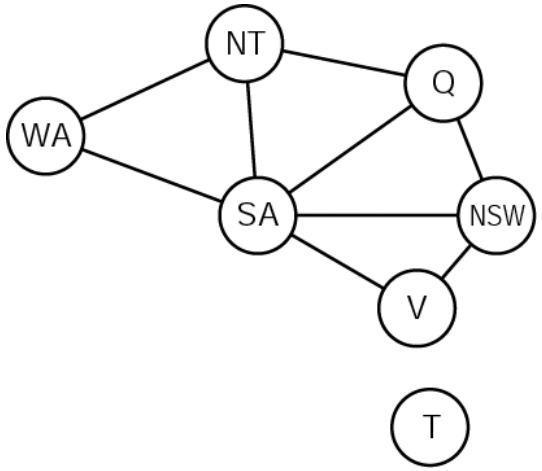
Example: Map-Coloring (contd.)



- ***Solutions*** are assignments satisfying all constraints,
  - e.g., {WA= red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green}

Constraint graph

- *Binary CSP*: each constraint relates at most two variables
- *Constraint graph*: nodes are variables, arcs show constraints



- General-purpose CSP algorithms use the graph structure to speed up search.
  - E.g., Tasmania is an independent subproblem!

Varieties of CSPs

- Discrete variables
  - finite domains; size $d \Rightarrow O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl.Boolean satisfiability (NP-complete)
  - infinite domains (integers, strings, etc.)
    - e.g., job scheduling, variables are start/end days for each job
    - need a *constraint language*, e.g., $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$
    - *linear* constraints solvable, *nonlinear* undecidable
- Continuous variables
  - e.g., start/end times for Hubble Telescope observations
  - linear constraints solvable in poly time by LP methods

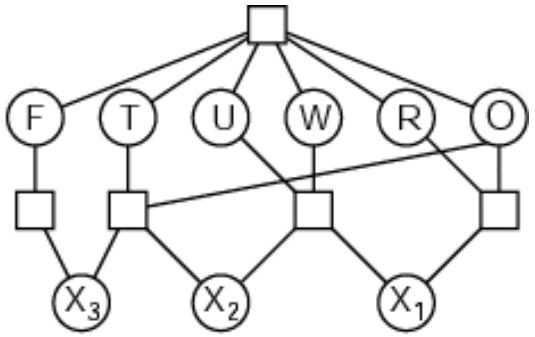
Varieties of constraints

- *Unary* constraints involve a single variable,
  - e.g., *SA* ≠ *green*
- *Binary* constraints involve pairs of variables,
  - e.g., *SA* ≠ *WA*
- *Higher-order* constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints
- *Preferences* (soft constraints), e.g., *red* is better than *green* often representable by a cost for each variable assignment

→ constrained optimization problems

Example: Cryptarithmetic

```
  T W O
+ T W O
-------
F O U R
```



- *Variables*: $F\ T\ U\ W\ R\ O\ X_1, X_2\ X_3$
- *Domains*: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- *Constraints*:
  - *alldiff(F, T, U, W, R, O)*
  - $O + O = R + 10 * X_1$, etc.

Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling
- Floorplanning

- *Note:* that many real-world problems involve real-valued variables

Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

- ***Initial state***: the empty assignment, {}
- ***Successor function***: assign a value to an unassigned variable that does not conflict with current assignment.
- ⇒ fail if no legal assignments (not fixable!)
- ***Goal test***: the current assignment is complete

1) This is the same for all CSPs!
2) Every solution appears at depth *n* with *n* variables
   ⇒ use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) $b = (n-l)d$ at depth l, hence $n!d^n$ leaves!!!!

Backtracking search

- Variable assignments are *commutative*, i.e.,
  - [*WA* = *red* then *NT* = *green*] same as [*NT* = *green* then *WA* = *red*]
- Only need to consider assignments to a single variable at each node

  $\Rightarrow$ $b = d$ and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called *backtracking* search
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve *n*-queens for $n \approx 25$

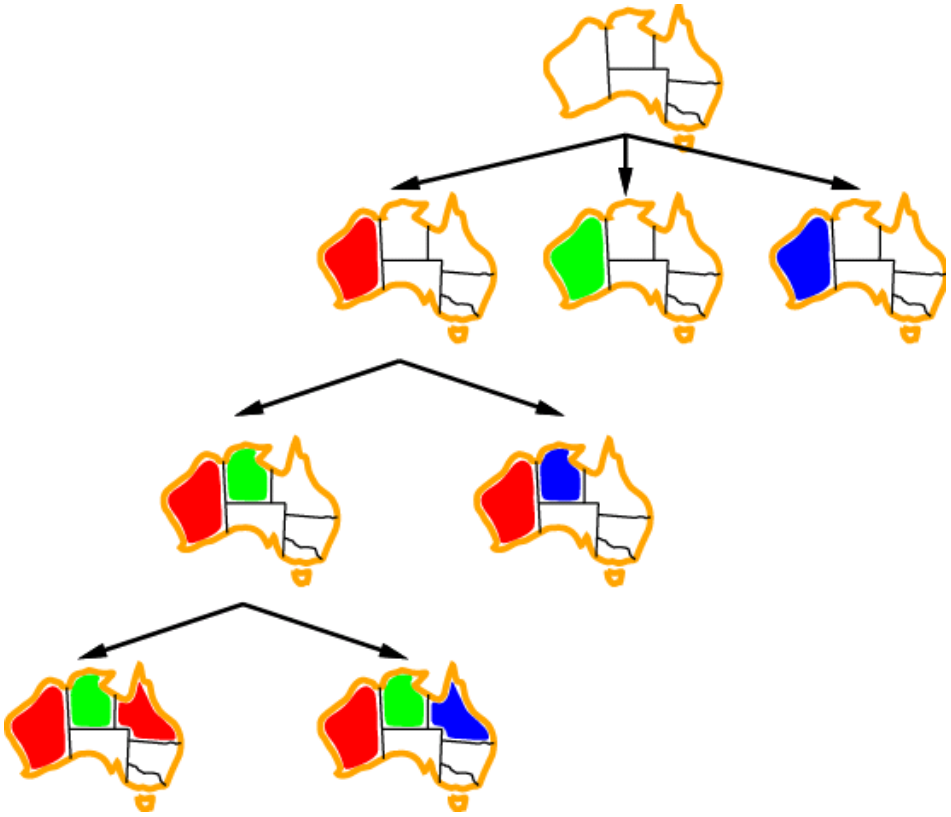
Backtracking search

function BACKTRACKING-SEARCH(*csp*) **returns** solution/failure
  return RECURSIVE-BACKTRACKING([ ], *csp*)

function RECURSIVE-BACKTRACKING(*assigned*, *csp*) **returns** solution/failure
  if *assigned* is complete **then return** *assigned*
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assigned*, *csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assigned*, *csp*) **do**
    if *value* is consistent with *assigned* according to CONSTRAINTS[*csp*] **then**
      result ← RECURSIVE-BACKTRACKING([*var* = *value*|*assigned*], *csp*)
      if *result* ≠ *failure* **then return** *result*
  end
  return *failure*

Backtracking example

Backtracking example



Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Most constrained variable

Most constrained variable:

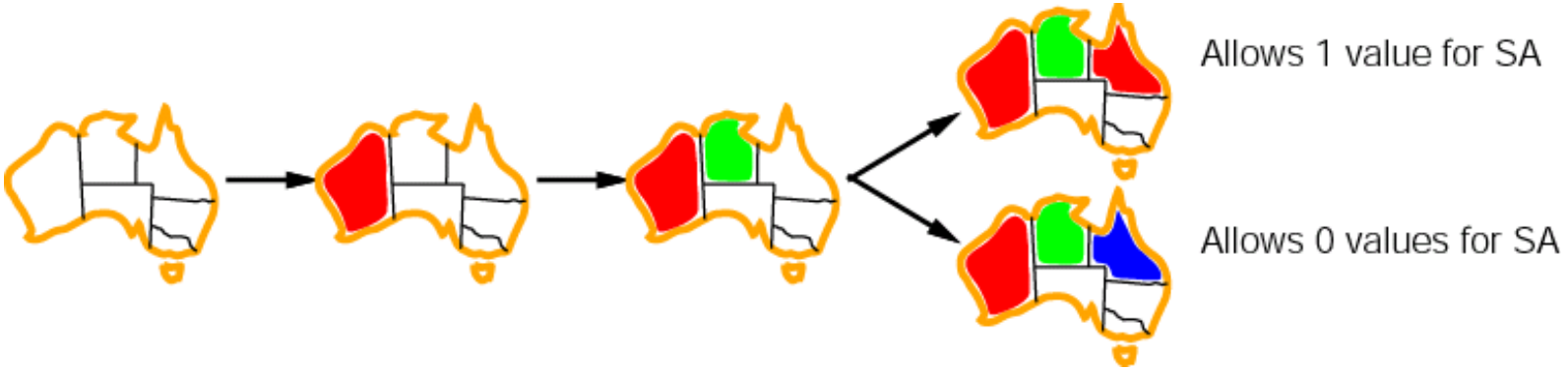
- choose the variable with the fewest legal values

Most constraining variable

- Tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables

Least constraining value

- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

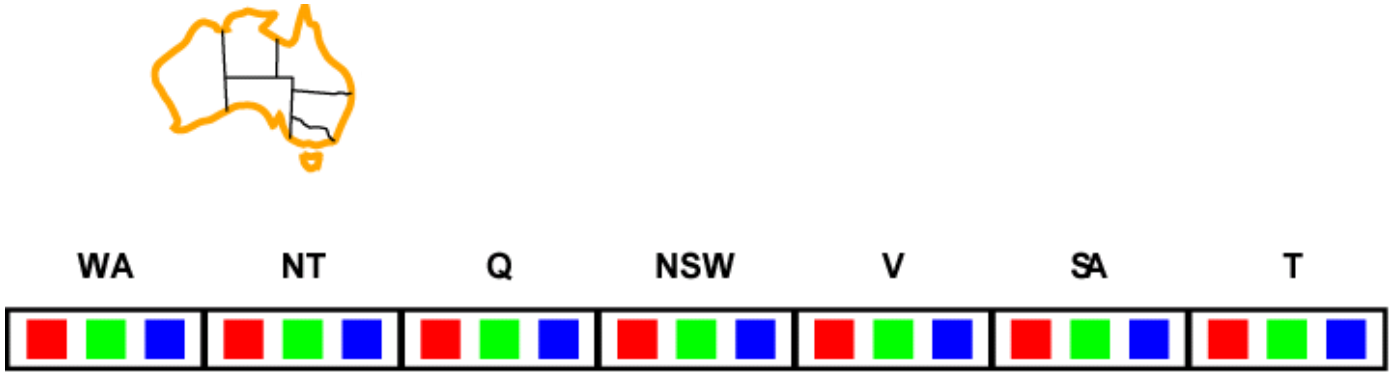


Combining these heuristics makes 1000 queens feasible

Forward checking

Idea: Keep track of remaining legal values for unassigned variables

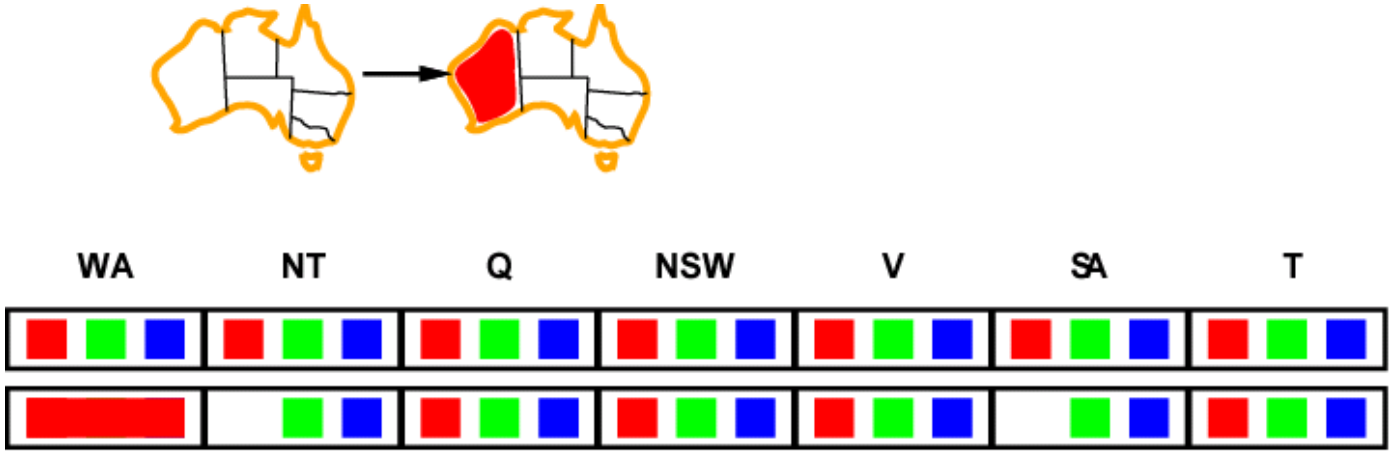
Terminate search when any variable has no legal values



