

CSEP 573

## Chapters 3-5

### Problem Solving using Search

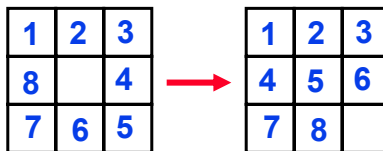


“First, they do an on-line search”

© The New Yorker collection. All rights reserved.  
From The New Yorker Book of Technology Cartoons.

© CSE AI Faculty

### Example: The 8-puzzle

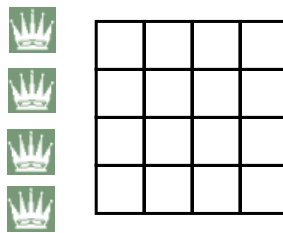


## Example: Route Planning



3

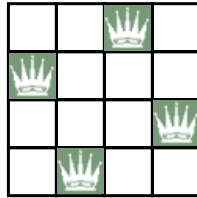
## Example: N Queens



4 Queens

4

## Example: N Queens



4 Queens

5

## State-Space Search Problems

**General problem:**

Given a *start state*, find a path to a *goal state*

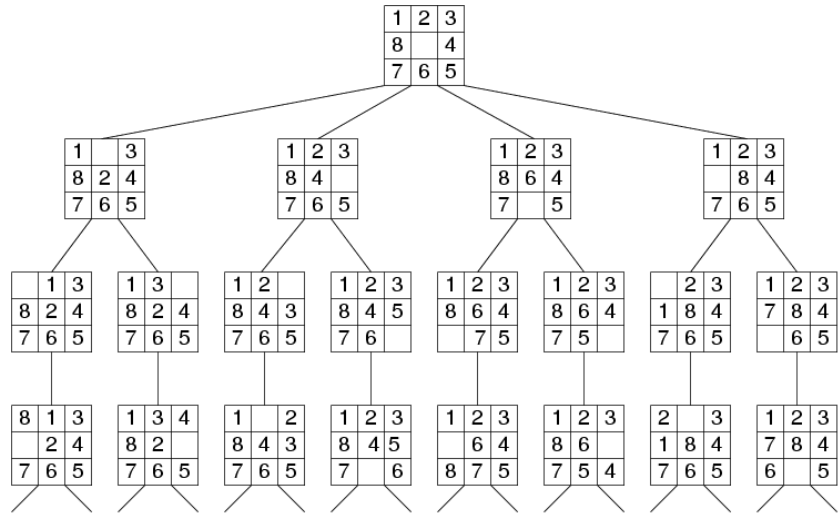
- Can test if a state is a goal
- Given a state, can generate its *successor* states

**Variants:**

- Find any path *vs.* a least-cost path
- Goal is completely specified, task is just to find the path
  - Route planning
- Path doesn't matter, only finding the goal state
  - 8 puzzle, N queens

6

## Tree Representation of 8-Puzzle Problem Space



7

### 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)
  
```

*fringe* (= *frontier* in the textbook) is the set of all leaf nodes available for expansion

8

## 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
  
```

9

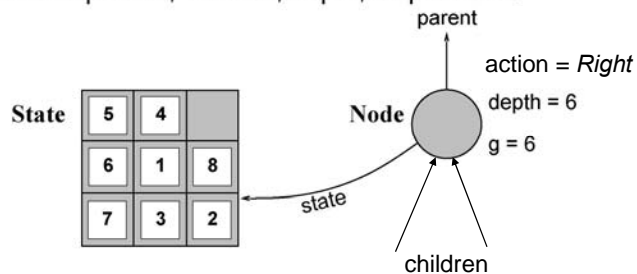
## Implementation: states vs. nodes

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!



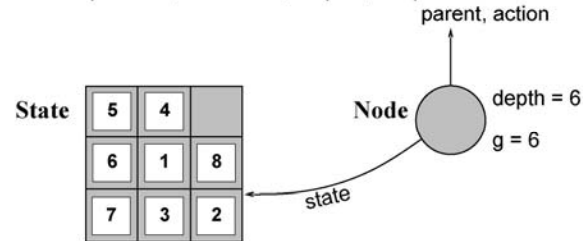
10

## Implementation: states vs. 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.

11

## 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?

*time 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  $\infty$ )

## 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

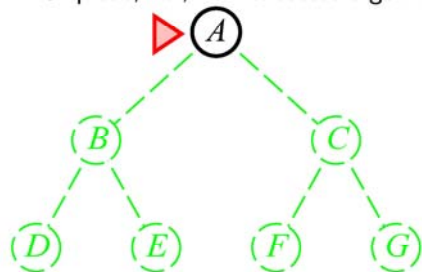
Iterative deepening search

## Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end

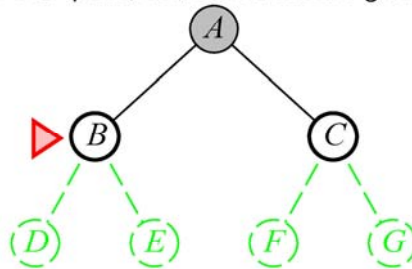


## Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end



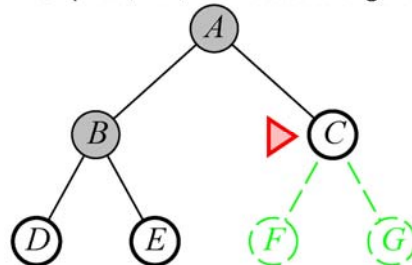
15

## Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end



16

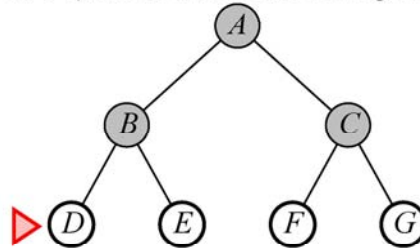


## Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end



17

## Properties of breadth-first search

Complete??

18

## Properties of breadth-first search

Complete?? Yes (if  $b$  is finite)

Time??

19

## Properties of breadth-first search

Complete?? Yes (if  $b$  is finite)

Time??  $b + b^2 + b^3 + \dots + b^d = O(b^d)$ , i.e., exponential in  $d$

Space??

20

## Properties of breadth-first search

Complete?? Yes (if  $b$  is finite)

Time??  $b + b^2 + b^3 + \dots + b^d = O(b^d)$ , i.e., exponential in  $d$

Space??  $O(b^d)$  (keeps every node in memory)

Optimal??

21

## Properties of breadth-first search

Complete?? Yes (if  $b$  is finite)

Time??  $b + b^2 + b^3 + \dots + b^d = O(b^d)$ , i.e., exponential in  $d$

Space??  $O(b^d)$  (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

**Space is the big problem for BFS.**

**Example:  $b = 10$ , 10,000 nodes/sec, 1KB/node**

**$d = 3 \rightarrow 1000$  nodes, 0.1 sec, 1MB**

**$d = 5 \rightarrow 100,000$  nodes, 10 secs, 100 MB**

**$d = 9 \rightarrow 10^9$  nodes, 31 hours, 1 TB**

22

## Uniform-cost search

Expand least-cost unexpanded node (used when step costs are unequal)

Implementation:

*fringe* = queue ordered by path cost (use priority queue)

