Intelligent Autonomous Agents
ICS 606 / EE 606
Fall 2011

Nancy E. Reed

nreed@hawaii.edu

Lecture #4
Agent Problems and Search

Problem types
1. Path planning (FWGC, n-puzzle,…)
2. Constraint satisfaction (8 queens, ..)
3. Two-player games (tic-tac-toe, ..)

Search strategies
1. Uninformed – breadth-first, depth-first
2. Informed – best-first

- References
  - Weiss, Chap 4
  - R & N, Chap 3
- Figures © MIT Press/Prentice Hall

Path-finding & CSP

Most algorithms for these classes were originally developed for a single-agent

What kinds of algorithms would be useful for cooperative problem solving by multiple agents?

Search Algorithm Graph Representation

A search problem can be represented by using a graph.

Some problems can be solved by accumulating local computations for each node in the graph.

Search Algorithms for Agents

Common search methods
1. Breadth-first (uninformed)
2. Depth-first (uninformed)
3. Best-first (informed – needs a heuristic evaluation function)

- References
  - Weiss, Chap 4
  - R & N, Chap 3

Search algorithms

Basic idea:

offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

Function General-Search(*problem*, *strategy*) returns a *solution*, or failure
initialize the search tree using the initial state 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 resulting nodes to the search tree
end

7

Implementation of search algorithms

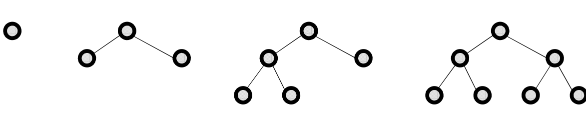
Function General-Search(problem, Queuing-Fn) **returns** a solution, or failure
nodes ← make-queue(make-node(initial-state[problem]))
loop do
if node is empty **then return** failure
node ← Remove-Front(nodes)
if Goal-Test[problem] applied to State(node) succeeds
then return node
nodes ← Queuing-Fn(nodes, Expand(node, Operators[problem]))
end

Queuing-Fn(*queue, elements*) is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.

8

Breadth First Search

Enqueue nodes in FIFO (first-in, first-out) order.



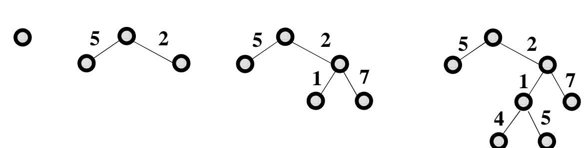
Intuition: Expand all nodes at depth i before expanding nodes at depth i + 1

- Complete? Yes.
- Optimal? Yes, if path cost is nondecreasing function of depth
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$, note that every node in the fringe is kept in the queue.

9

Uniform Cost Search

Enqueue nodes in order of cost



Intuition: Expand the cheapest node. Where the cost is the path cost g(n)

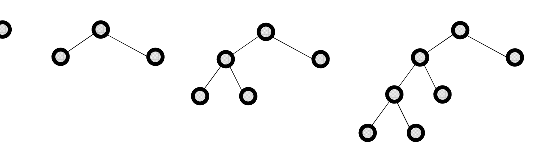
- Complete? Yes.
- Optimal? Yes, if path cost is nondecreasing function of depth
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$, note that every node in the fringe keep in the queue.

Note that Breadth First search can be seen as a special case of Uniform Cost Search, where the path cost is just the depth.

10

Depth First Search

Enqueue nodes in LIFO (last-in, first-out) order.



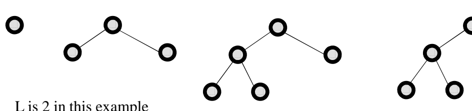
Intuition: Expand node at the deepest level (breaking ties left to right)

- Complete? No (Yes on finite trees, with no loops).
- Optimal? No
- Time Complexity: $O(b^m)$, where m is the maximum depth.
- Space Complexity: O(bm), where m is the maximum depth.