Forward checking

Idea: Keep track of remaining legal values for unassigned variables

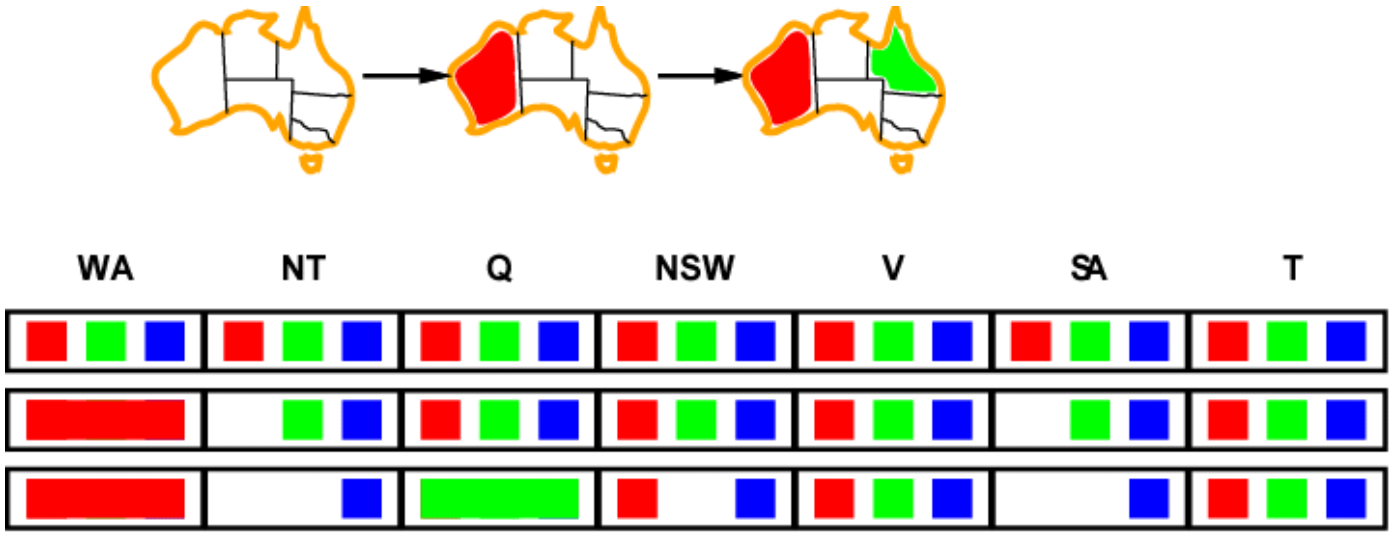
Terminate search when any variable has no legal values



Forward checking

Idea: Keep track of remaining legal values for unassigned variables

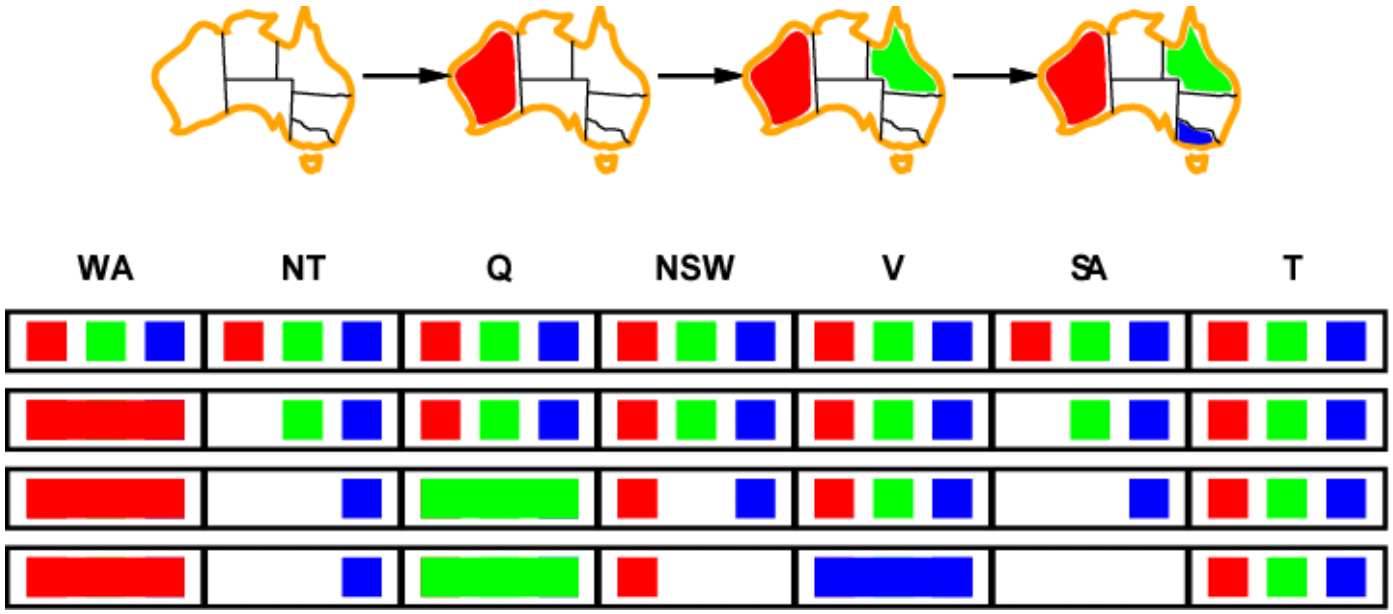
Terminate search when any variable has no legal values



Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values



Constraint propagation

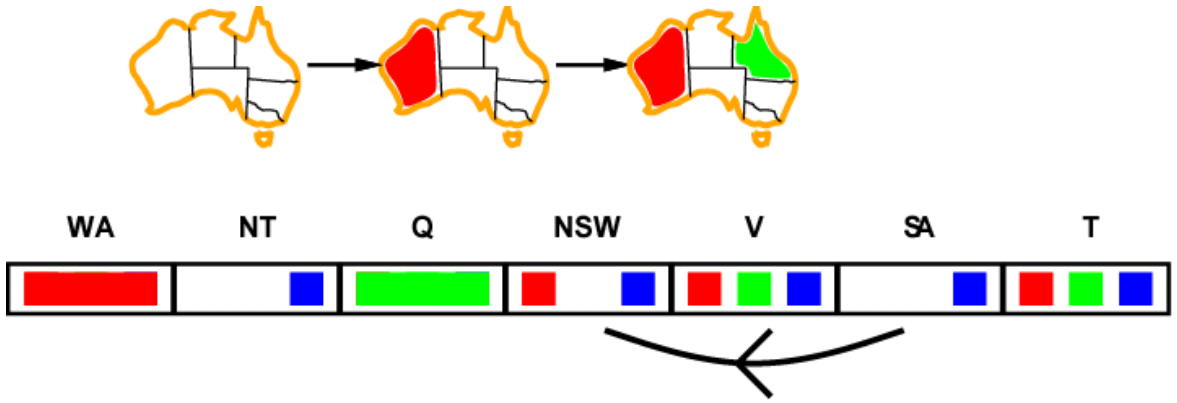
Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc consistency

- Simplest form of propagation makes each arc *consistent*
- *X* → *Y* is consistent *iff*
  - for *every* value *x* of *X* there is *some* allowed *y*

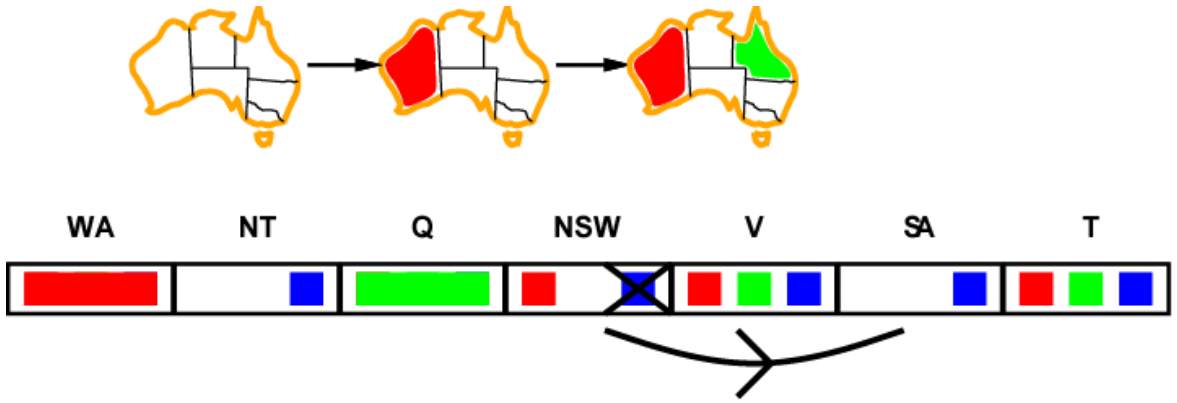


Arc consistency

Simplest form of propagation makes each arc *consistent*

$X \rightarrow Y$ is consistent *iff*

- for *every* value *x* of *X* there is *some* allowed *y*

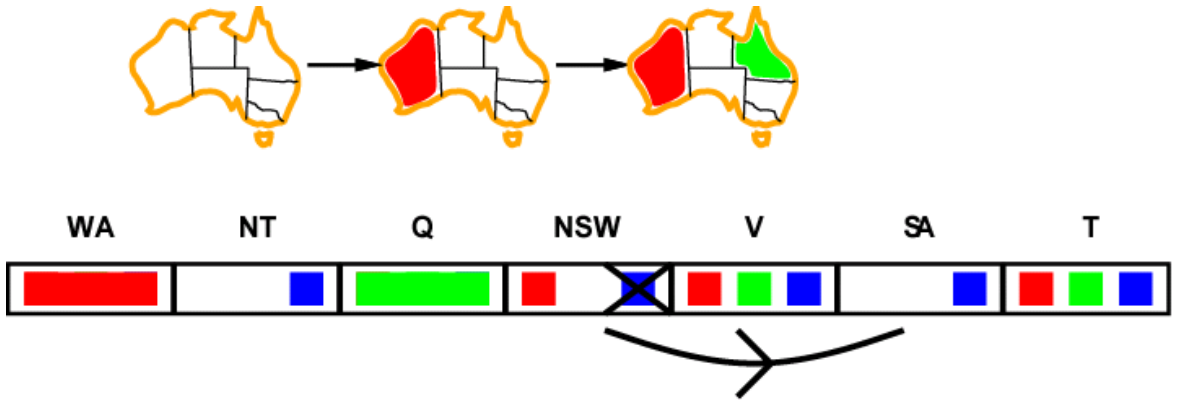


Arc consistency

Simplest form of propagation makes each arc *consistent*

$X \rightarrow Y$ is consistent *iff*

- for *every* value *x* of *X* there is *some* allowed *y*

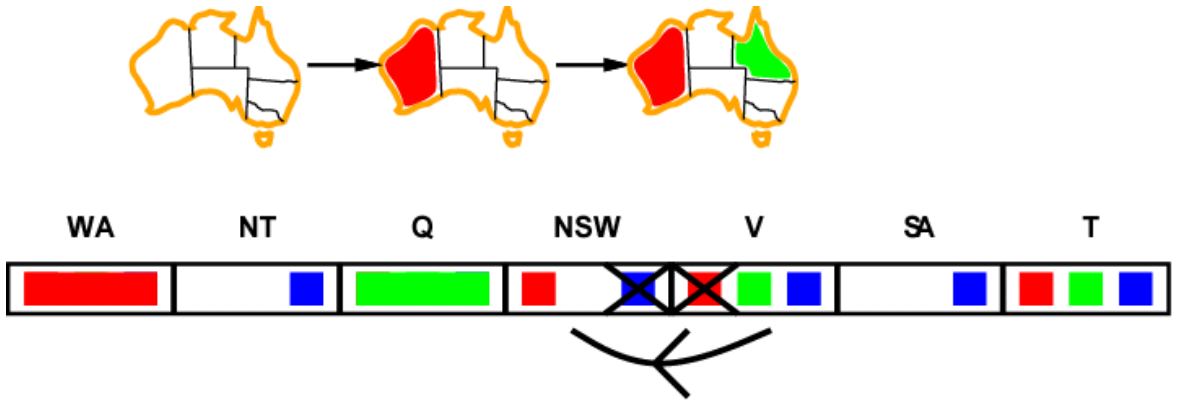


Arc consistency

Simplest form of propagation makes each arc *consistent*

X → *Y* is consistent *iff*

- for *every* value *x* of *X* there is *some* allowed *y*



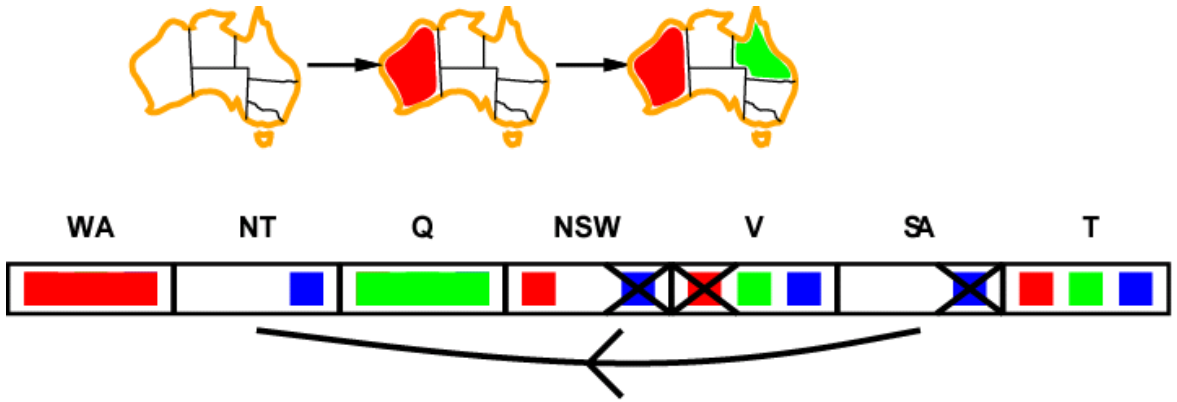
If *X* loses a value, neighbors of X need to be rechecked

Arc consistency

Simplest form of propagation makes each arc *consistent*

$X \rightarrow Y$ is consistent *iff*

- for *every* value *x* of *X* there is *some* allowed *y*



If *X* loses a value, neighbors of X need to be rechecked

Arc consistency detect failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc consistency algorithm

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

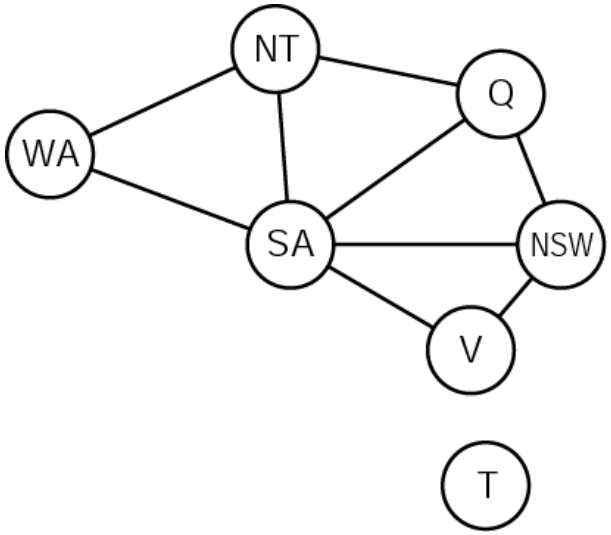
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*

---

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns**
true iff we remove a value
  removed $\leftarrow$ *false*
  for each x **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy constraint between X_i and X_j
      then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

- $O(n^2 d^3)$, can be reduced to $O(n^2 d^2)$ but cannot detect all failures in poly time!