Equivalent to breadth-first if step costs all equal

**Complete??** Yes, if step cost  $\geq \epsilon$  (small positive constant; 0 cost may cause infinite loop)

**Time??** # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$   
where  $C^*$  is the cost of the optimal solution

**Space??** # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$

**Optimal??** Yes—nodes expanded in increasing order of  $g(n)$

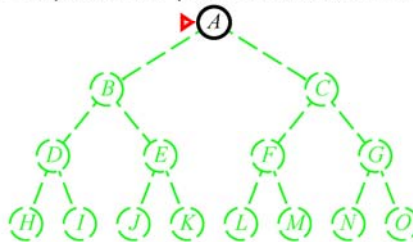
23

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



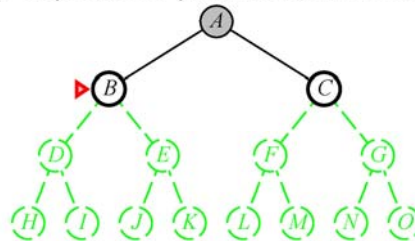
24

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



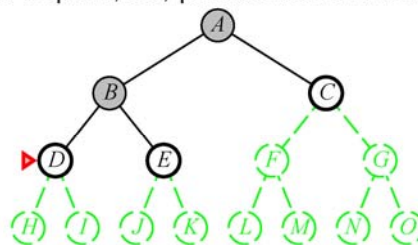
25

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



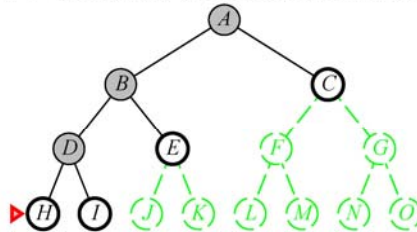
26

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



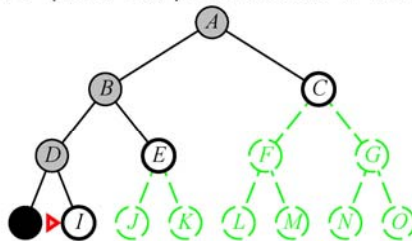
27

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



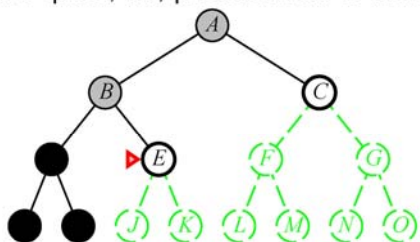
28

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



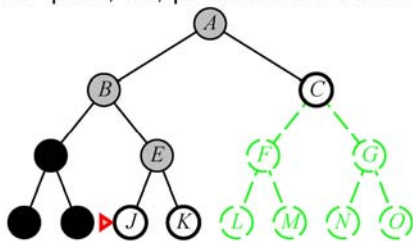
29

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



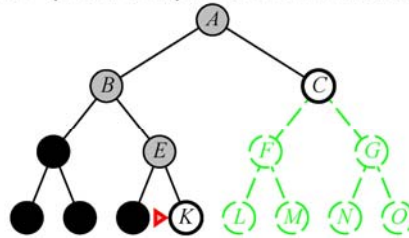
30

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



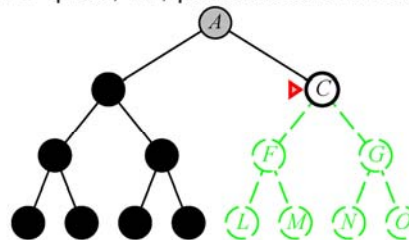
31

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



32

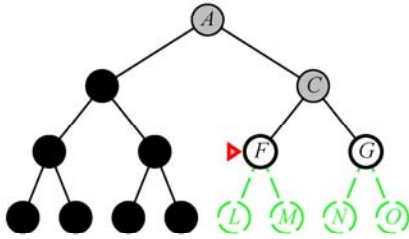


## Depth-first search

Expand deepest unexpanded node

Implementation:

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



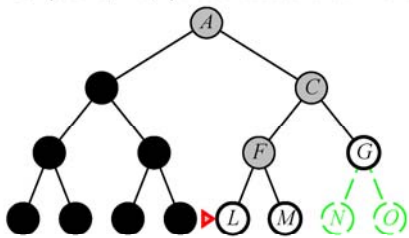
33

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



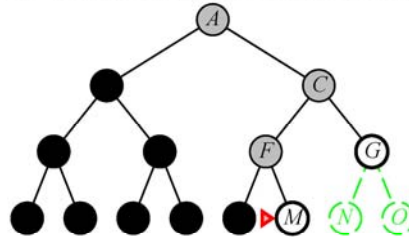
34

## Depth-first search

Expand deepest unexpanded node

Implementation:

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



35

## Properties of depth-first search

Complete??

36

## Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops  
Modify to avoid repeated states along path ("GRAPH-SEARCH" in textbook)  
⇒ complete in finite spaces

Time??

37

## Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops  
Modify to avoid repeated states along path ("GRAPH-SEARCH" in textbook)  
⇒ complete in finite spaces

Time??  $O(b^m)$ : terrible if  $m$  is much larger than  $d$  ( $m = \text{maximum depth}$ )  
but if solutions are dense, may be much faster than breadth-first

Space??

38

## Properties of depth-first search

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??

39

## Properties of depth-first search

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 (may find a solution but least cost solution  
may be on a different branch)

40

## Depth-limited search

= depth-first search with depth limit  $l$ ,  
i.e., nodes at depth  $l$  have no successors

Recursive implementation:

```
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
```