11

Depth-Limited Search

Enqueue nodes in LIFO (last-in, first-out) order. But limit depth to L



L is 2 in this example

Intuition: Expand node at the deepest level, but limit depth to L

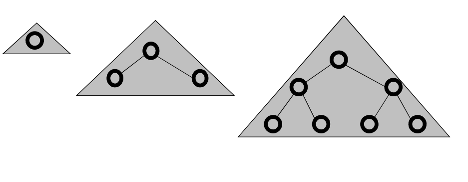
- Complete? Yes if there is a goal state at a depth less than L
- Optimal? No
- Time Complexity: $O(b^L)$, where L is the cutoff.
- Space Complexity: O(bL), where L is the cutoff.

Picking the right value for L is a difficult, Suppose we chose 7 for FWDC, we will fail to find a solution...

12

Iterative Deepening Search I

Do depth limited search starting a L = 0, keep incrementing L by 1.

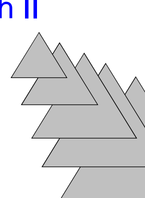


Intuition: Combine the Optimality and completeness of Breadth first search, with the low space complexity of Depth first search

- Complete? Yes
- Optimal? Yes
- Time Complexity: $O(b^d)$, where d is the depth of the solution.
- Space Complexity: O(bd), where d is the depth of the solution.

Iterative Deepening Search II

Iterative deepening looks wasteful because we re-explore parts of the search space many times...



Consider a problem with a branching factor of 10 and a solution at depth 5...

```
1+10+100+1000+10,000+100,000 = 111,111

1
1+10
1+10+100
1+10+100+1000
1+10+100+1000+10,000
1+10+100+1000+10,000+100,000
                              = 123,456
```


Heuristic Search

The search techniques we have seen so far...

- Breadth first search
- Uniform cost search
- Depth first search
- Depth limited search
- Iterative Deepening

← uninformed search
blind search

...are all too slow for most real world problems

Some states may appear better that others...

7	8	4
3	5	1
6	2	

1	2	3
4	5	6
7		8

FWG / C

G / FW C

...we can use this knowledge of the relative merit of states to guide search

Heuristic Search (informed search)

A **heuristic** is a function that, when applied to a state, returns a number that is an *estimate of the merit of the state*, with respect to the goal.

In other words, the heuristic tells us approximately how far the state is from the goal state*.

Note we said "approximately". Heuristics might underestimate or overestimate the merit of a state.
Heuristics that *never* overestimate are very desirable, and are called **admissible**.

*I.e Smaller numbers are better (you are closer to a goal).

The A* Algorithm ("A-Star")

Enqueue nodes based on an estimate of cost to a goal, *f*(n)

g(n) is the cost to get to a node.
h(n) is the estimated cost to go from the current node to a goal.

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

We can think of *f*(n) as the estimated cost of the cheapest solution that goes through node n

Note that we can use the general search algorithm we used before.
All that we have changed is the queuing strategy.

If the heuristic is optimistic, that is to say, it *never overestimates* the distance to the goal, then…
A* is optimal and complete!
= Will always find a lowest-cost solution.

How fast is A*?

A* is the fastest search algorithm. That is, for any given heuristic, no algorithm can expand fewer nodes than A*.

How fast is it? Depends of the quality of the heuristic.

- If the heuristic is useless (ie *h*(n) is hardcoded to equal 0), the algorithm degenerates to uniform cost.
- If the heuristic is perfect, there is no real search, we just march down the tree to the goal.

Generally we are somewhere in between the two situations above. The time taken depends on the quality of the heuristic.

What is A*'s space complexity?

A* has *worst case* O(b^d^) space complexity, i.e. all parent nodes are examined before finding a goal,

A* *average* space complexity is much better. For example, use iterative deepening version is possible (IDA*)

Best case space complexity is?