Problem structure

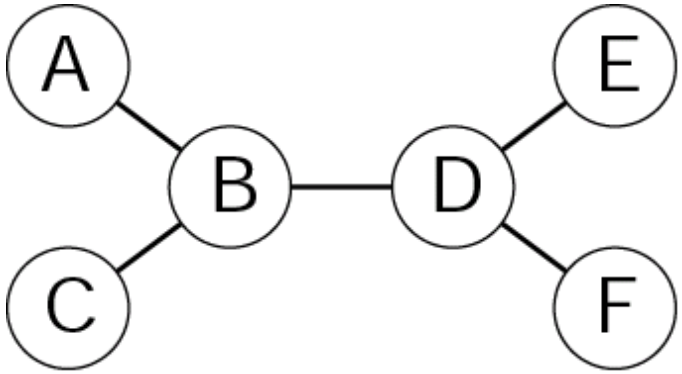


- Tasmania and mainland are *independent subproblems*
- Identifiable as *connected components* of constraint graph

Problem structure (contd.)

- Suppose each subproblem has *c* variables out of *n* total
- Worst-case solution cost is n/c * d^c, *linear* in *n*
- E.g., *n* = 80, *d* = 2, *c* = 20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - 4 * 2^^{20} = 0.4 seconds at 10 million nodes/sec

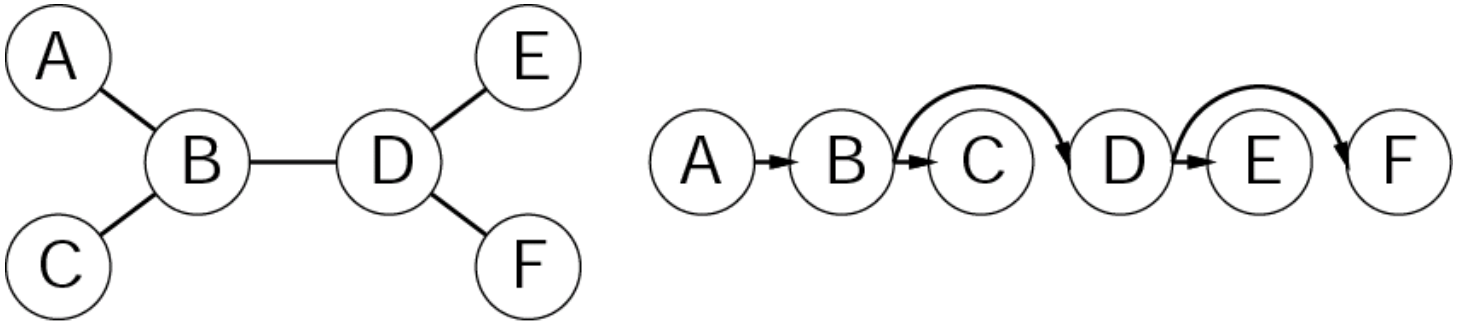
Tree-structured CSPs



- ***Theorem***: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
- Compare to general CSPs, where worst-case time is $O(d^n)$
- This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.

Algorithm for tree-structured CSPs

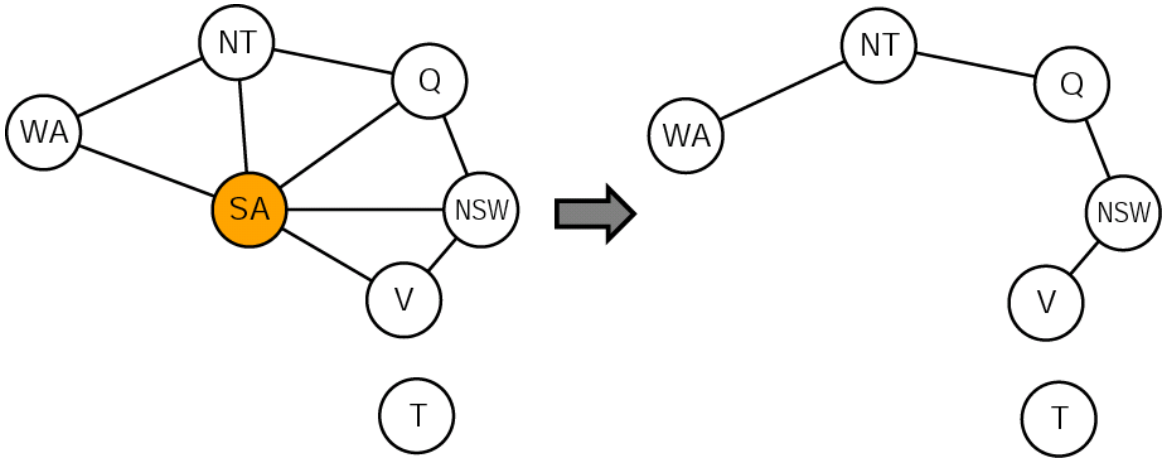
1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



2. For j from n down to 2, apply **`RemoveInconsistent`**$(Parent(X_j),X_j)$
3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

Nearly tree-structured CSPs

- ***Conditioning***: instantiate a variable, prune its neighbors' domains



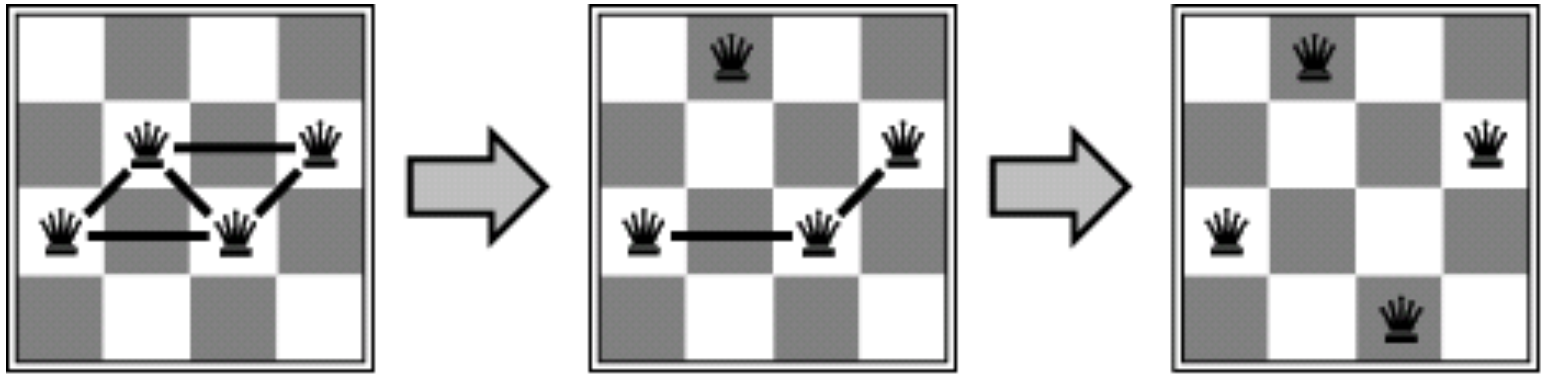
- ***Cutset conditioning***: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size $\Rightarrow$ runtime $O(d^c \bullet (n-c)d^2)$, very fast for small c

Iterative algorithms for CSPs

- Hill-climbing, simulated annealing typically work with "complete"' states, i.e., all variables assigned
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators *reassign* variable values
- Variable selection: randomly select any conflicted variable
- Value selection by *min-conflicts* heuristic:
  - choose value that violates the fewest constraints
  - i.e., hillclimb with *h(n)* = total number of violated constraints

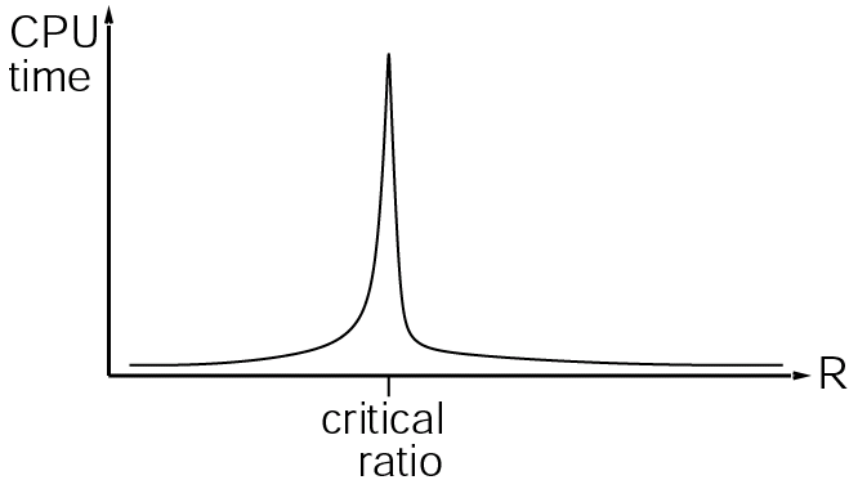
Example: 4-Queens

- ***States***: 4 queens in 4 columns (4^4 = 256 states)
- ***Operators***: move queen in column
- ***Goal test***: no attacks
- ***Evaluation***: *h(n)* = number of attacks



Performance of min-conflicts

- Given random initial state, can solve *n*-queens in almost constant time for arbitrary *n* with high probability (e.g., *n* = 10,000,000)
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio: $R = \frac{\text{number of constraints}}{\text{number of variables}}$



Summary

- CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test defined by *constraints* on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- The CSP representation allows analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice

Practical AI

(a.k.a. Business Intelligence)

Part 1- Intelligent Search

Search

(Intelligent) Search

Constraint Satisfaction

Adversarial Search (Games)

Universität Zürich[UZH]

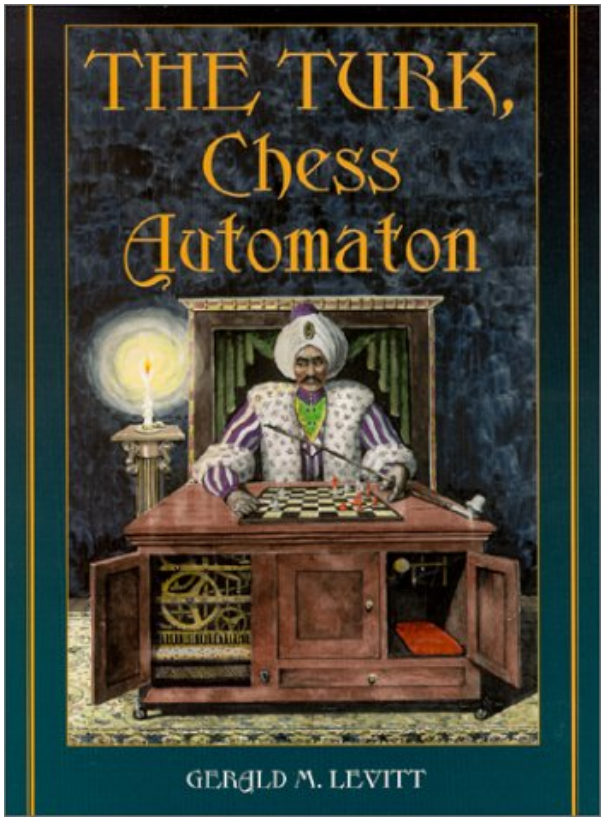
Dynamic and Distributed Information Systems

Overview – Part 1

- Problem Definition?
- Search
  - Breadth first, Depth first, Iterative deepening
  - Best First
- Informed (or Intelligent) Search
  - A*, Hill climbing, Simulated annealing
- Constraint Satisfaction
  - Forward checking, Constraint propagation
- **Adversarial Search (Games)**

Outline

- Perfect play
- Resource limits
- α–β pruning
- Games of chance
- Games of imperfect information



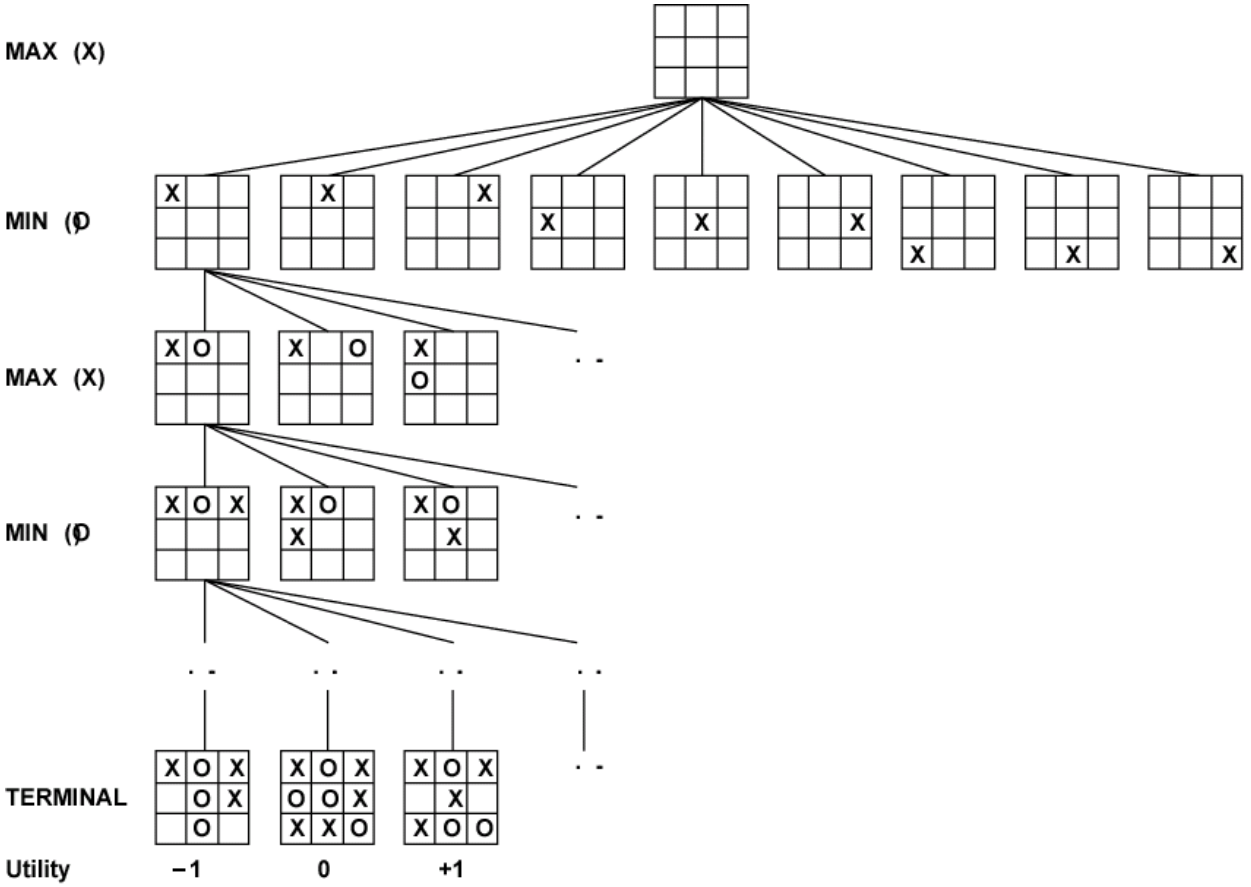
Games vs. search problems

- „Unpredictable“ opponent → solution is a strategy specifying a move for every possible opponent reply
- Time limits → unlikely to find goal, must approximate
- Plan of attack:
  - Computer considers possible lines of play (Babbage, 1846)
  - Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
  - Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
  - First chess program (Turing, 1951)
  - Machine learning to improve evaluation accuracy (Samuel, 1952-57)
  - Pruning to allow deeper search (McCarthy, 1956)