41

## Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
  inputs: problem, a problem
  for depth ← 0 to  $\infty$  do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
  end
```

42

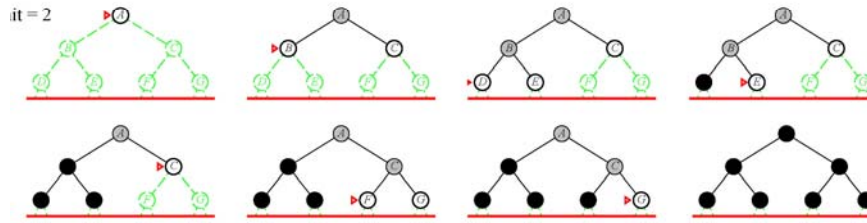
Iterative deepening search  $l = 0$



Iterative deepening search  $l = 1$

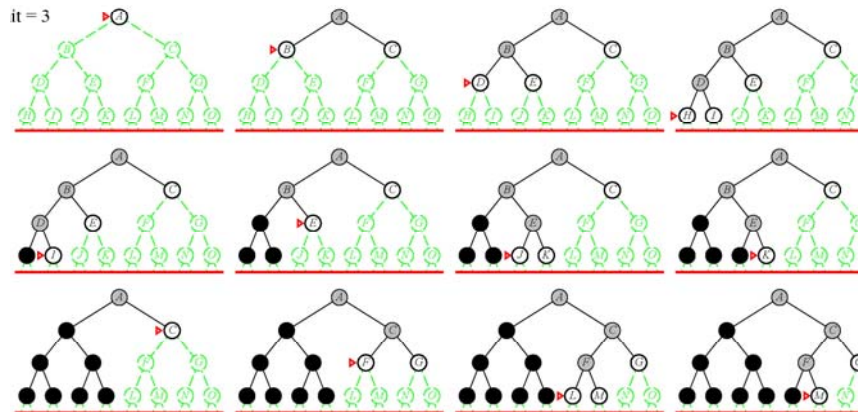


### Iterative deepening search $l = 2$



45

### Iterative deepening search $l = 3$



46

## Properties of iterative deepening search

Complete??

47

## Properties of iterative deepening search

Complete?? Yes

Time??

48



## Properties of iterative deepening search

Complete?? Yes

Time??  $db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$

Space??

49

## Properties of iterative deepening search

Complete?? Yes

Time??  $db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$

Space??  $O(bd)$

Optimal??

50

## Properties of iterative deepening search

**Complete??** Yes

**Time??**  $db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$

**Space??**  $O(bd)$

**Optimal??** Yes, if step cost = 1

Can be modified to explore uniform-cost tree

**Increasing path-cost limits instead of depth limits**

**This is called Iterative lengthening search (exercise 3.17)**

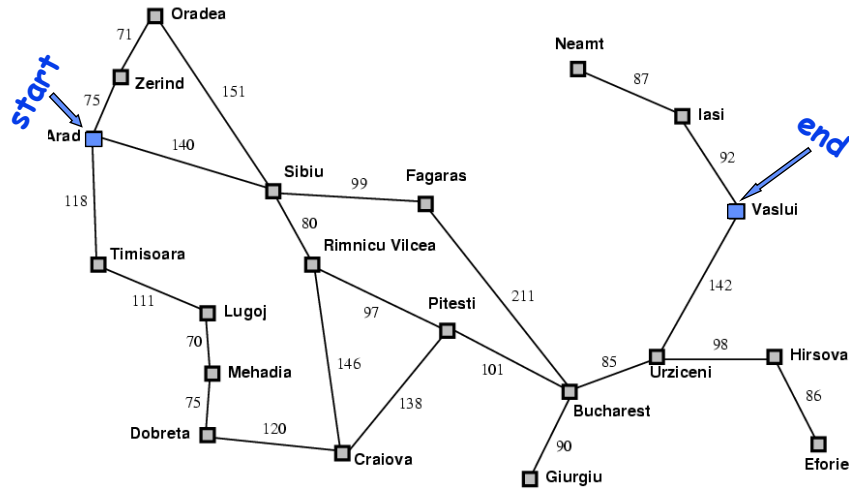
51

## 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	$b^d$	$b^{\lceil C^*/\epsilon \rceil}$	$b^m$	$b^l$	$b^d$
Space	$b^d$	$b^{\lceil C^*/\epsilon \rceil}$	$bm$	$bl$	$bd$
Optimal?	Yes*	Yes*	No	No	Yes

52

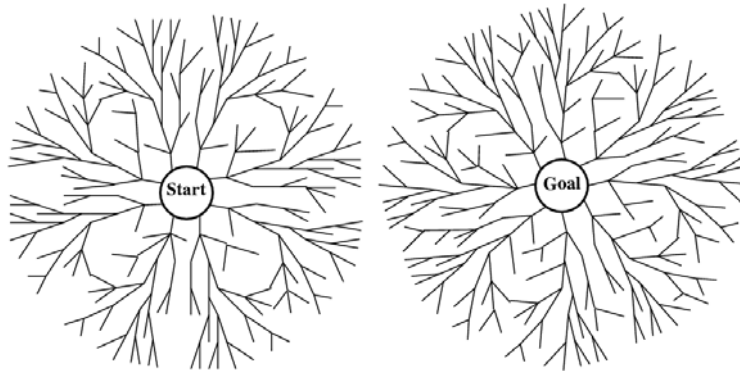
## Forwards vs. Backwards



Problem: Find the shortest route

53

## Bidirectional Search



Motivation:  $b^{d/2} + b^{d/2} \ll b^d$

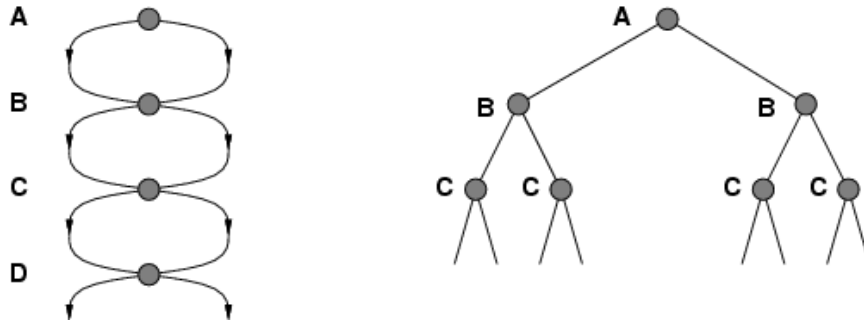
Can use breadth-first search or uniform-cost search

Hard for implicit goals e.g., goal = "checkmate" in chess

54

## Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one! (e.g., repeated states in 8 puzzle)



**Graph search algorithm:** Store expanded nodes in a set called *closed* (or *explored*) and only add new nodes to the fringe

55

## 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 SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

56

All these methods are slow (blind)

Can we do better?

57

## Informed Search

Use problem-specific knowledge to guide search (use “**heuristic function**”)



58

## **Best-first Search**

**Generalization of breadth first search**

**Priority queue of nodes to be explored**

**Evaluation function  $f(n)$  used for each node**

**Insert initial state into priority queue**

**While queue not empty**

**Node = head(queue)**

**If goal(node) then return node**

**Insert children of node into pr. queue**

59

## **Who's on (best) first?**

**Breadth first search is special case of best first**

- with  $f(n) = \text{depth}(n)$

**Dijkstra's Algorithm is best first**

- with  $f(n) = g(n)$

**where  $g(n)$  = sum of edge costs from start to n**

60

## Greedy best-first search

Evaluation function  $f(n) = h(n)$  (**h**euristic) = estimate of cost from  $n$  to goal

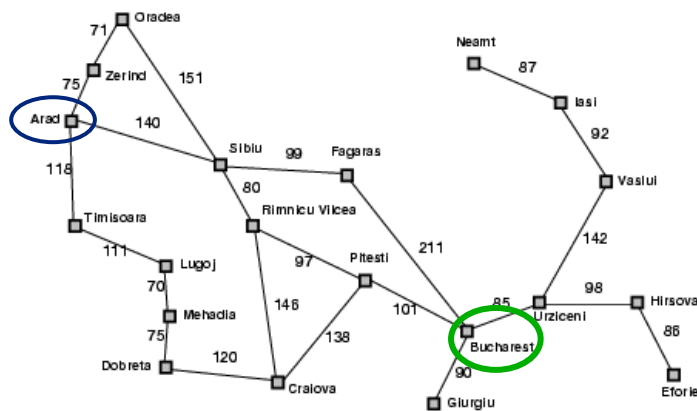
e.g., Route finding problems:  $h_{SLD}(n)$  = straight-line distance from  $n$  to destination

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

61

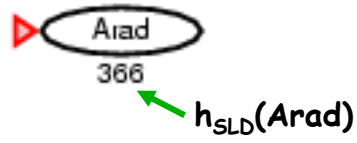
## Example: Lost in Romania

Need: Shortest path from Arad to Bucharest



62

**Example: Greedily Searching for Bucharest**



63

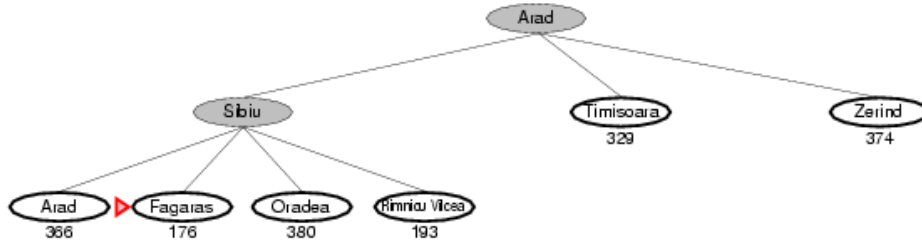
**Example: Greedily Searching for Bucharest**



64

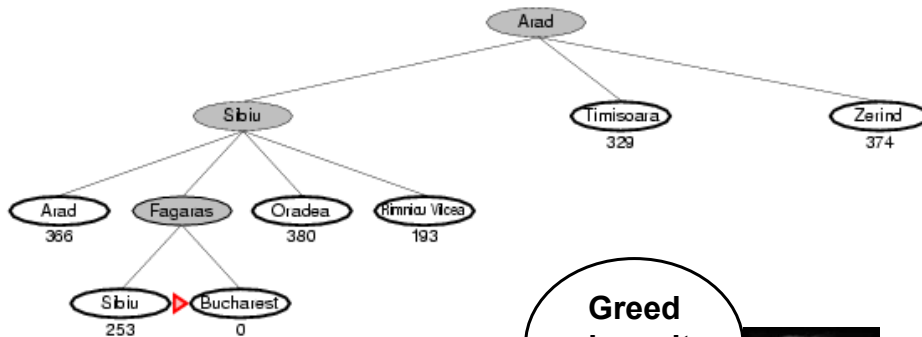


### Example: Greedily Searching for Bucharest



65

### Example: Greedily Searching for Bucharest



Greed doesn't pay!



Not optimal!  
Arad, Sibiu, Rimnicu Vilcea, Pitesti, Bucharest shorter

66

## Properties of Greedy Best-First Search

Complete? No – can get stuck in loops (unless *closed* list is used)

Time?  $O(b^m)$ , but a good heuristic can give dramatic improvement

Space?  $O(b^m)$  -- keeps all nodes in memory *a la* breadth first search

Optimal? No, as our example illustrated

67

## A\* Search

(Hart, Nilsson & Rafael 1968)

- Best first search with  $f(n) = g(n) + h(n)$

$g(n)$  = sum of edge costs from start to  $n$

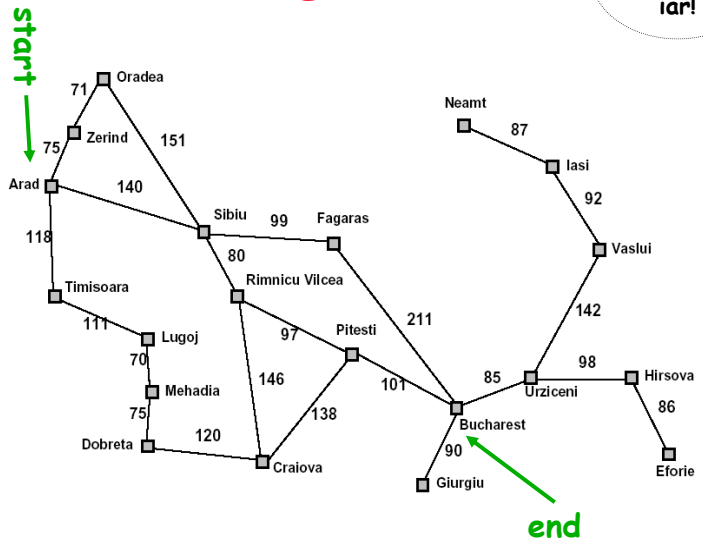
$h(n)$  = **heuristic function** = estimate of lowest cost path from  $n$  to goal

- If  $h(n)$  is “admissible” then search will be optimal

↑ { Underestimates cost of any solution which can be reached from node