Example - Problem Design

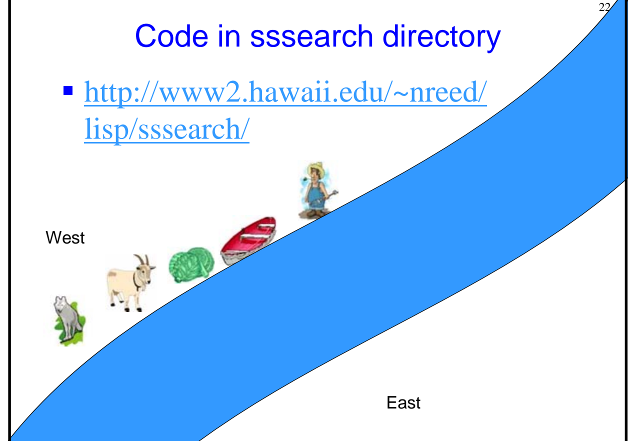
- State-space search problem
  - World can be in one of several states
  - Moves can change the state of the world
- Design/create data structures to represent the "world"
- Design/create functions representing "moves" in the world

Example: Farmer, Wolf, Goat, and Cabbage Problem

- A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross.
- There is a boat at the river's edge, but, of course, only the farmer can row.
- The boat can carry a maximum of the farmer and one other animal/vegetable.
- If the wolf is ever left alone with the goat, the wolf will eat the goat.
- Similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage.
- Devise a sequence of crossings of the river so that all four of them arrive safely on the other side of the river.

Code in sssearch directory

- http://www2.hawaii.edu/~nreed/lisp/sssearch/



FWGC State Creation and Access

- Functions to create and access states

```
;; Use a list to represent the state of the world –
;;i.e. the side of the river that each of the 4
;;characters is on.  East and West are the sides.

(defun make-state (f w g c) (list f w g c))

(defun farmer-side (state)
 (nth 0 state)) ; or (car state)

(defun wolf-side (state)
 (nth 1 state))  ; or (cadr state)

(defun goat-side (state)
 (nth 2 state))  ; or (caddr state)

(defun cabbage-side (state)
 (nth 3 state))   ; or (cadddr state)
```


FWGC Start, Goal and Moves

- Global variables

;; Represent start and goal states and list of possible moves

```
; Start state *start*
(setq *start* (make-state 'e 'e 'e 'e))
; Goal state *goal*
(setq *goal* (make-state 'w 'w 'w 'w))

; Possible moves *moves*
(setq *moves* '(farmer-takes-self farmer-takes-wolf
  farmer-takes-goat farmer-takes-cabbage))
; The order moves are listed here determines the
; order they are expanded.
; Placement in expanded node list depends on the
  queuing function (type of search
```

25

Utility Functions – Opposite and Safe

;; Note – these are *very simple*, however they abstract out details which makes it easy
;; to change parameters. For example: N&S instead of E&W.

```
(defun opposite (side) ; Return the opposite side of the river
 (cond
  ((equal side 'e) 'w)
  ((equal side 'w) 'e)))

      ; if state is safe, return the state, else return nil/false
(defun safe (state)
 (cond
  ((and (equal (goat-side state) (wolf-side state))
       (not (equal (farmer-side state) (wolf-side state))))
       nil)          ; wolf eats goat
  ((and (equal (goat-side state) (cabbage-side state))
       (not (equal (farmer-side state) (goat-side state))))
         nil)       ; goat eats cabbage
  (t state)))       ; otherwise it is safe
```

26

Move Functions

```
(defun farmer-takes-self (state)
 (cond
  ((safe             ; check if resulting sate would be safe
     (make-state     ; make new state resulting from move
     (opposite (farmer-side state))
            (wolf-side state)
            (goat-side state)
            (cabbage-side state)
     ))              ; return the new state
  (t nil)))          ; otherwise not safe - return nil/False

(defun farmer-takes-wolf (state)
 (cond ; possible to take wolf if on same side as farmer
  ((equal (farmer-side state) (wolf-side state))
    (safe ; check if resulting sate would be safe
     (make-state     ; make new state resulting from move
            (wolf-side state)
            (opposite (farmer-side state) )
            (opposite (wolf-side state))
            (goat-side state)
           (cabbage-side state))
     ))               ; return the new state
   (t nil)))          ; otherwise not safe - return nil/False
; remaining move functions are similar.  See Lisp files
```