Types of games

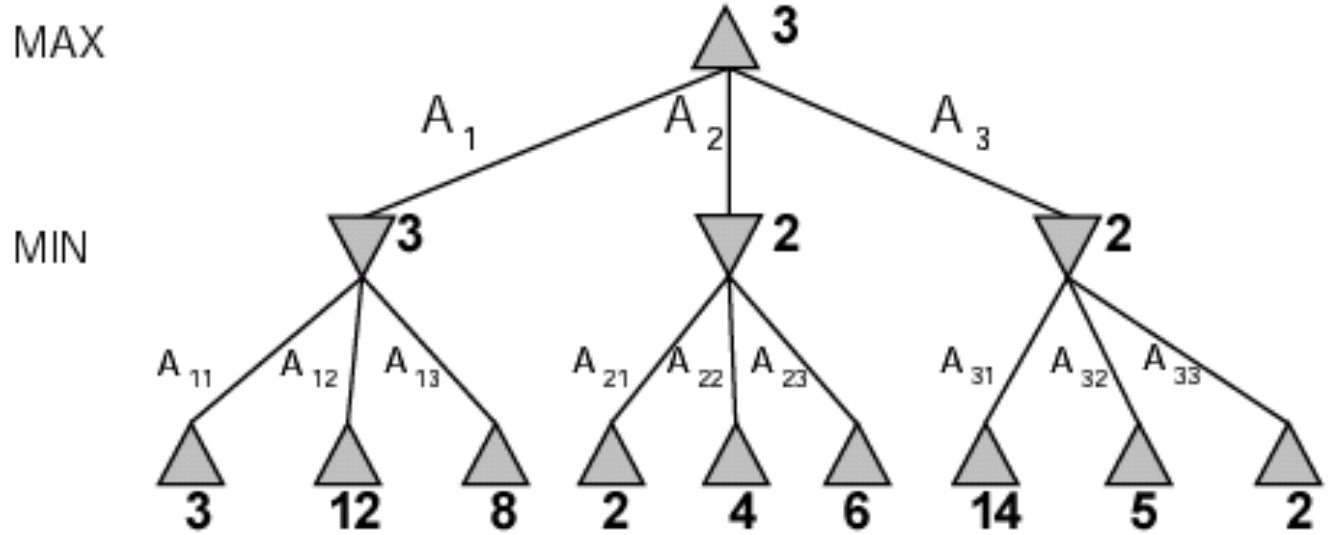
	Deterministic	*Chance*
Perfect information	Chess, checkers, go, othello	Backgammon, monopoly
Imperfect information		Bridge, poker, scrabble, nuclear war

Game tree (2-player, deterministic, turns)



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*, *game*) **returns** *an action*

action, *state* $\leftarrow$ the a, s **in** SUCCESSORS(*state*)
such that MINIMAX-VALUE(s, *game*) is maximized
return *action*

---

function MINIMAX-VALUE(*state*, *game*) **returns** *a utility value*

if TERMINAL-TEST(*state*) **then**
return UTILITY(*state*)
else if MAX is to move in *state* **then**
return the highest MINIMAX-VALUE of SUCCESSORS(*state*)
else
return the lowest MINIMAX-VALUE of SUCCESSORS(*state*)

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
  → exact solution completely infeasible

Resource limits

- Suppose we have 100 seconds, explore 10^4 nodes/second

  → 10^6 nodes per move

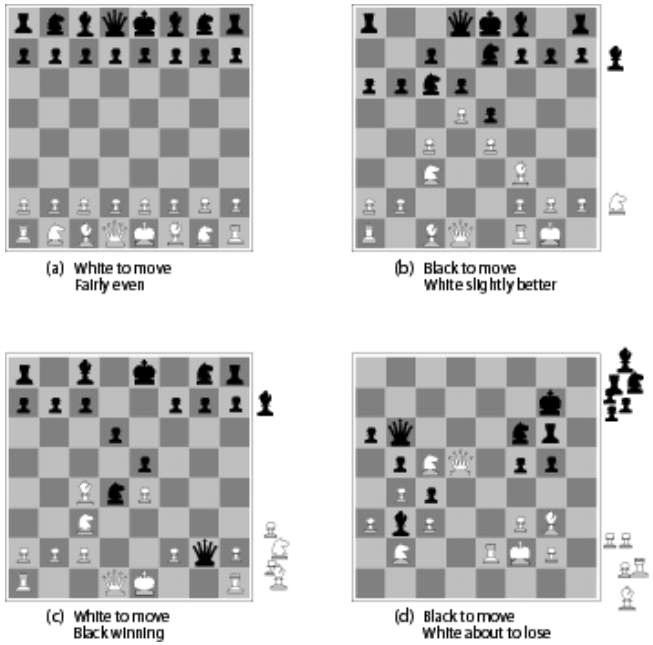
- Standard approach:
  - *cutoff test*

    e.g., depth limit (perhaps add *quiescence search*)
  - *evaluation function*

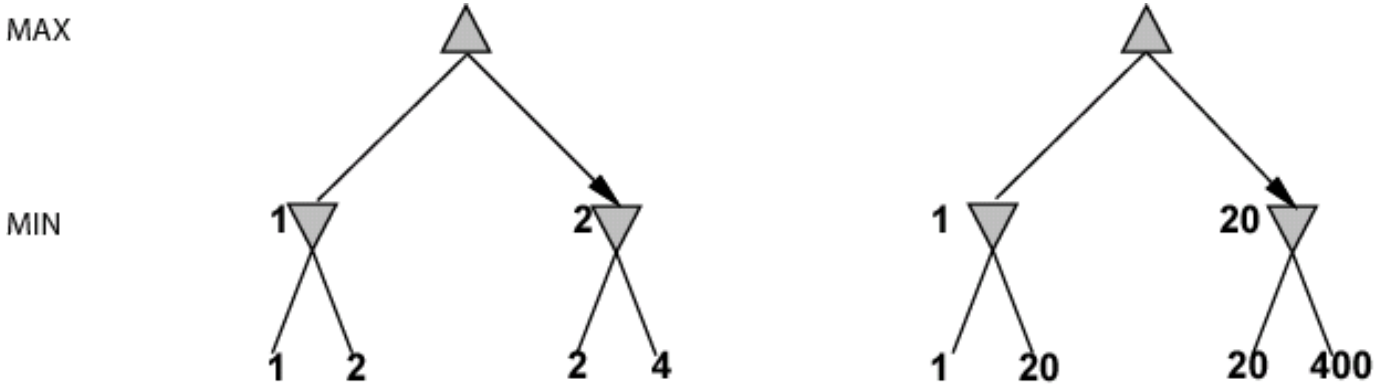
    = estimated desirability of position

Evaluation functions

- For chess, typically *linear* weighted sum of *features*
  $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
- e.g., $w_1 = 9$ with
  $f_1(s)$ = (number of white queens) – (number of black queens), etc.



Digression: Exact values don't matter



- Behaviour is preserved under any *monotonic* transformation of **`Eval`**
- Only the order matters:

  payoff in deterministic games acts as an *ordinal utility* function

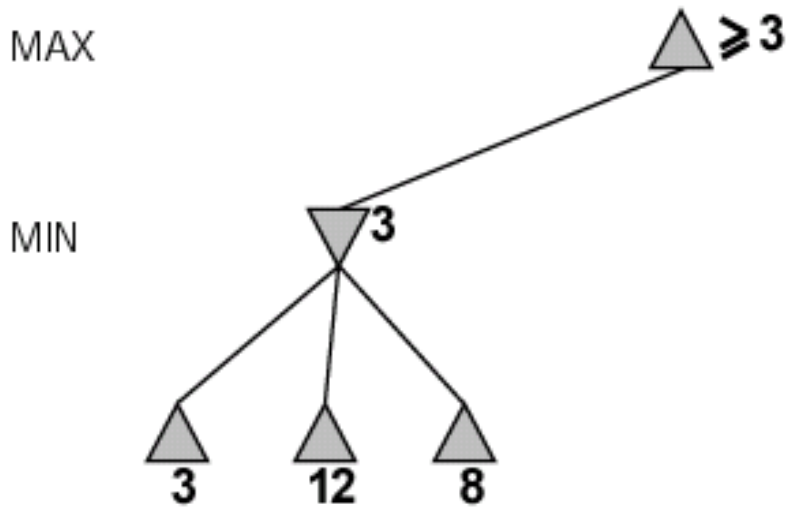
Cutting off search

- **`MinimaxCutoff`** is identical to **`MinimaxValue`** except:
  1. **`Terminal?`** is replaced by **`Cutoff?`**
  2. **`Utility`** is replaced by **`Eval`**

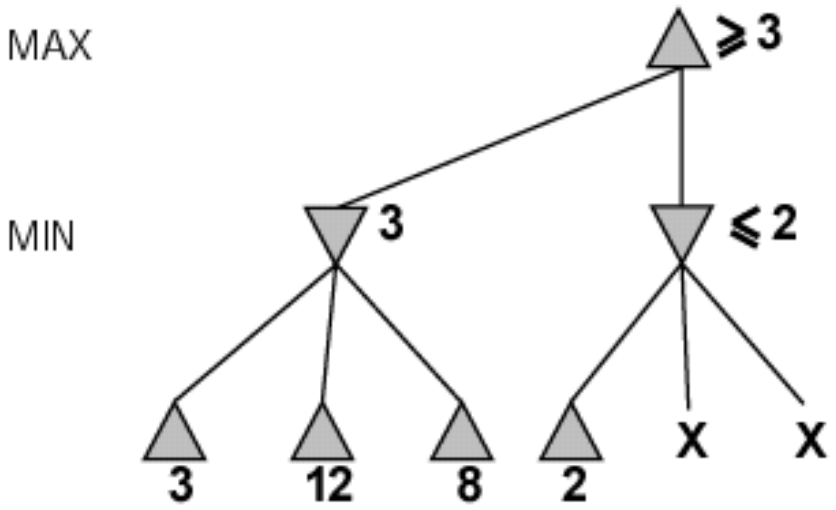
- Does it work in practice?
  - $b^m = 10^6$, $b=35$ → $m=4$

  - 4-ply lookahead is a hopeless chess player!
  - 4-ply ≈ human novice
  - 8-ply ≈ typical PC, human master
  - 12-ply ≈ Deep Blue, Kasparov