68

# Back in Romania Again

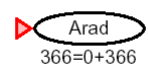


Straight-line distance to Bucharest

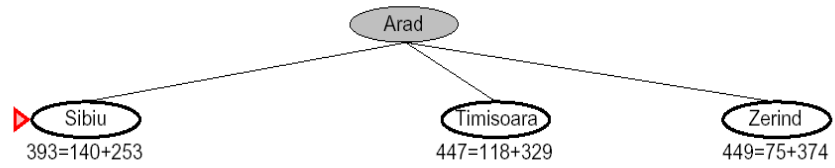
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

# A\* Example for Romania

$f(n) = g(n) + h(n)$  where  
 $g(n)$  = sum of edge costs from start to  $n$   
 $h(n) = h_{SLD}(n)$  = straight-line distance from  $n$  to destination

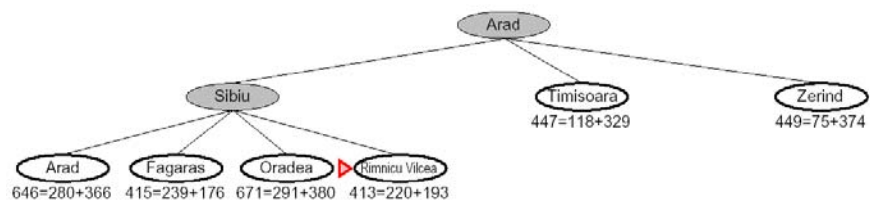


## A\* Example



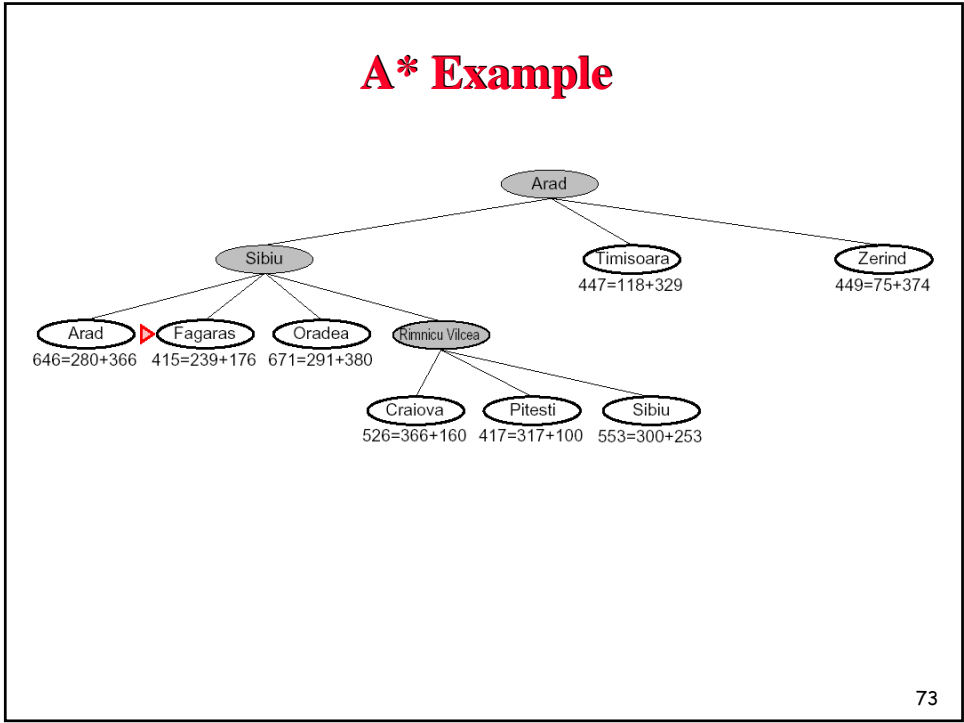
71

## A\* Example



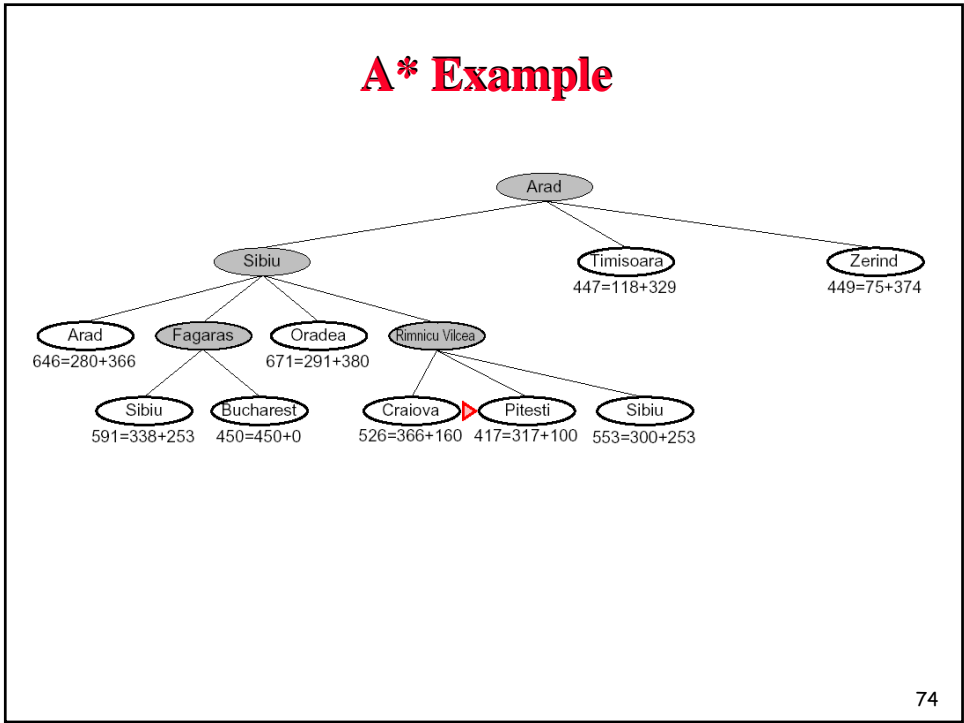
72

## A\* Example



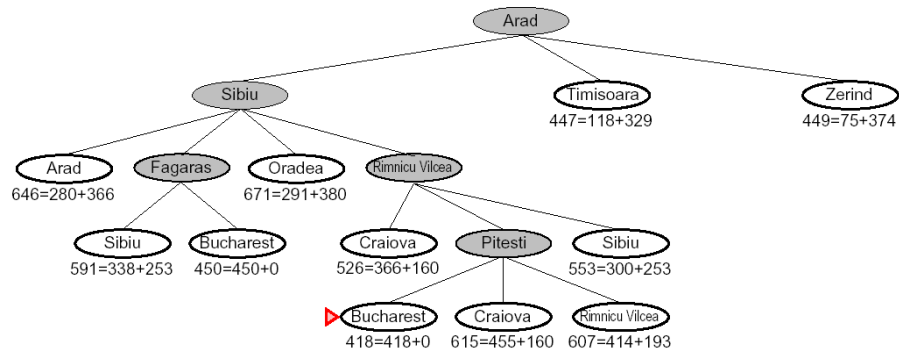
73

## A\* Example



74

## A\* Example



75

## Admissible heuristics

A heuristic  $h(n)$  is **admissible** if  
for every node  $n$ ,

$$h(n) \leq h^*(n)$$

where  $h^*(n)$  is the **true** cost to reach the goal state from  $n$ .

An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**

76

## Admissible Heuristics

Is the Straight Line Distance heuristic  $h_{SLD}(n)$  admissible?

77

## Admissible Heuristics

Is the Straight Line Distance heuristic  $h_{SLD}(n)$  admissible?

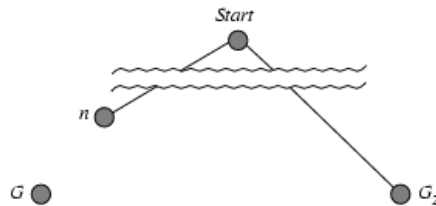
Yes, it never overestimates the actual road distance

**Theorem:** If  $h(n)$  is admissible, A\* using TREE-SEARCH is optimal.

78

## Optimality of A\* (proof)

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

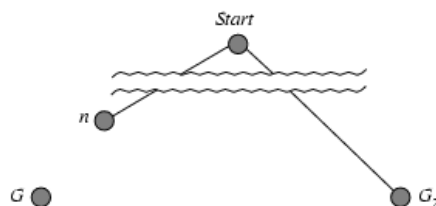


$$\begin{array}{ll}
 f(G_2) = g(G_2) & \text{since } h(G_2) = 0 \\
 > g(G) & \text{since } G_2 \text{ is suboptimal} \\
 f(G) = g(G) & \text{since } h(G) = 0 \\
 f(G_2) > f(G) & \text{from above}
 \end{array}$$

79

## Optimality of A\* (cont.)

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



$$\begin{array}{ll}
 f(G_2) > f(G) & \text{from prev slide} \\
 h(n) \leq h^*(n) & \text{since } h \text{ is admissible} \\
 g(n) + h(n) \leq g(n) + h^*(n) \\
 f(n) \leq f(G) < f(G_2)
 \end{array}$$

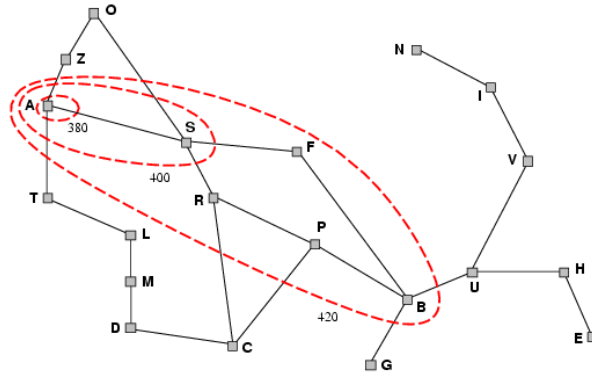
Hence  $f(n) < f(G_2) \Rightarrow A^*$  will never select  $G_2$  for expansion.

80



## Optimality of A\*

A\* expands nodes in order of increasing  $f$  value  
Gradually adds " $f$ -contours" of nodes



81

Okay, proof is done!  
Time to wake up...



82

## Properties of A\*

**Complete?** Yes (unless there are infinitely many nodes with  $f \leq f(G)$ )

**Time?** Exponential (for most heuristic functions in practice)

**Space?** Keeps all generated nodes in memory (exponential number of nodes)

**Optimal?** Yes

83

## Admissible heuristics

E.g., for the 8-puzzle, what are some admissible heuristic functions? (for # steps to goal state)

$h_1(n) = ?$

$h_2(n) = ?$

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

84

## Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$  = number of misplaced tiles

$h_2(n)$  = total Manhattan distance (no. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

S

$h_1(S) = ?$

$h_2(S) = ?$

85

## Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$  = number of misplaced tiles

$h_2(n)$  = total Manhattan distance (no. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

$h_1(S) = ?$  8

$h_2(S) = ?$   $3+1+2+2+2+3+3+2 = 18$

86

## Dominance

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

87

## Dominance

E.g., for 8-puzzle heuristics  $h_1$  and  $h_2$ , typical search costs (average number of nodes expanded for solution depth  $d$ ):

$d=12$  IDS = 3,644,035 nodes  
 $A^*(h_1) = 227$  nodes  
 $A^*(h_2) = 73$  nodes

$d=24$  IDS = too many nodes  
 $A^*(h_1) = 39,135$  nodes  
 $A^*(h_2) = 1,641$  nodes

88

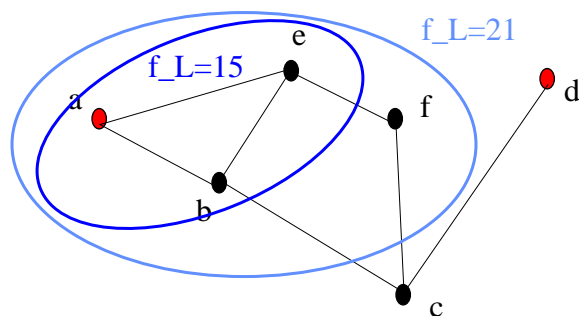
**In general, A\* not practical for large scale problems due to memory requirements (all generated nodes in memory)**

Idea: Use iterative deepening

### **Iterative-Deepening A\***

Like iterative-deepening search, but cutoff is f cost (= g + h) rather than depth

At each iteration, cutoff is smallest f cost among nodes that exceeded cutoff on prev iteration

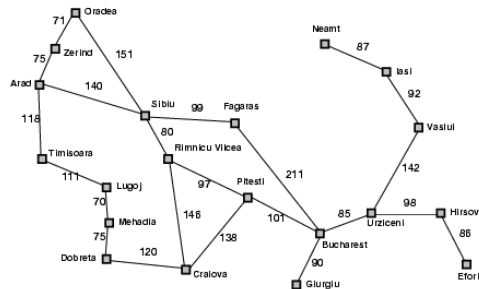


## Back to Admissible Heuristics

$$f(x) = g(x) + h(x)$$

**g:** cost so far

**h:** underestimate of remaining costs



e.g.,  $h_{SLD}$

Where do heuristics come from?

91

## Relaxed Problems

Derive admissible heuristic from **exact** cost of a solution to a **relaxed** version of problem

- *For route planning, what is a relaxed problem?*

Relax requirement that car stay on road →  
Straight Line Distance becomes optimal cost