27

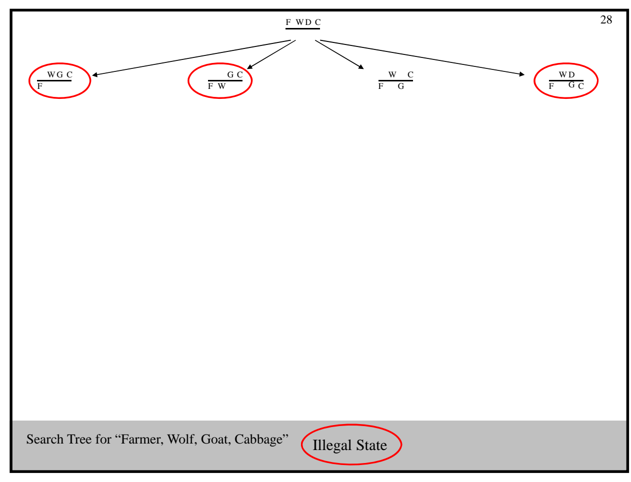
Another view of states

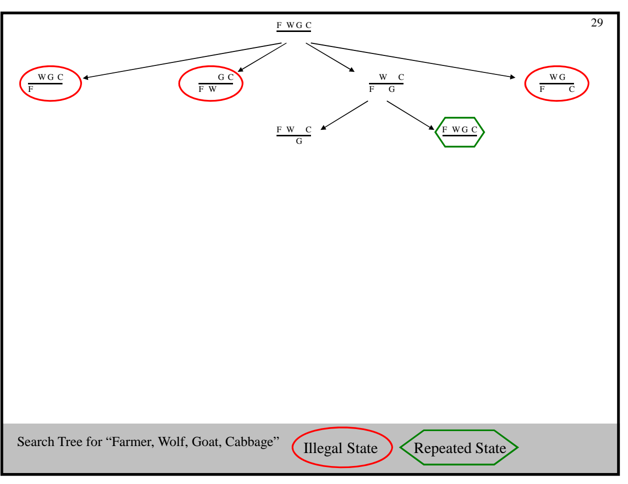
This means that everybody/everything is on the same side of the river.

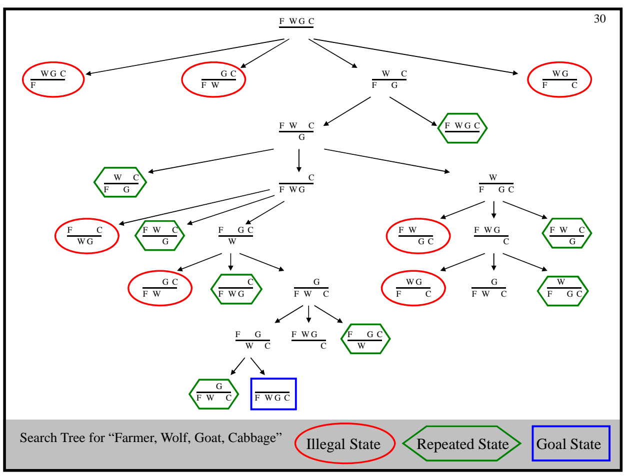
$$\underline{\text{F W G C}}$$

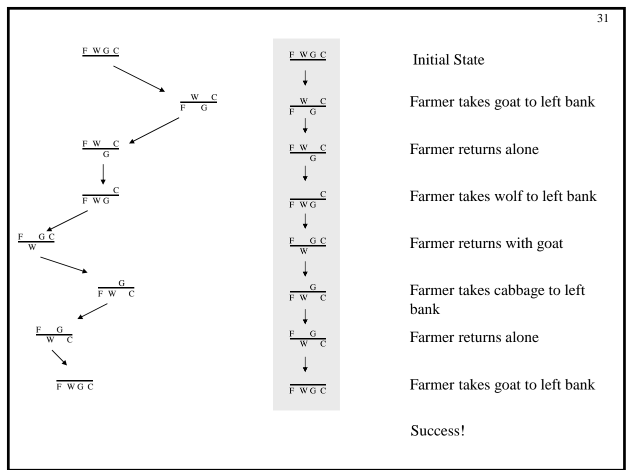
This means that we somehow got the Wolf to the other side.