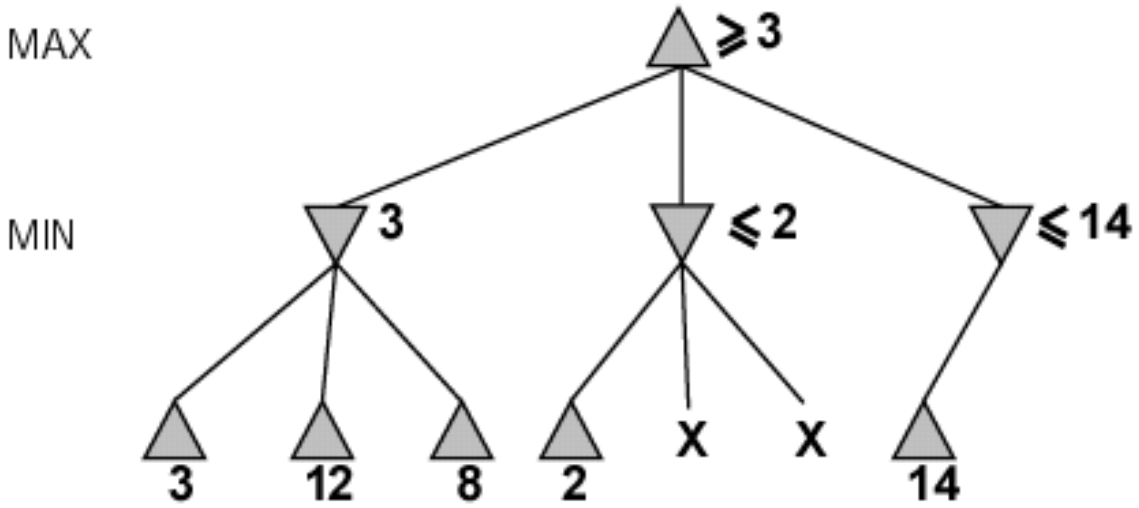
α–β pruning example



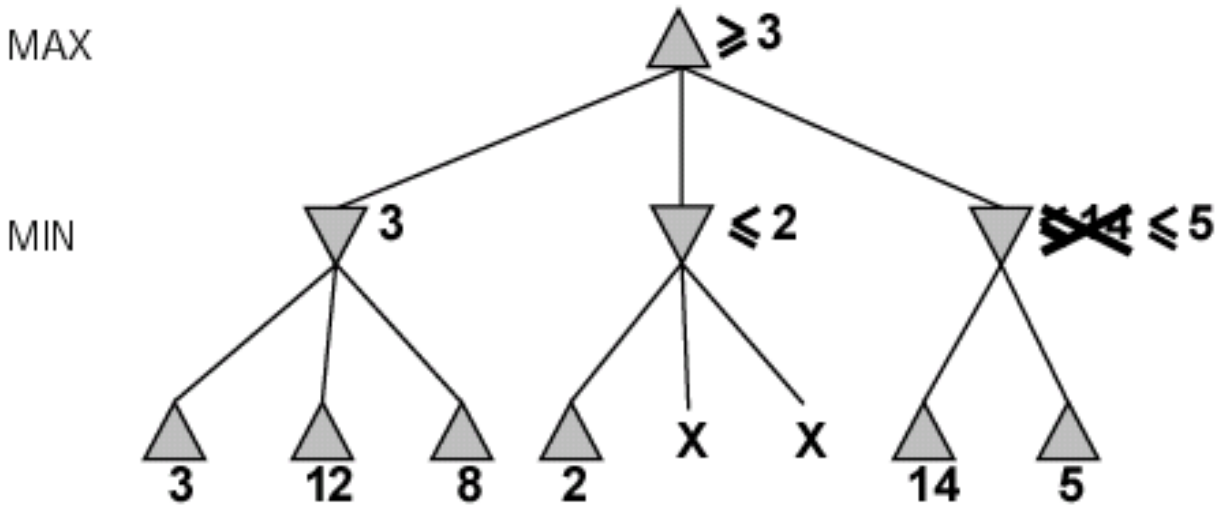
α–β pruning example



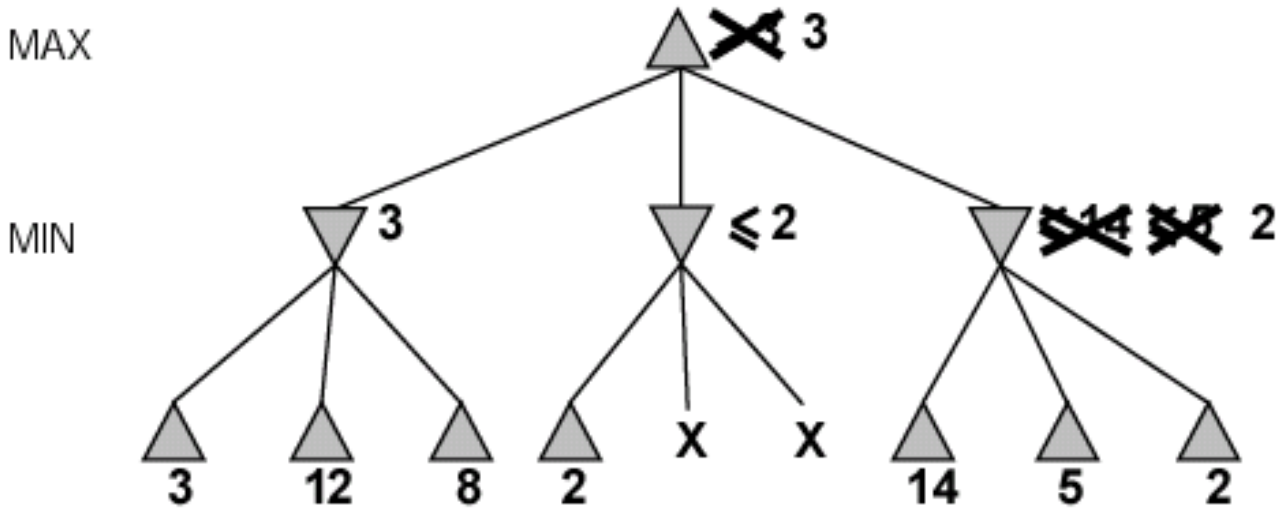
α–β pruning example



α–β pruning example



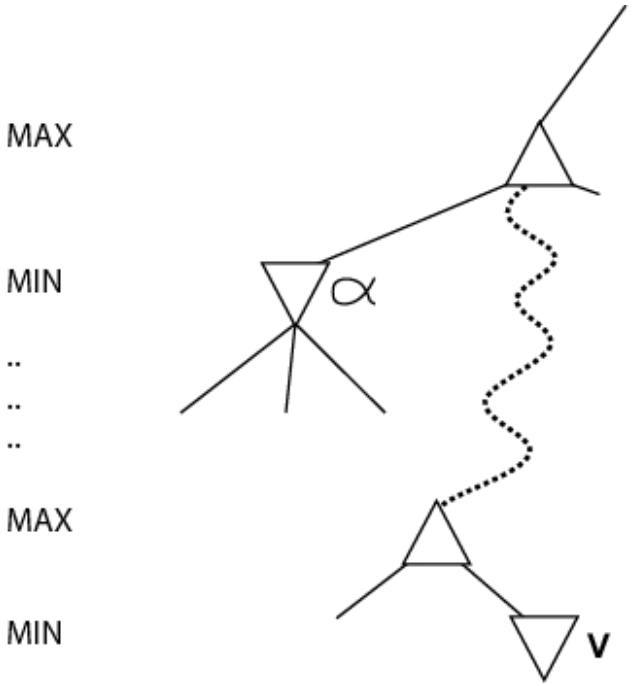
α–β pruning example



Properties of α–β

- Pruning *does not* affect final result
- Good move ordering improves effectiveness of pruning
- With „perfect ordering," time complexity = $O(b^{m/2})$
  - → *doubles* depth of search
  - → can easily reach depth 8 and play good chess
- A simple example of the value of reasoning about which computations are relevant (a form of *metareasoning*)

Why is it called α–β?



- α is the best value **`Max`** found so far off the current path
- If *V* is worse than α **`Max`** will avoid it $\Rightarrow$ prune that branch
- Define β similarly for **`Min`**

The α–β algorithm

function ALPHA-BETA-SEARCH(*state*, *game*) **returns** an action
  action, *state* ← the *a*, *s* **in** SUCCESSORS[*game*](*state*)
    such that MIN-VALUE(*s*, *game*, $-\infty$, $+\infty$) is maximized
  return *action*

---

function MAX-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\alpha \leftarrow \max(\alpha,$ MIN-VALUE$(s, game, \alpha, \beta))$
    if $\alpha \geq \beta$ **then return** β
  return α

---

function MIN-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow \min(\beta,$ MAX-VALUE$(s, game, \alpha, \beta))$
    if $\beta \leq \alpha$ **then return** α
  return β

Deterministic games in practice

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

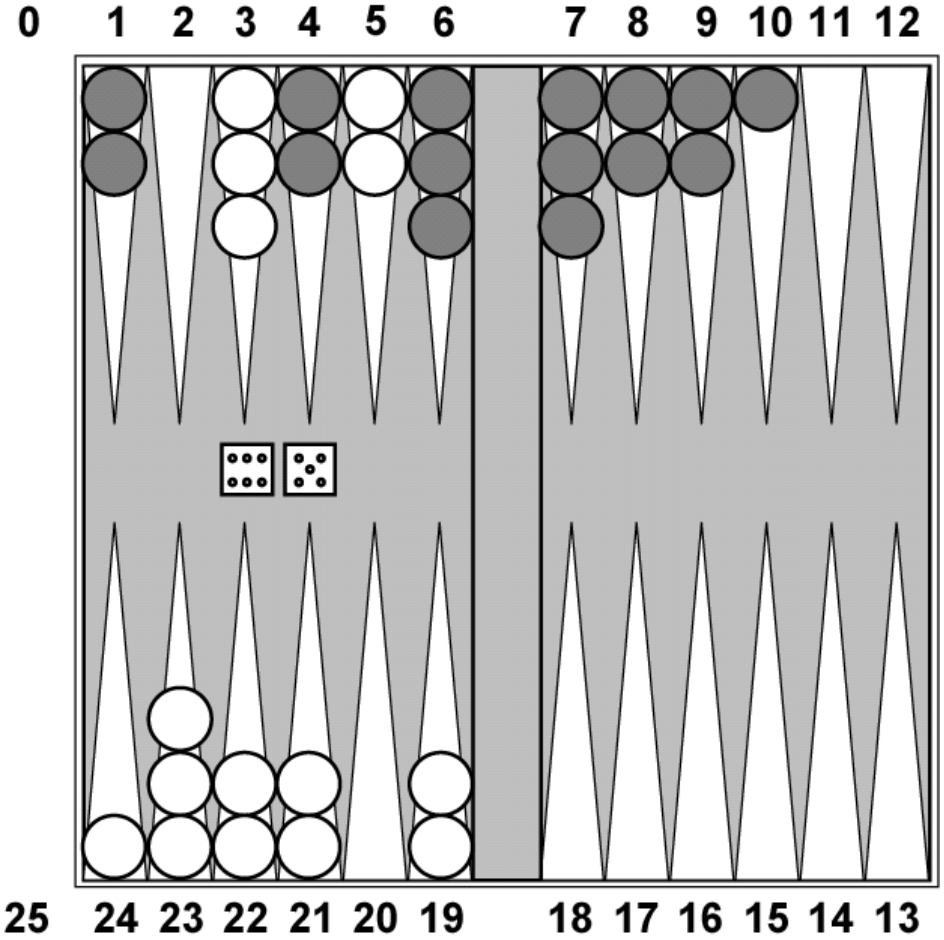
Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

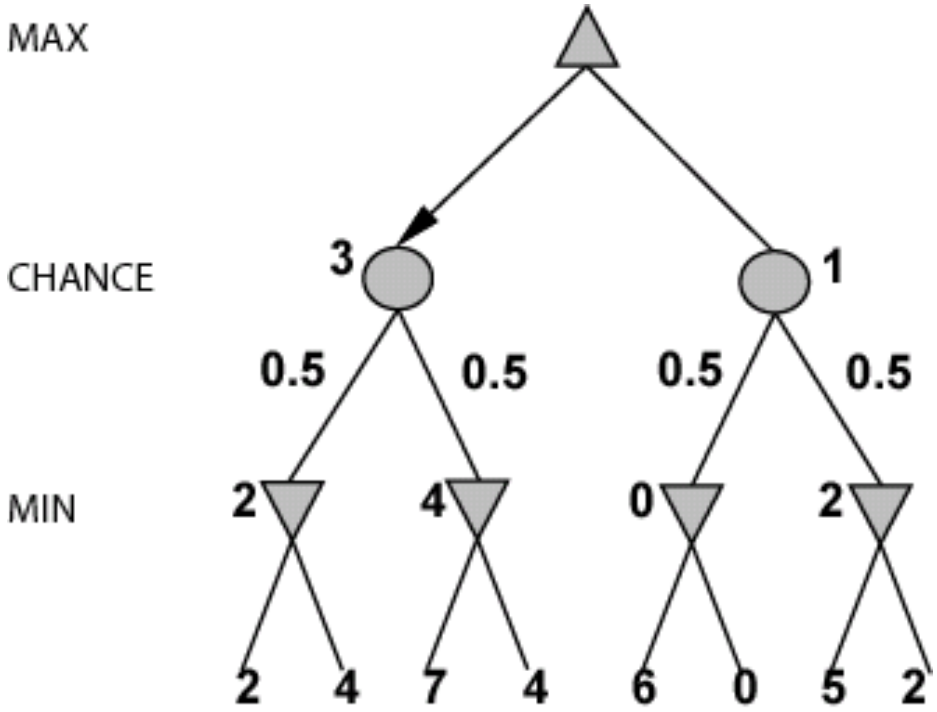
What happened here?
Monte Carlo Tree search and ML

Nondeterministic games: backgammon



Nondeterministic games in general

- In nondeterministic games, chance introduced by dice, card-shuffling
- Simplified example with coin-flipping:



Algorithm for nondeterministic games

- **`Expectiminimax`** gives perfect play
- Just like **`Minimax`**, except we must also handle chance nodes:

…

If *state* is a **`Max`** node then
    return the highest **`ExpectiMinimax-Value`** of **`Successors`** (*state*)

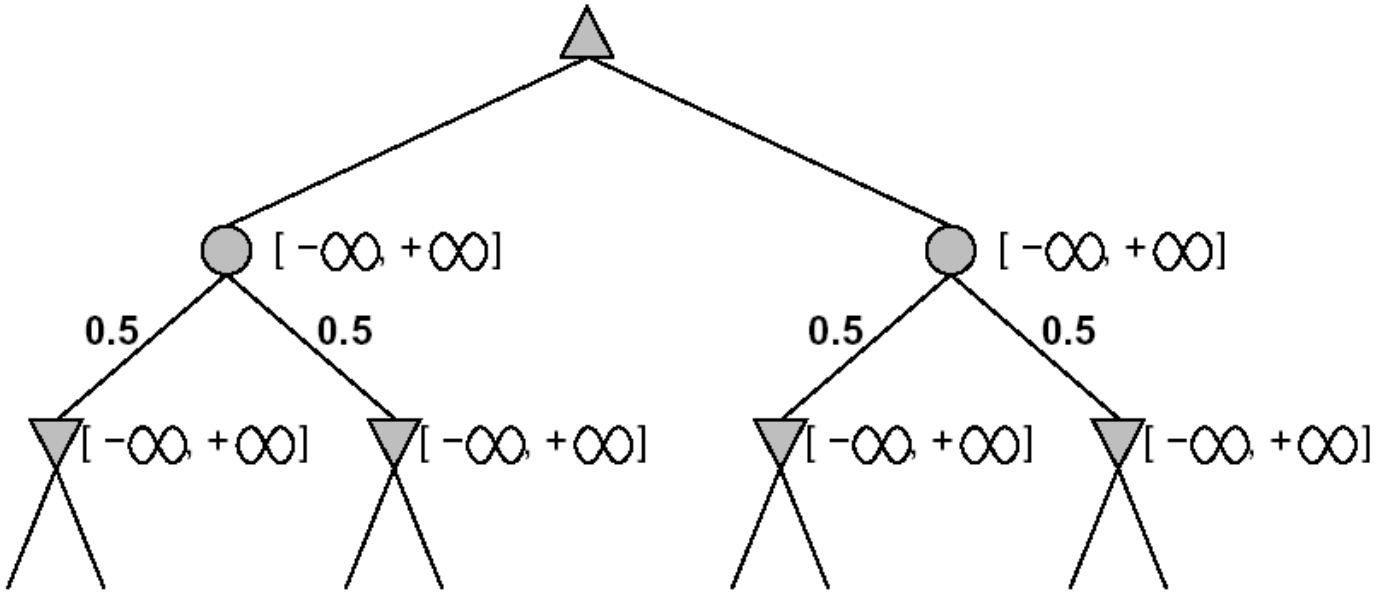
If *state* is a **`Min`** node then
    return the lowest **`ExpectiMinimax-Value`** of **`Successors`** (*state*)