Cost of optimal solution to relaxed problem  $\leq$   
cost of optimal solution for real problem

92

## Heuristics for eight puzzle

7	2	3
5	1	6
8	4	■

 → 

1	2	3
4	5	6
7	8	■

start

goal

What can we relax?

Original Problem: Tile can move from location A to B if  
A is horizontally or vertically next to B *and* B is blank

93

## Heuristics for eight puzzle

7	2	3
5	1	6
8	4	■

 → 

1	2	3
4	5	6
7	8	■

Relaxed 1: Tile can move from any location A to any location B

Cost =  $h_1$  = number of misplaced tiles

Relaxed 2: Tile can move from A to B if A is horizontally or  
vertically next to B (*note*: B does not have to be blank)

Cost =  $h_2$  = total Manhattan distance

You can try other possible heuristics in your HW #1

94

## Need for Better Heuristics

### Performance of $h_2$ (Manhattan Distance Heuristic)

- 8 Puzzle < 1 second
- 15 Puzzle 1 minute
- 24 Puzzle 65000 years

Can we do better?

Adapted from Richard Korf presentation 95

## Creating New Heuristics

Given admissible heuristics  $h_1, h_2, \dots, h_m$ , none of them dominating any other, how to choose the best?

Answer: No need to choose only one! Use:

$$h(n) = \max \{h_1(n), h_2(n), \dots, h_n(n)\}$$

$h$  is admissible (why?)

$h$  dominates all  $h_i$  (by construction)

Can we do better with:

$$h'(n) = h_1(n) + h_2(n) + \dots + h_n(n)?$$

96



## Pattern Databases

**Idea:** Use solution cost of a subproblem as heuristic. For 8-puzzle: pick any subset of tiles

E.g., 3, 7, 11, 12

Precompute a table

- Compute optimal cost of solving just these tiles
  - This is a lower bound on actual cost with all tiles
- For all possible configurations of these tiles
  - Could be several million
- Use breadth first search back from goal state
  - State = position of just these tiles (& blank)
- Admissible heuristic  $h_{DB}$  for complete state = cost of corresponding sub-problem state in database

Adapted from Richard Korf presentation 97

## Combining Multiple Databases

Can choose another set of tiles

- Precompute multiple tables

How to combine table values?

- Use the *max* trick!

E.g. Optimal solutions to Rubik's cube

- First found w/ IDA\* using pattern DB heuristics
- Multiple DBs were used (diff subsets of cubies)
- Most problems solved optimally in 1 day
- Compare with *574,000 years* for IDS

Adapted from Richard Korf presentation 98

## Drawbacks of Standard Pattern DBs

Since we can only take *max*

- Diminishing returns on additional DBs

Would like to be able to *add* values

- But not exceed the actual solution cost (to ensure admissible heuristic)
- How?

Adapted from Richard Korf presentation

99

## Disjoint Pattern DBs

Partition tiles into disjoint sets

- For each set, precompute table
- Don't count moves of tiles not in set
  - This makes sure costs are disjoint
  - Can be added without overestimating!
  - E.g. For 15 puzzle shown, 8 tile DB has 519 million entries
  - And 7 tile DB has 58 million

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	■

During search

- Look up costs for each set in DB
- *Add values to get heuristic function value*
  
- Manhattan distance is a special case of this idea where each set is a single tile

Adapted from Richard Korf presentation

100

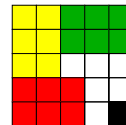
## Performance

**15 Puzzle:** 2000x speedup vs Manhattan dist

- IDA\* with the two DBs solves 15 Puzzle optimally in 30 milliseconds

**24 Puzzle:** 12 millionx speedup vs Manhattan

- IDA\* can solve random instances in 2 days.
- Requires 4 DBs as shown
  - Each DB has 128 million entries
- Without PDBs: 65000 years



Adapted from Richard Korf presentation

101

## Next: Local Search

How to climb hills

How to reach the top by annealing

How to simulate and profit from evolution

102

## Local search algorithms

In many optimization problems, the **path** to the goal is irrelevant; the goal state itself is the solution

Find configuration satisfying constraints,  
e.g., *n*-queens

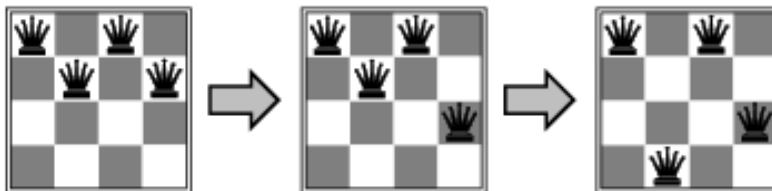
In such cases, we can use **local search algorithms**

Keep a single "current" state, try to improve it

103

## Example: *n*-queens

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



104

## Hill-climbing search

"Like climbing Everest in thick fog with amnesia"

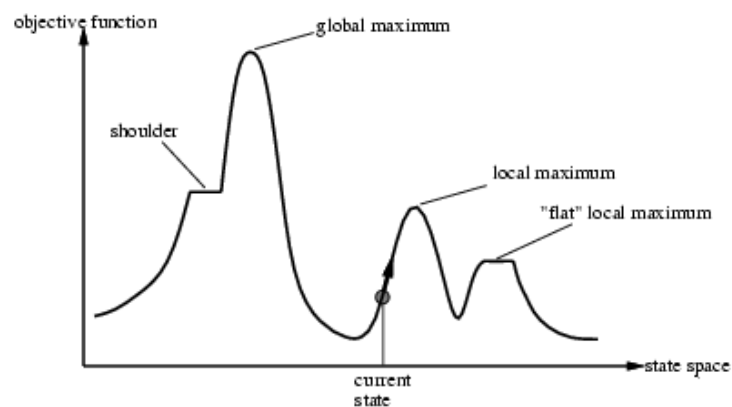
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                  neighbor, a node

  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
```

105

## Hill-climbing search

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



106

## Example: 8-queens problem

Heuristic?  
(Value function)

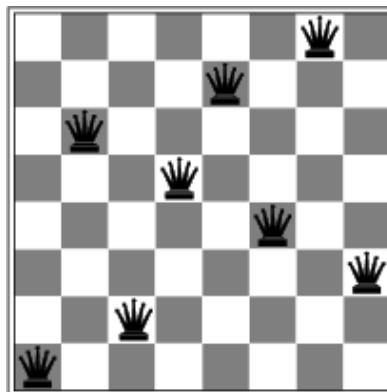
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♙	13	16	13	16
♙	14	17	15	♙	14	16	16
17	♙	16	18	15	♙	15	♙
18	14	♙	15	15	14	♙	16
14	14	13	17	12	14	12	18

$h$  = number of pairs of queens that are attacking each other, either directly or indirectly

$h = 17$  for the above state (would like to minimize this)

107

## Example: 8-queens problem



A local minimum with  $h = 1$ . Need  $h = 0$   
How to find global minimum (or maximum)?

108

## Simulated Annealing

Idea: escape local maxima by allowing some "bad" moves but gradually decrease their 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 ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
     $\Delta E \leftarrow \text{VALUE}[\textit{next}] - \text{VALUE}[\textit{current}]$ 
    if  $\Delta E > 0$  then current ← next
    else current ← next only with probability  $e^{\Delta E/T}$ 
```

109

## Properties of simulated annealing

One can prove: If  $T$  decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1

Widely used in VLSI layout, airline scheduling, etc

110

## Local Beam Search

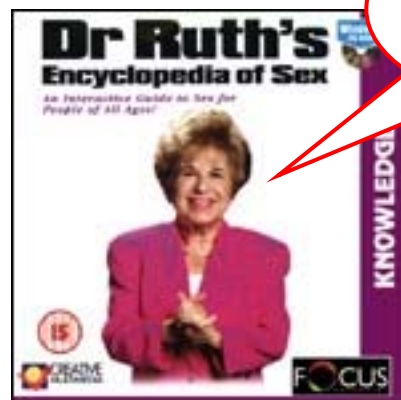
Keep track of  $k$  states rather than just one

Start with  $k$  randomly generated states

At each iteration, all the successors of all  $k$  states are generated

If any one is a goal state, stop; else select the  $k$  best successors from the complete list and repeat.

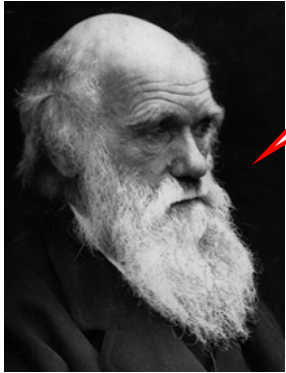
111



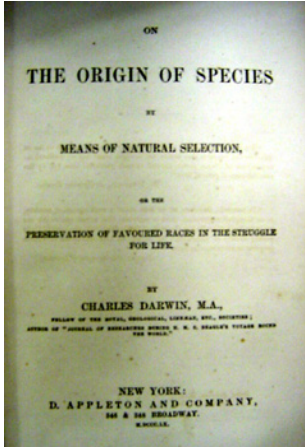
Hey, perhaps sex can improve search?

112





Sure - check out ye book.



ON  
THE ORIGIN OF SPECIES  
BY  
MEANS OF NATURAL SELECTION,  
OR THE  
PRESERVATION OF FAVOURED RACES IN THE STRUGGLE  
FOR LIFE.  
BY  
CHARLES DARWIN, M.A.  
FELLOW OF THE ROYAL SOCIETY, LONDON, ETC., SURVIVOR;  
AUTHOR OF "JOURNAL OF RESEARCHES INTO THE NATURAL HISTORY OF THE  
ZEPHYRUS."  
NEW YORK:  
D. APPLETON AND COMPANY,  
360 & 362 BROADWAY.  
KNOXVILLE.

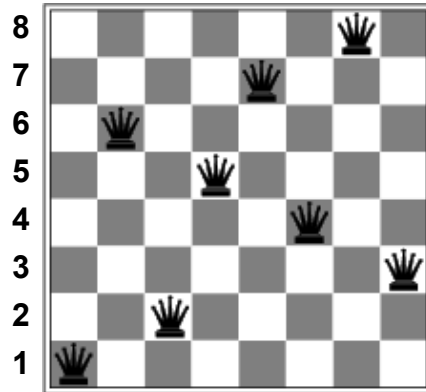
113

## Genetic Algorithms

- A successor state is generated by combining two parent states
- Start with  $k$  randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

114

## Example: 8-queens problem



String  
Representation:  
16257483

Can we evolve a solution through genetic algorithms?

115

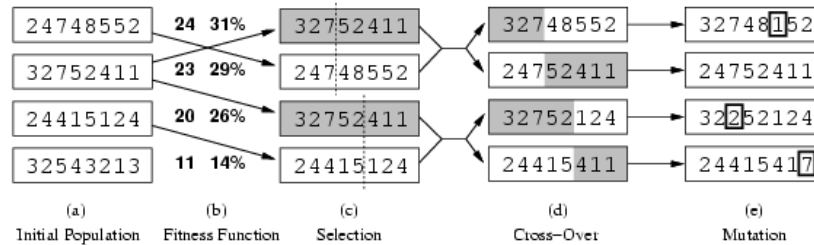