If *state* is a chance node then
    return the average **`ExpectiMinimax-Value`** of **`Successors`** (*state*)

…

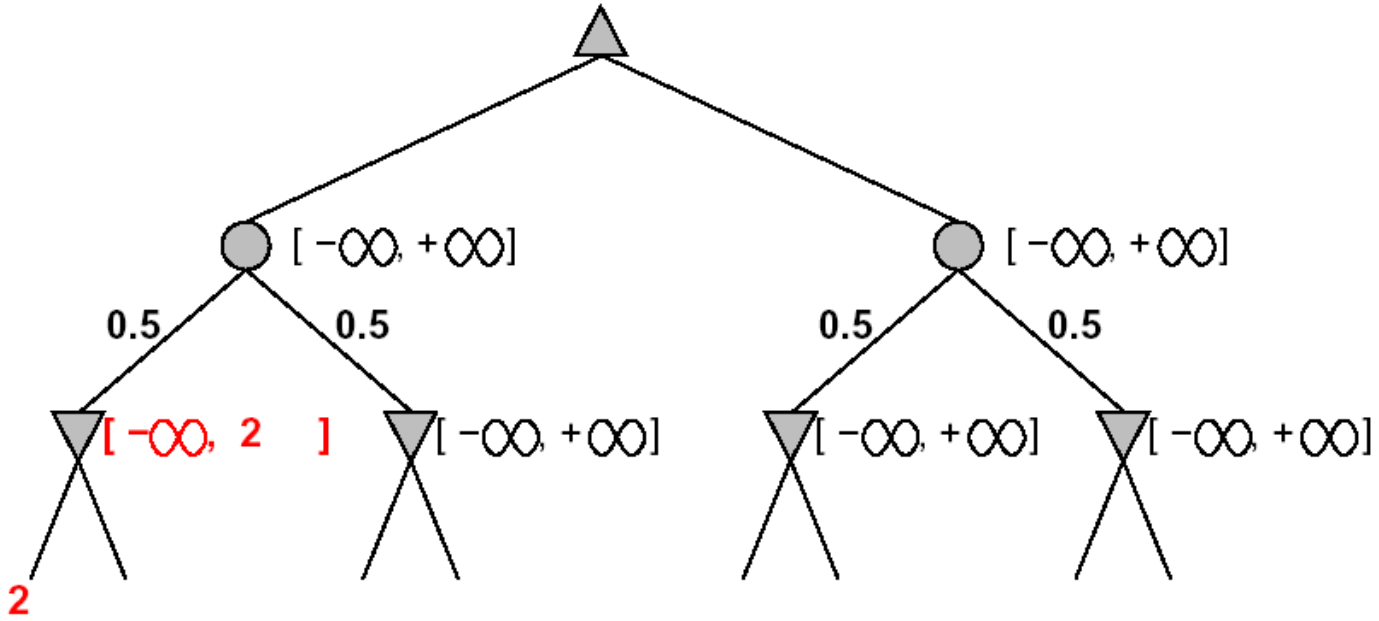
Pruning in nondeterministic game trees

- A version of α–β pruning is possible:



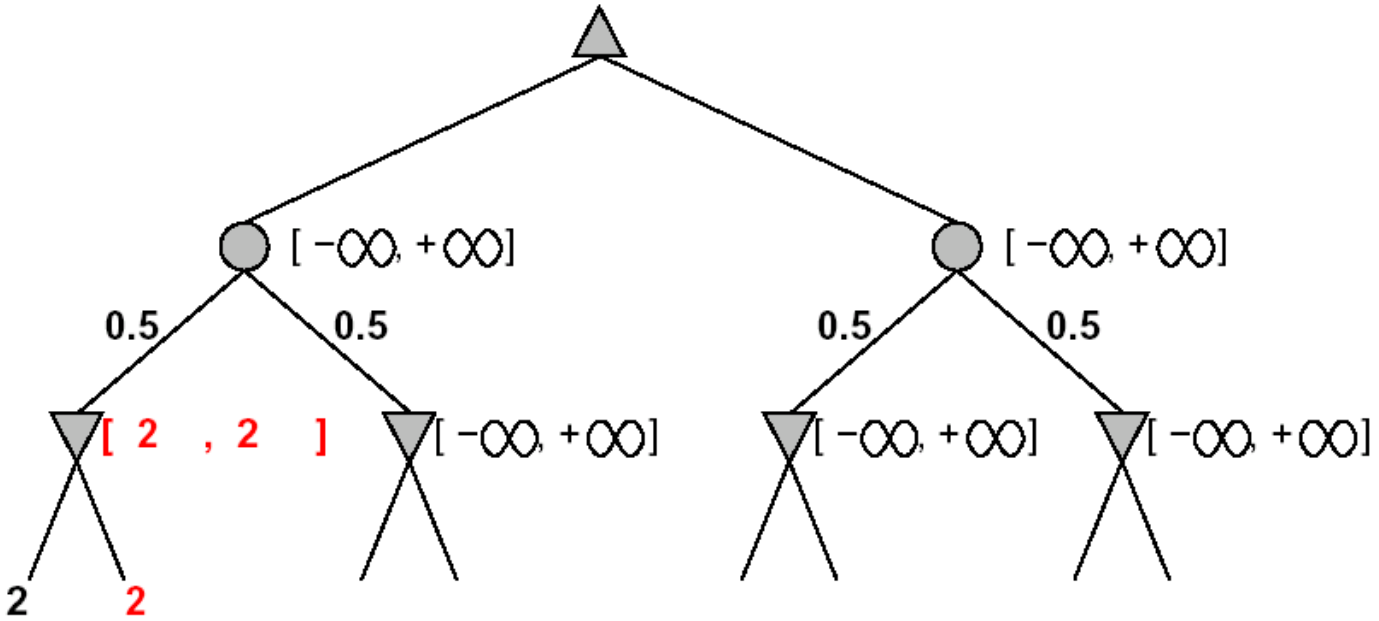
Pruning in nondeterministic game trees

- A version of α–β pruning is possible:



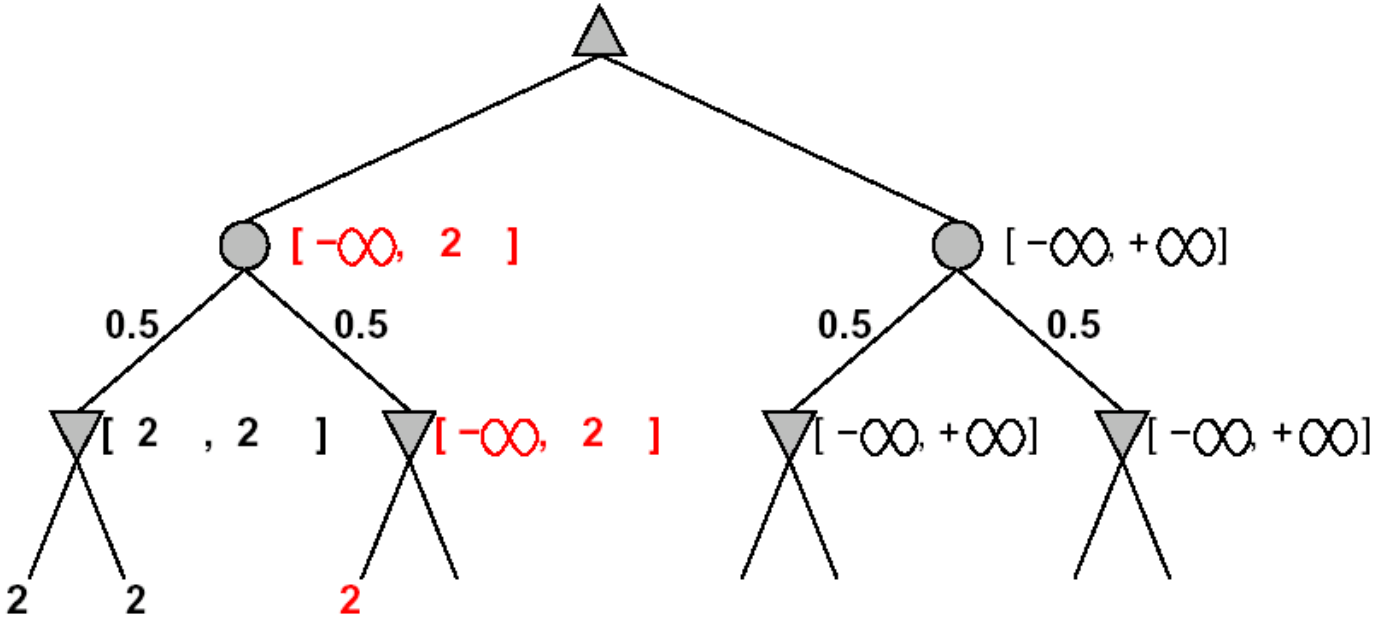
Pruning in nondeterministic game trees

- A version of α–β pruning is possible:



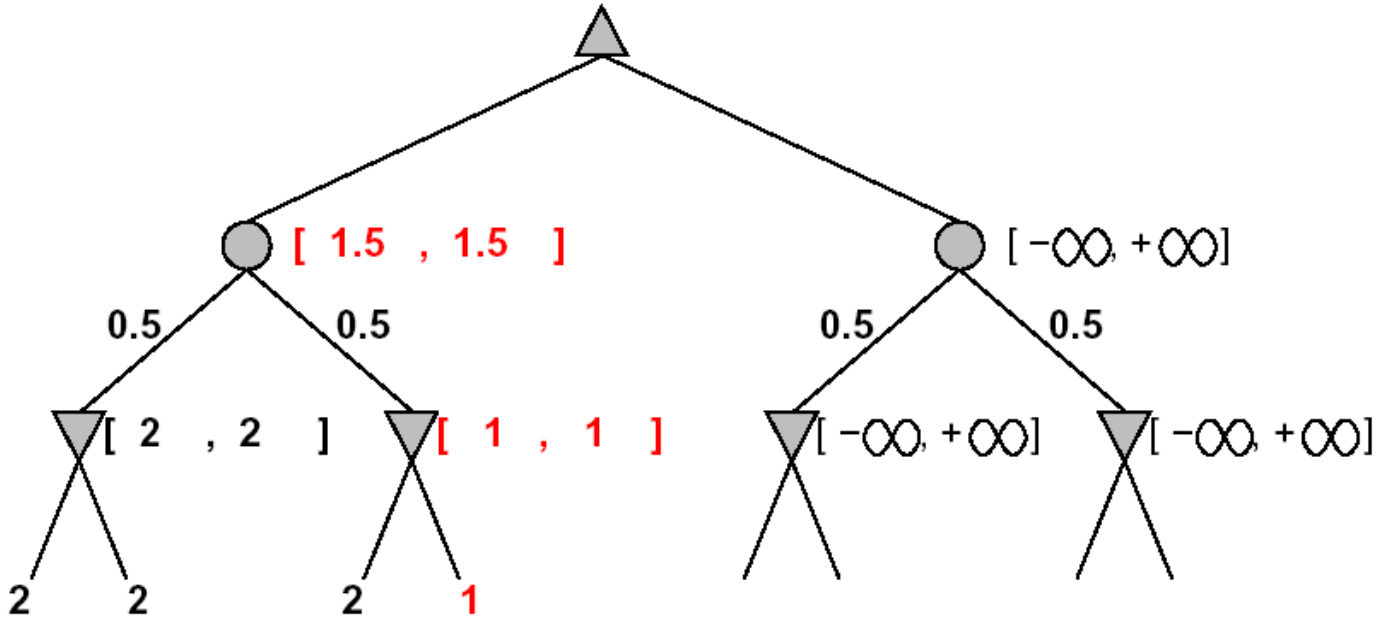
Pruning in nondeterministic game trees

- A version of α–β pruning is possible:



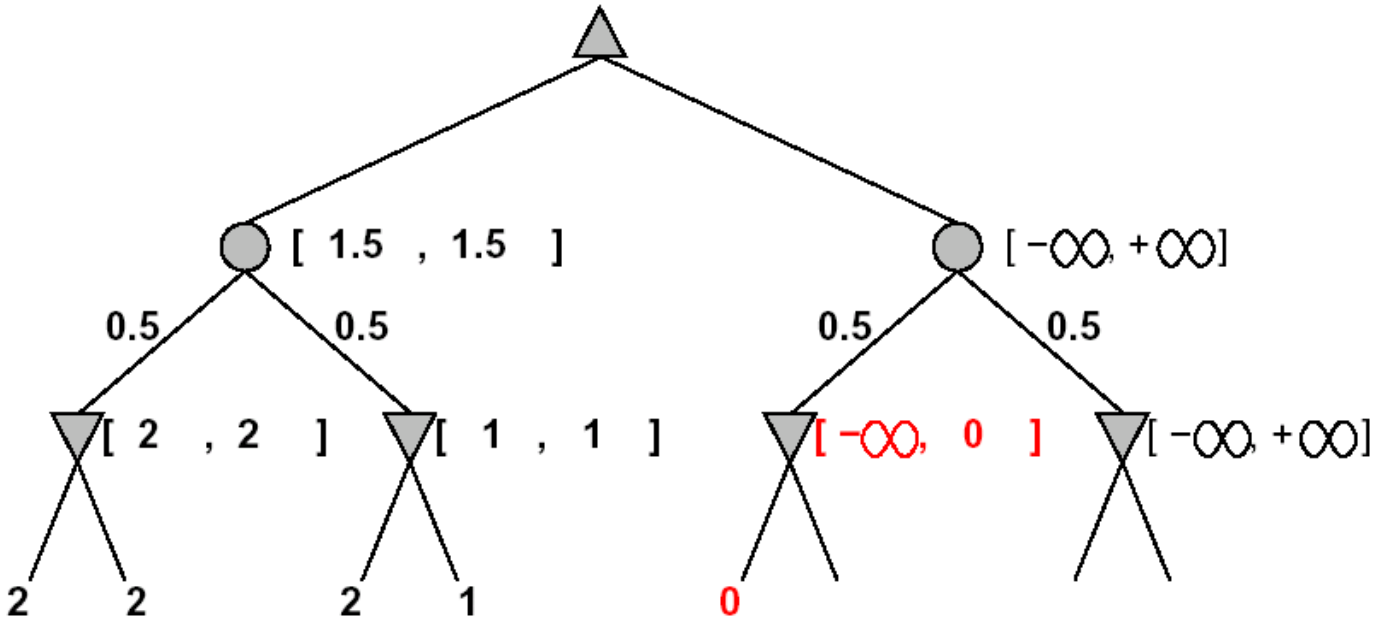
Pruning in nondeterministic game trees

- A version of α–β pruning is possible:



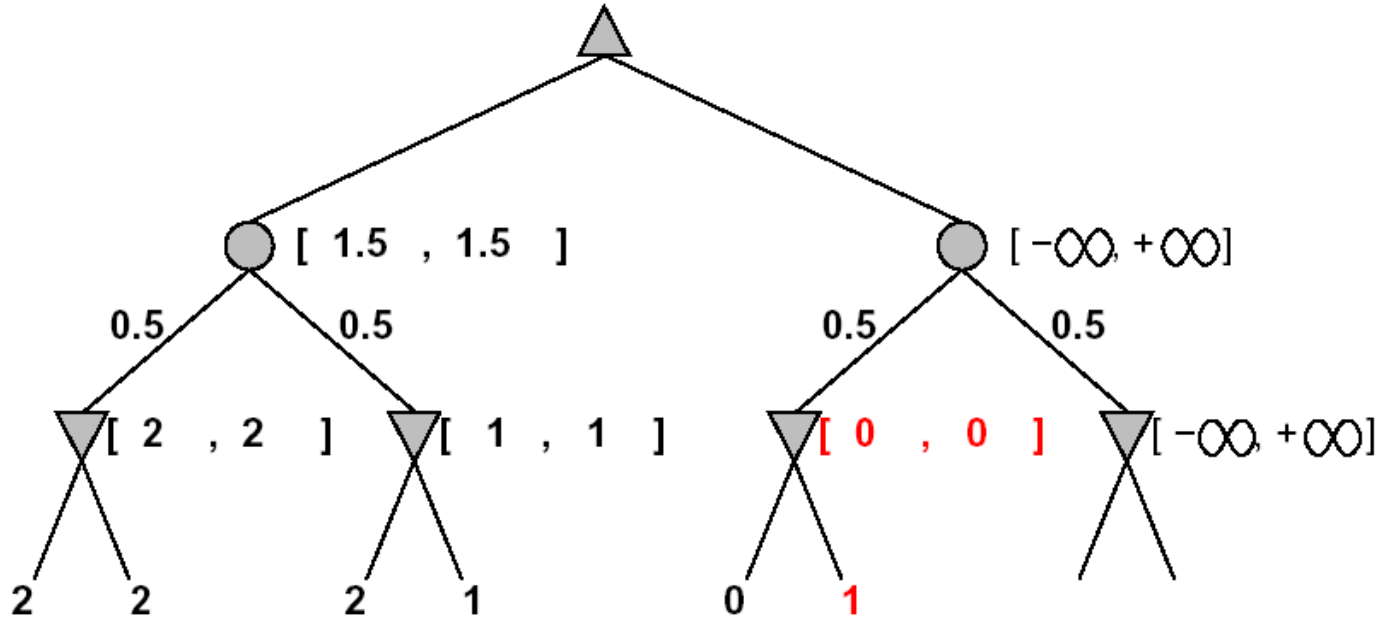
Pruning in nondeterministic game trees

- A version of α–β pruning is possible:



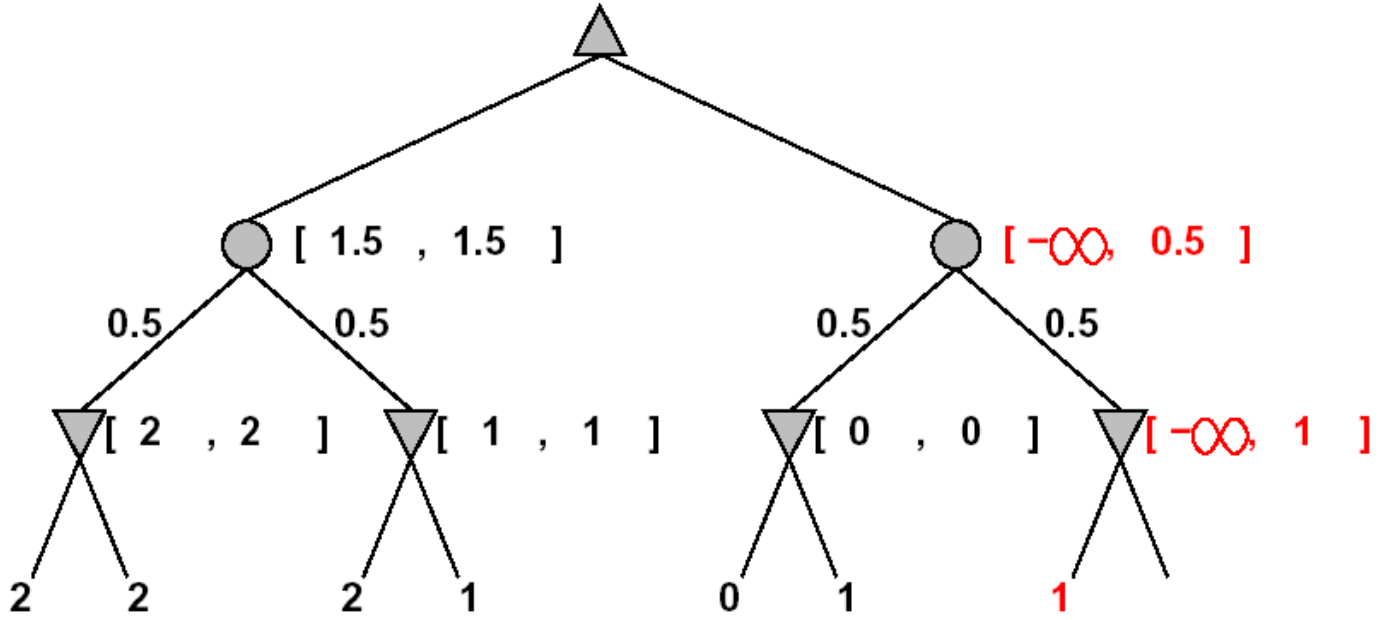
Pruning in nondeterministic game trees

- A version of α–β pruning is possible:



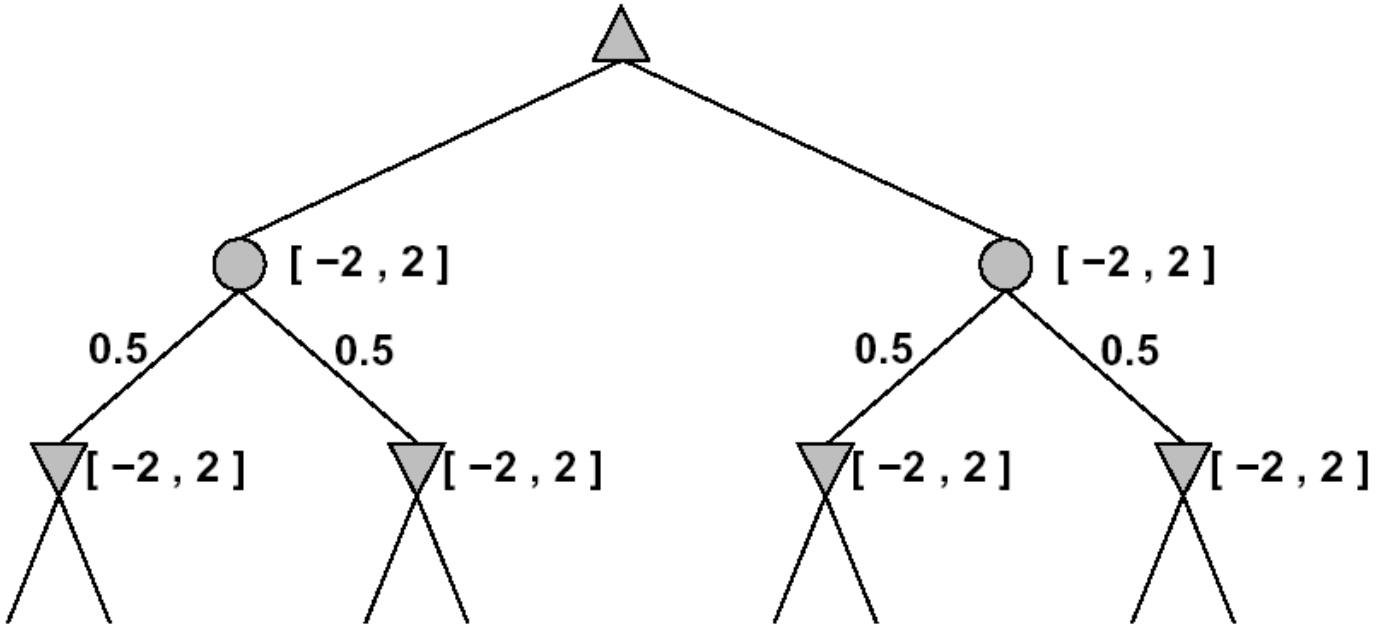
Pruning in nondeterministic game trees

- A version of α–β pruning is possible:



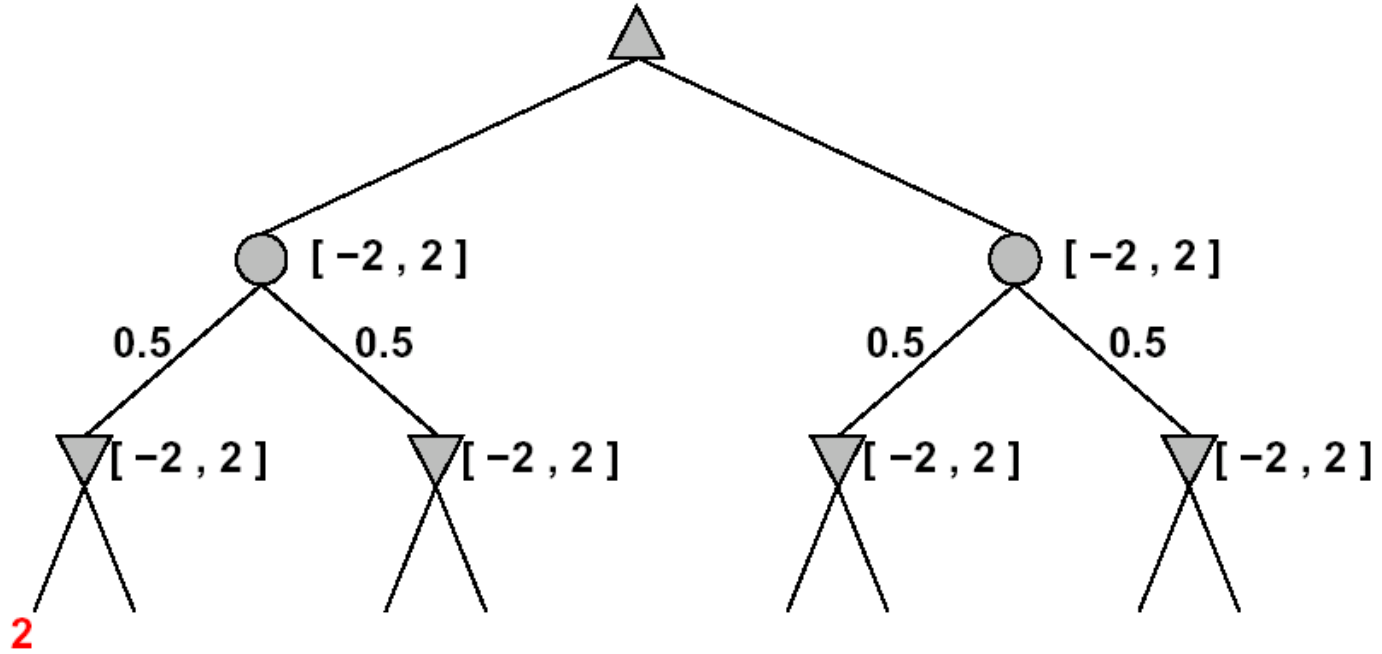
Pruning (contd.)

- More pruning occurs if we can bound the leaf values



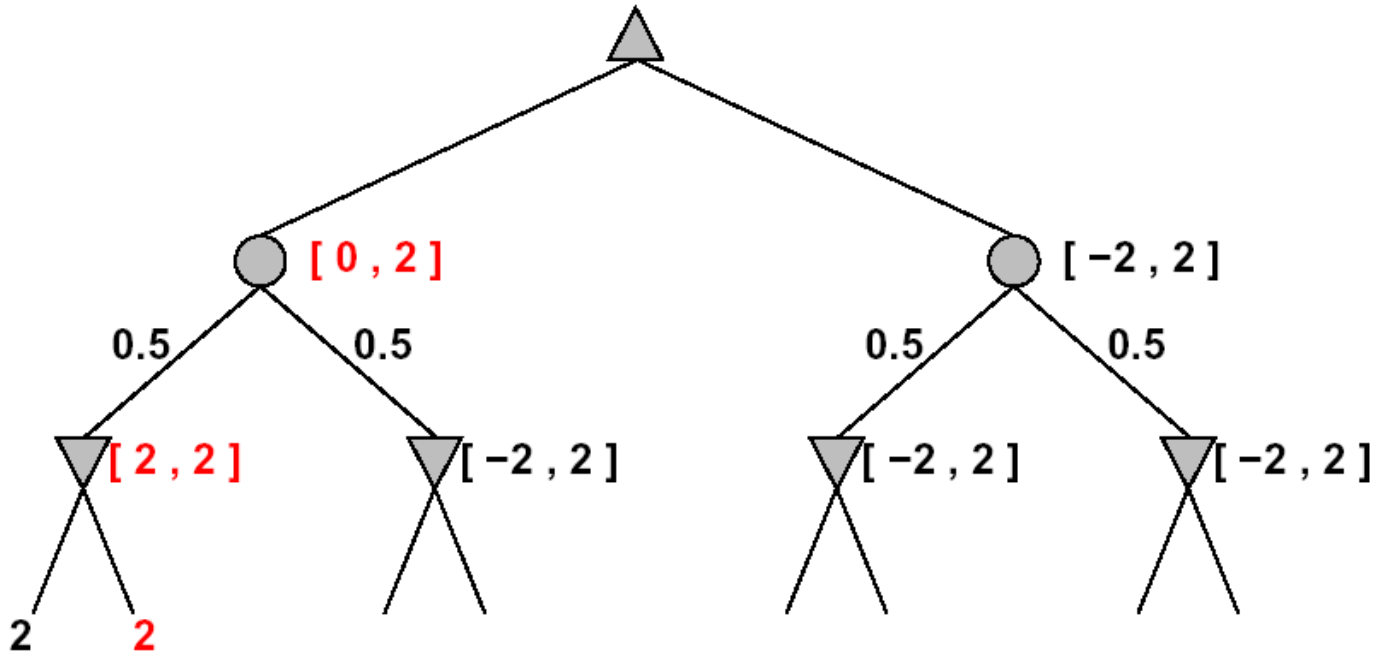
Pruning (contd.)