## Example: Evolving 8 Queens



Sorry, wrong queens

116

## Example: Evolving 8 Queens



Fitness function: number of non-attacking pairs of queens  
(min = 0, max =  $8 \times 7/2 = 28$ )

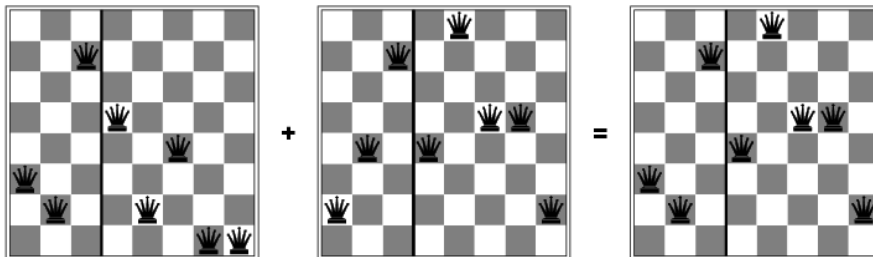
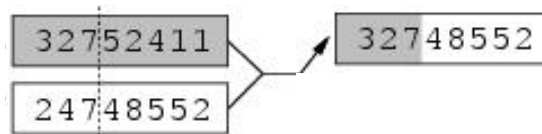
$24/(24+23+20+11) = 31\%$  probability of selection for reproduction

$23/(24+23+20+11) = 29\%$  etc

117

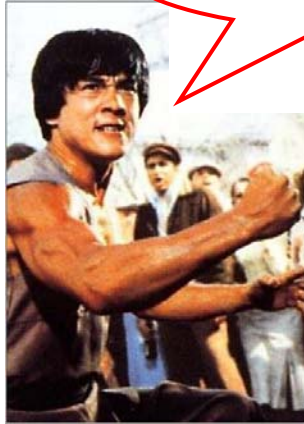


## Queens crossing over



118

Let's move on to  
adversarial games



## Adversarial Games

Programs that can play competitive board  
games

Minimax Search

Alpha-Beta Pruning



## Games Overview

	deterministic	chance
Perfect information	chess, checkers, go, othello	backgammon, monopoly
Imperfect information		poker, bridge, scrabble

121

## Games & Game Theory

When there is *more than one agent*, the future is not easily predictable anymore for the agent

In *competitive* environments (conflicting goals), *adversarial search* becomes necessary

In AI, we usually consider special type of games:

- *board games*, which can be characterized as *deterministic, turn-taking, two-player, zero-sum games with perfect information*

122

## Games as Search

### Components:

- **States:**
- **Initial state:**
- **Successor function:**
- **Terminal test:**
- **Utility function:**

123

## Games as Search

### Components:

- **States:** board configurations
- **Initial state:** the board position and which player will move
- **Successor function:** returns list of *(move, state)* pairs, each indicating a legal move and the resulting state
- **Terminal test:** determines when the game is over
- **Utility function:** gives a numeric value in terminal states (e.g., -1, 0, +1 in chess for loss, tie, win)

124

## Games as Search

**Convention:** first player is called MAX,

2nd player is called MIN

MAX moves first and they take turns until game is over

Winner gets reward, loser gets penalty

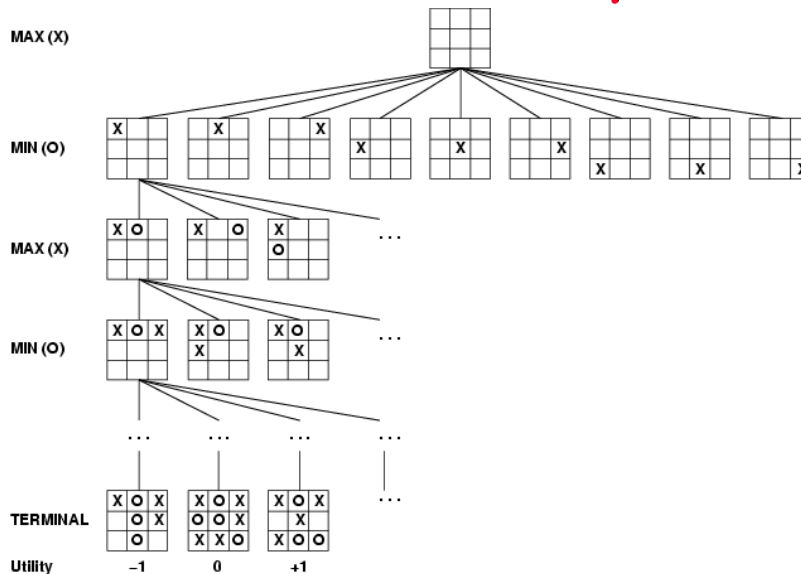
Utility values stated from MAX's perspective

Initial state and legal moves define the *game tree*

MAX uses game tree to determine next move

125

## Tic-Tac-Toe Example



126

## Optimal Strategy: Minimax Search

Find the contingent *strategy* for MAX assuming an infallible MIN opponent

Assumption: Both players play optimally!

Given a game tree, the optimal strategy can be determined by using the *minimax* value of each node (defined recursively):

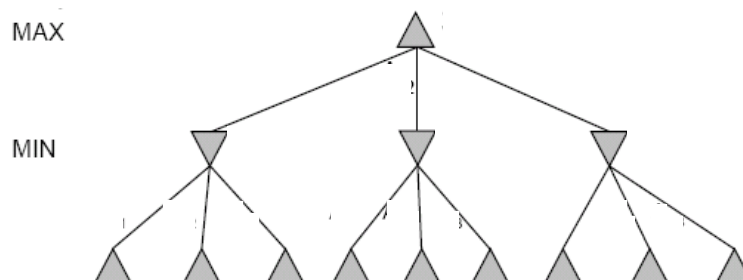
MINIMAX-VALUE( $n$ )=

UTILITY( $n$ )                      If  $n$  is a terminal  
 $\max_{s \in \text{succ}(n)} \text{MINIMAX-VALUE}(s)$    If  $n$  is a MAX node  
 $\min_{s \in \text{succ}(n)} \text{MINIMAX-VALUE}(s)$    If  $n$  is a MIN node

127

## Two-Ply Game Tree

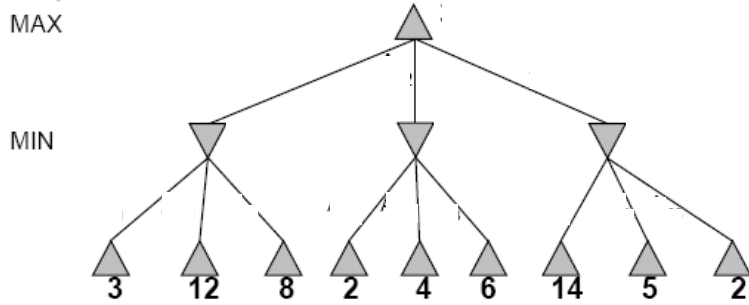
“Ply” = move by 1 player



128

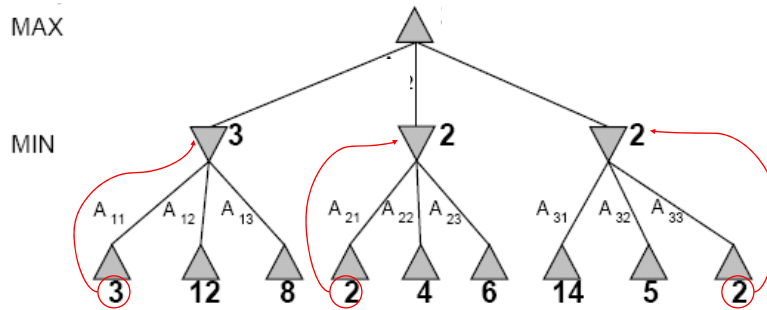


## Two-Ply Game Tree



129

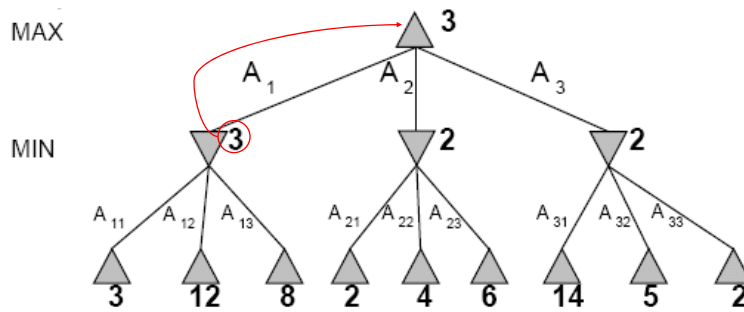
## Two-Ply Game Tree



130

## Two-Ply Game Tree

Minimax decision =  $A_1$



*Minimax maximizes the worst-case outcome for max*

131

## What if MIN does not play optimally?

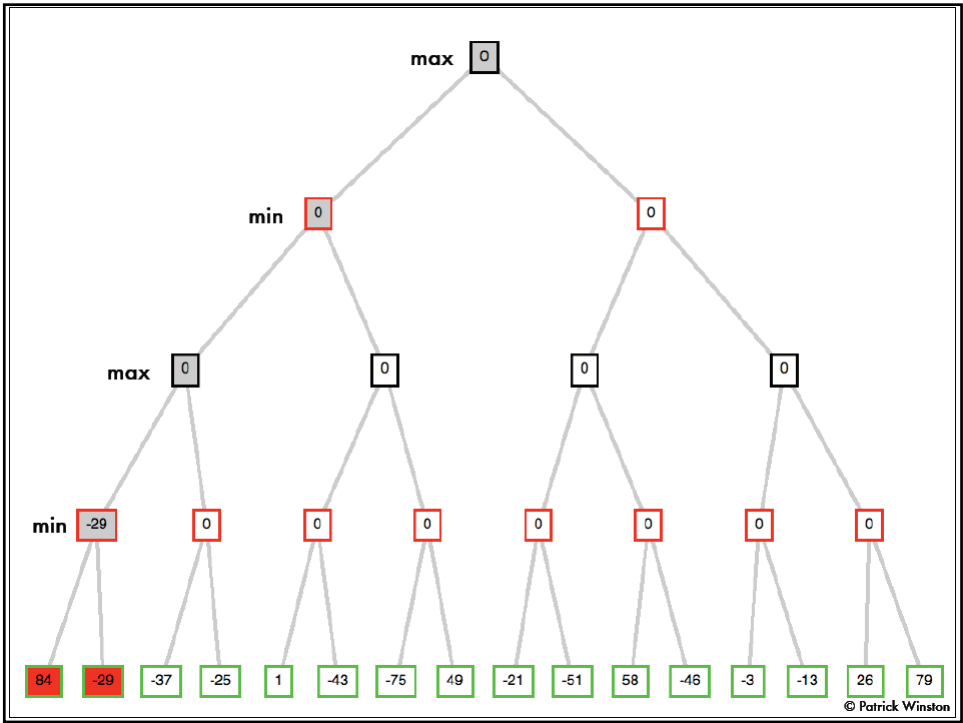
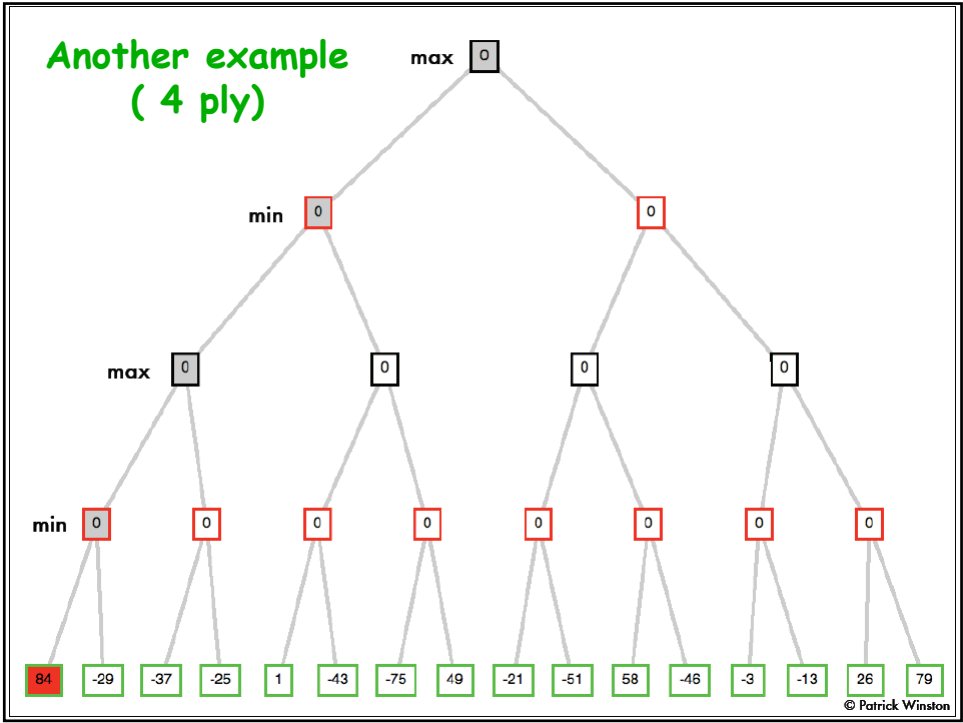
Definition of optimal play for MAX assumes MIN plays optimally

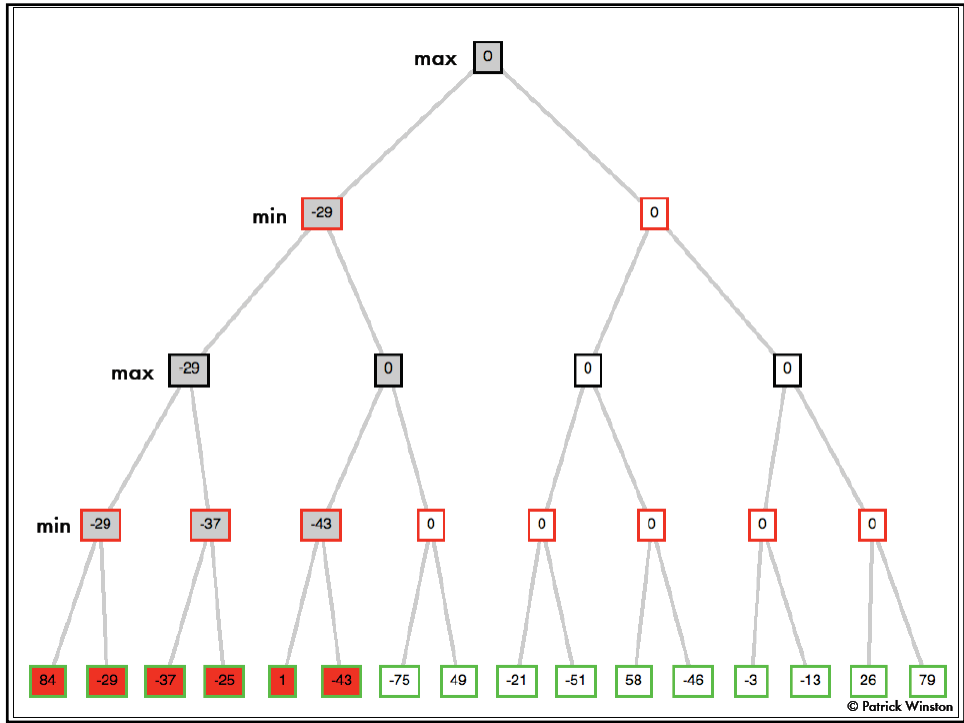
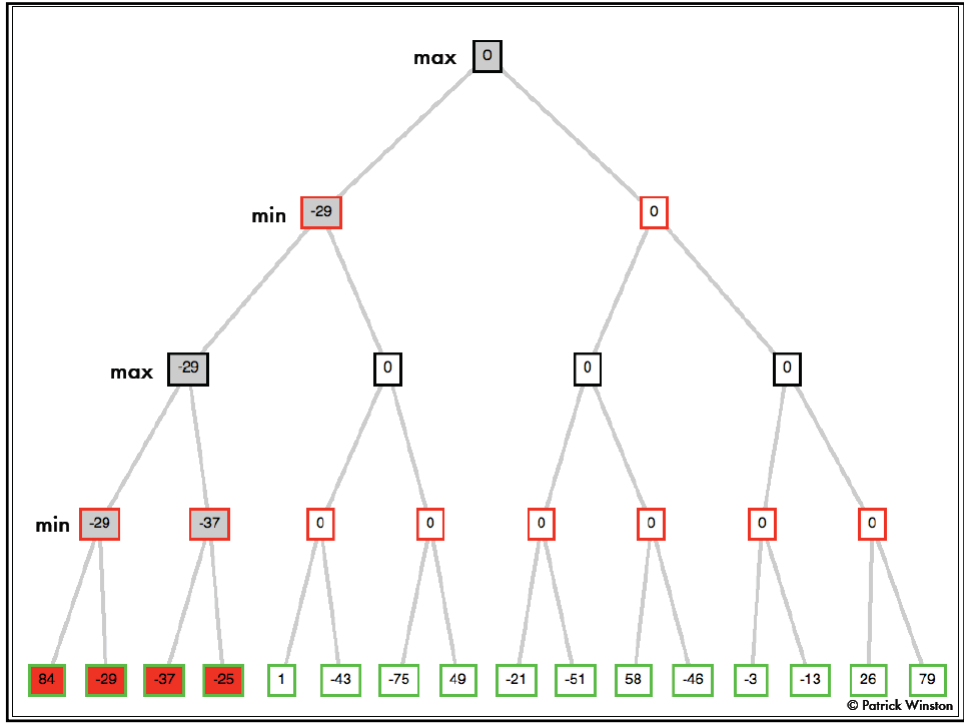
- Maximizes worst-case outcome for MAX

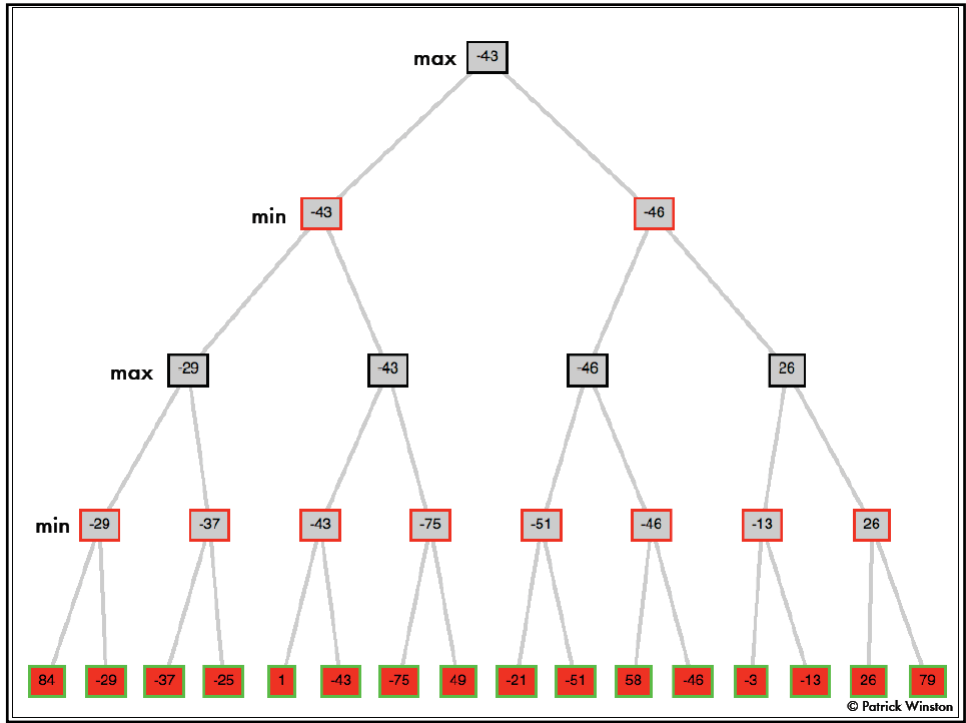
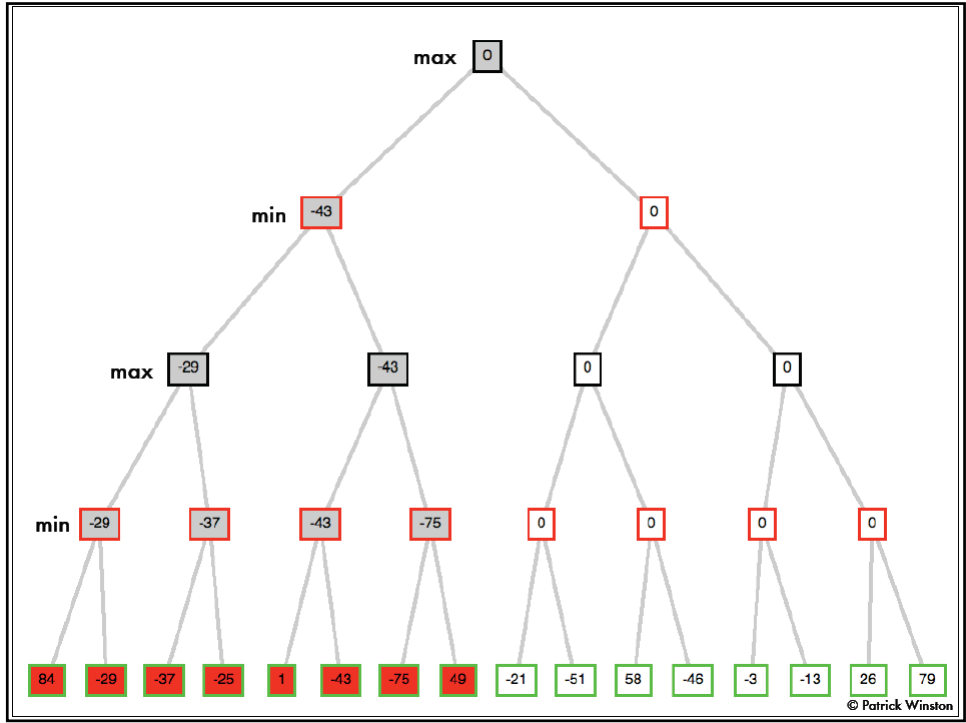
If MIN does not play optimally, MAX will do even better (i.e. at least as much or more utility obtained than if MIN was optimal) [Exercise 5.7 in textbook]

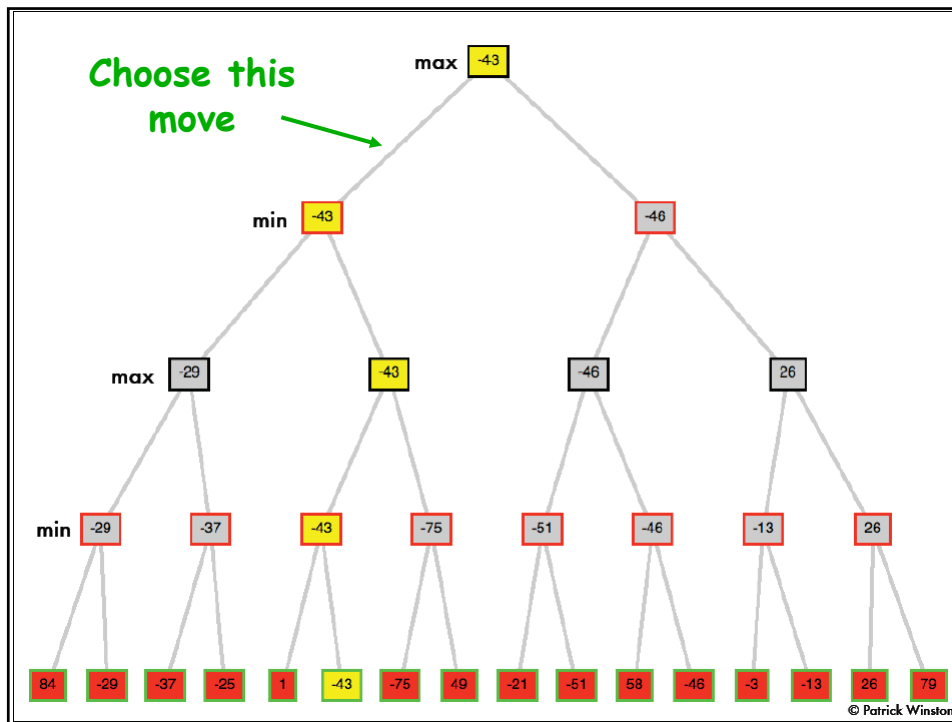
132

Another example  
( 4 ply)









## Minimax Algorithm

**function** MINIMAX-DECISION(*state*) *returns an action*

$v \leftarrow \text{MAX-VALUE}(\textit{state})$

**return** the action in SUCCESSORS(*state*) with value  $v$

---

**function** MAX-VALUE(*state*) *returns a utility value*

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow -\infty$

**for**  $a, s$  in SUCCESSORS(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$

**return**  $v$

---

**function** MIN-VALUE(*state*) *returns a utility value*

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow \infty$

**for**  $a, s$  in SUCCESSORS(*state*) **do**

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$

**return**  $v$

## Properties of minimax

**Complete?** Yes (if tree is finite)

**Optimal?** Yes (against an optimal opponent)

**Time complexity?**  $O(b^m)$

**Space complexity?**  $O(bm)$  (depth-first exploration)

141

## Good enough?

**Chess:**

- branching factor  $b \approx 35$
- game length  $m \approx 100$
- search space  $b^m \approx 35^{100} \approx 10^{154}$

**The Universe:**

- number of atoms  $\approx 10^{78}$
- age  $\approx 10^{21}$  milliseconds

**Can we search more efficiently?**

142

**Next Class:  
Wrap up of search  
Logic and Reasoning**



**To do:  
Homework #1  
Sign up for class mailing list**