- More pruning occurs if we can bound the leaf values



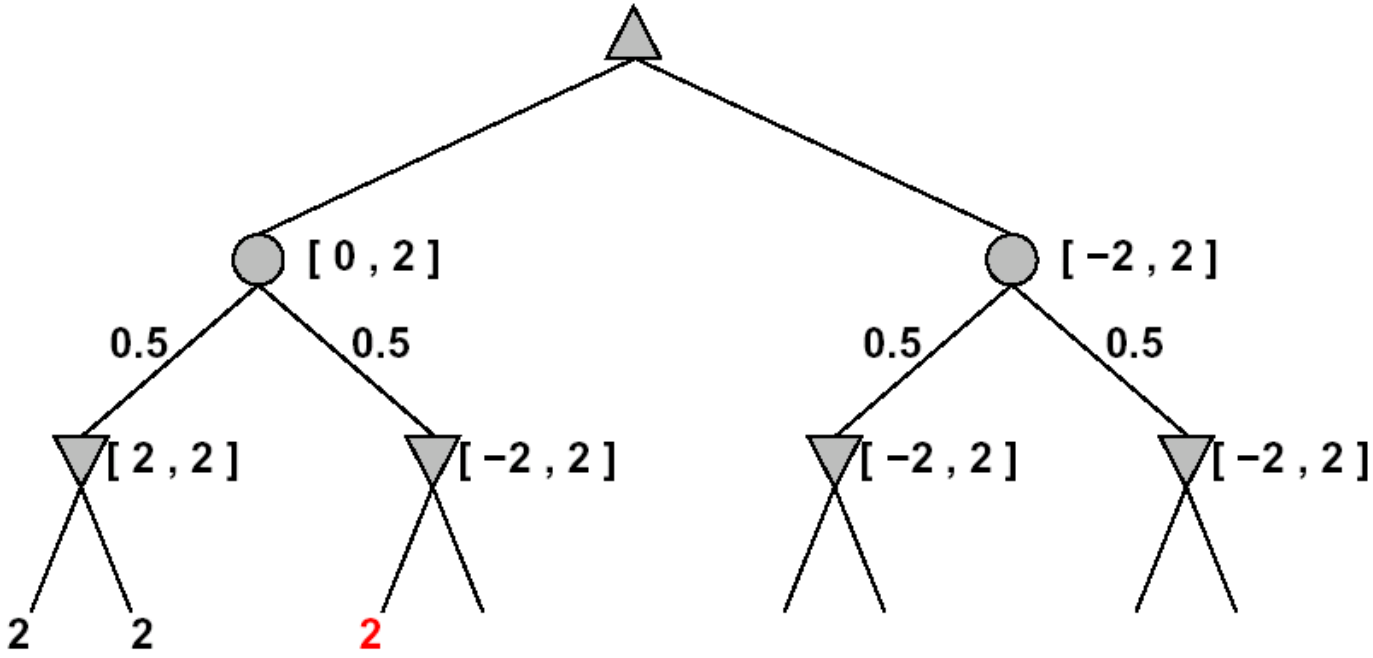
Pruning (contd.)

- More pruning occurs if we can bound the leaf values



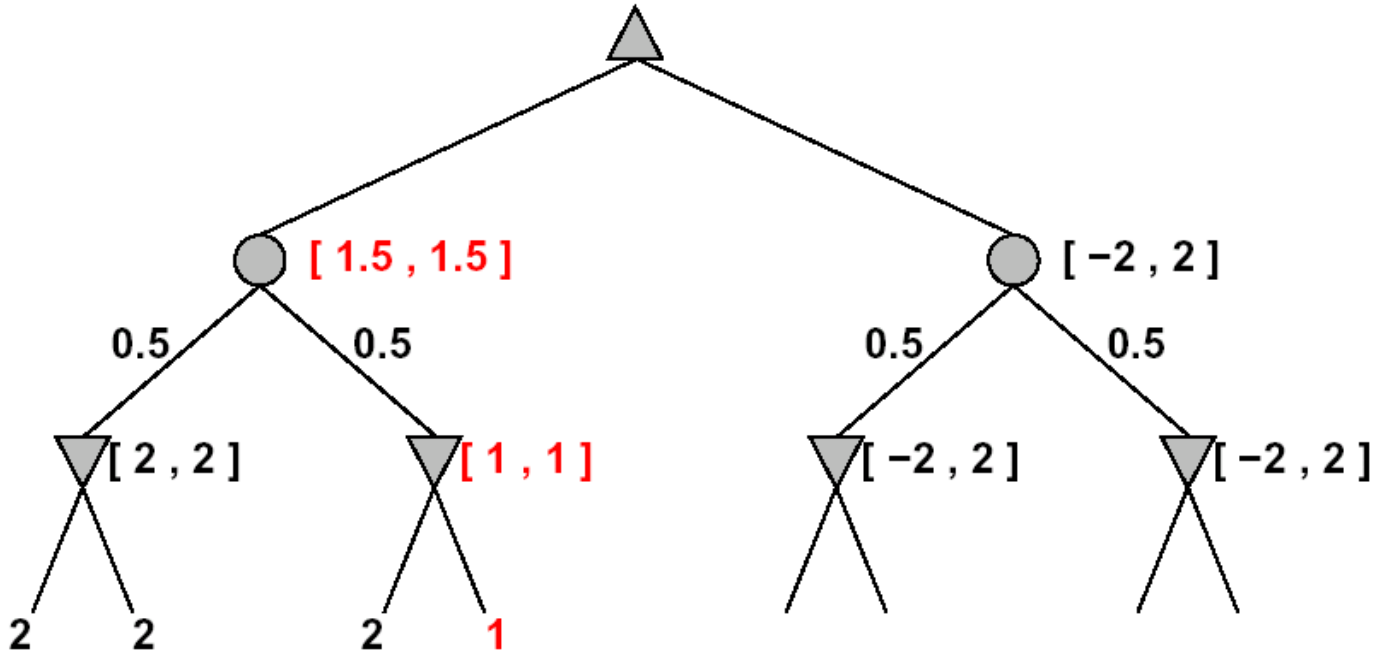
Pruning (contd.)

- More pruning occurs if we can bound the leaf values



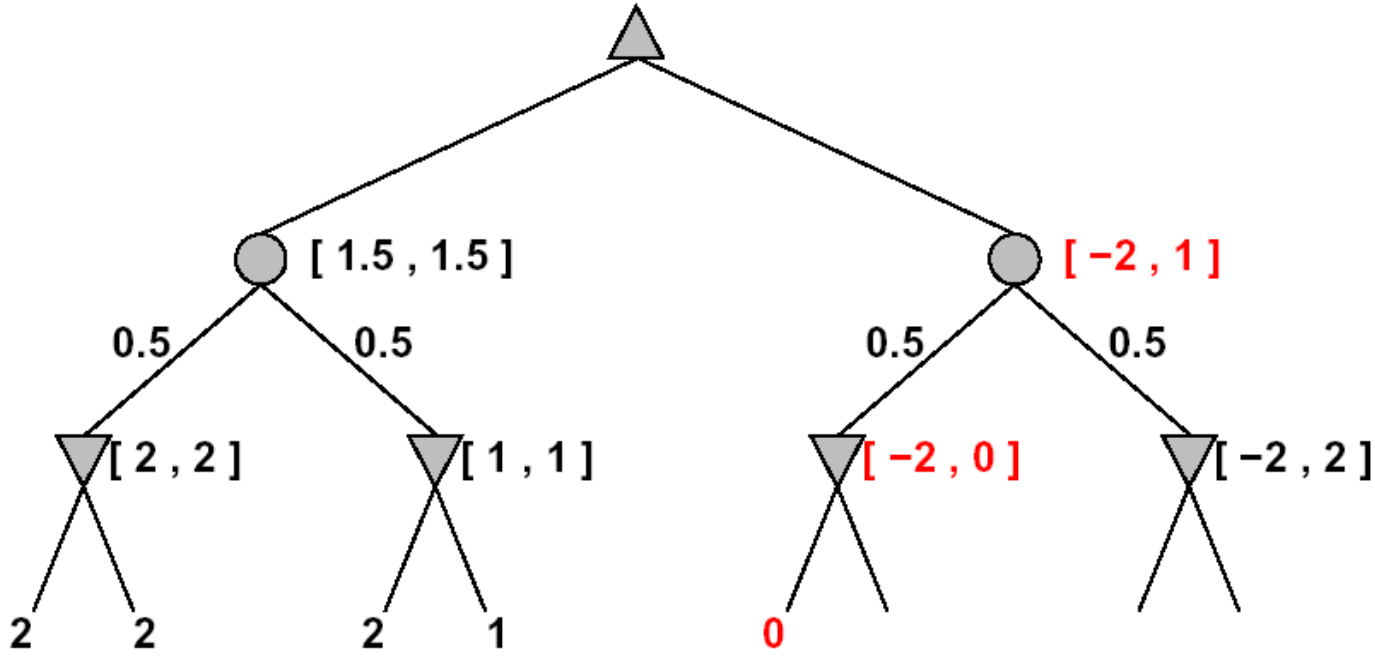
Pruning (contd.)

- More pruning occurs if we can bound the leaf values



Pruning (contd.)

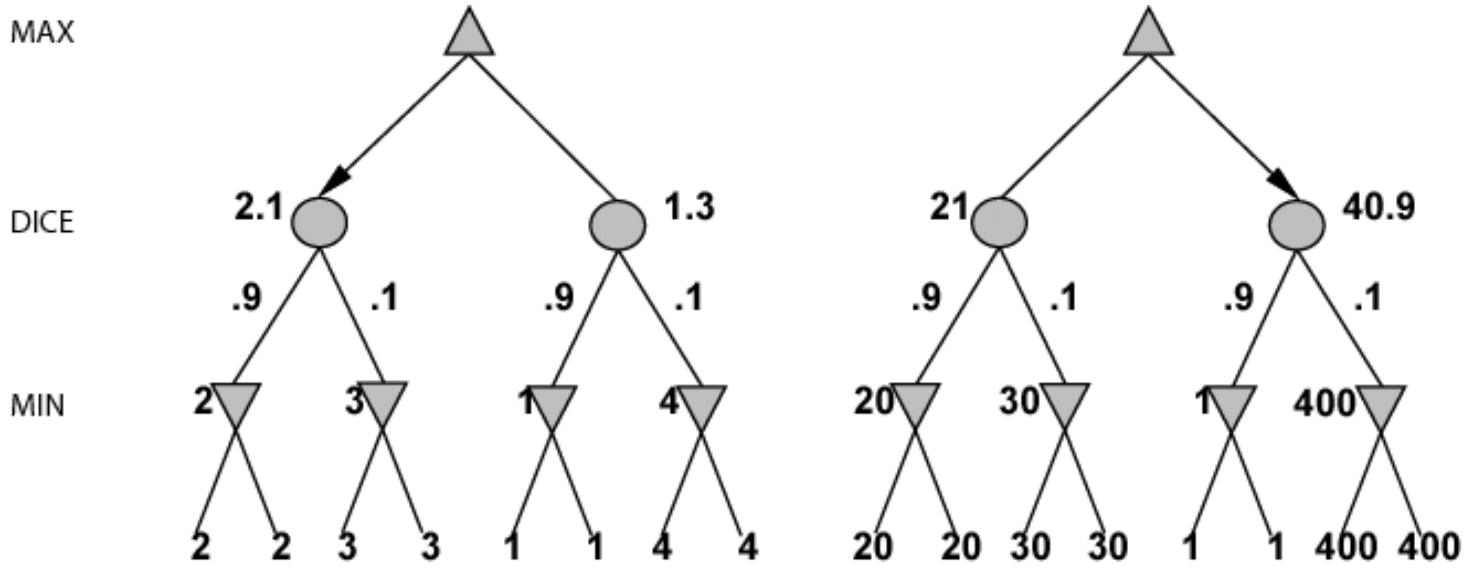
- More pruning occurs if we can bound the leaf values



Nondeterministic games in practice

- Dice rolls increase *b*: 21 possible rolls with 2 dice
- Backgammon ≈ 20 legal moves (can be 6,000 with 1-1 roll)
  depth 4 = 20 $(21 \times 20)^3 \approx 1.2 \times 10^9$
- As depth increases, probability of reaching a given node shrinks
  $\Rightarrow$ value of lookahead is diminished
- α–β pruning is much less effective
- **`TDGammon`** uses depth-2 search + very good **`Eval`**
  ≈ world-champion level

Digression:
Exact values DO matter



- Behaviour is preserved only by *positive linear* transformation ofc **`Eval`**
- Hence **`Eval`** should be proportional to the expected payoff

Games of imperfect information

- E.g., card games, where opponent's initial cards are unknown
- Typically we can calculate a probability for each possible deal
- Seems just like having one big dice roll at the beginning of the game*

Idea: compute the minimax value of each action in each deal,
then choose the action with highest expected value over all deals*

- Special case: if an action is optimal for all deals, it's optimal.*
- GIB, current best bridge program, approximates this idea by

1) generating 100 deals consistent with bidding information
2) picking the action that wins most tricks on average

•https://www.youtube.com/watch?v=jLXPGwJNLHk

Summary

- Games are fun to work on!
  (and dangerous)
- They illustrate several important points about AI
  - perfection is unattainable $\Rightarrow$ must approximate
  - good idea to think about what to think about
  - uncertainty constrains the assignment of values to states
- Games are to AI as grand prix racing is to automobile design