$$\frac{\text{F} \quad \text{G C}}{\text{W}}$$







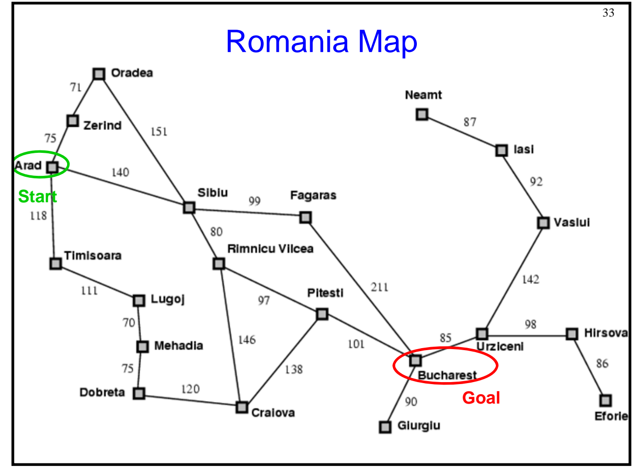


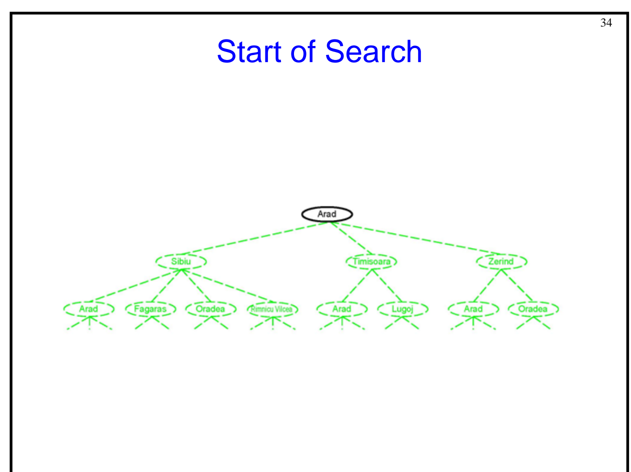
32

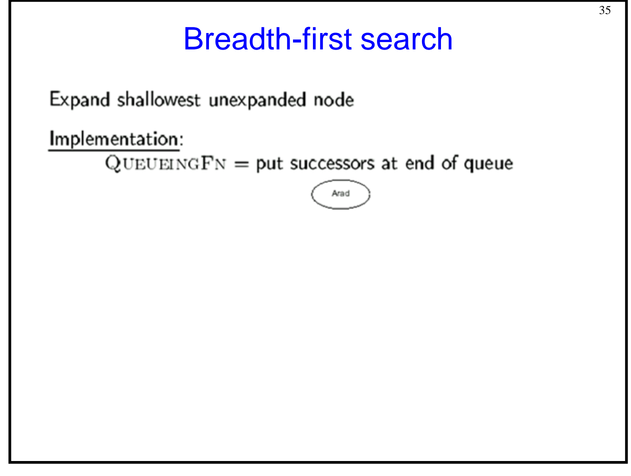
Another Example: Travel in Romania

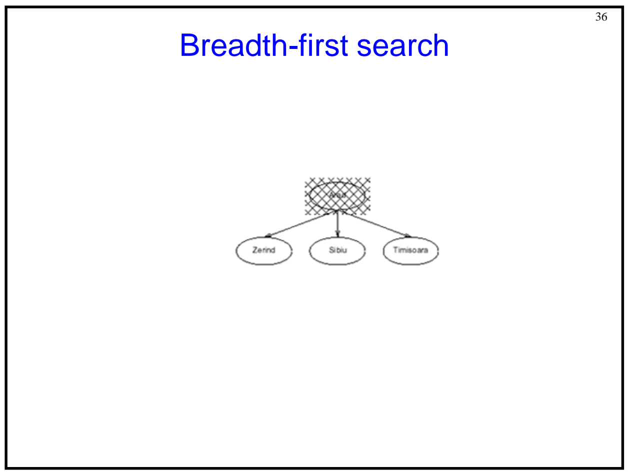
- You're in Romania, on vacation in Arad.
- Your flight leaves tomorrow from Bucharest.

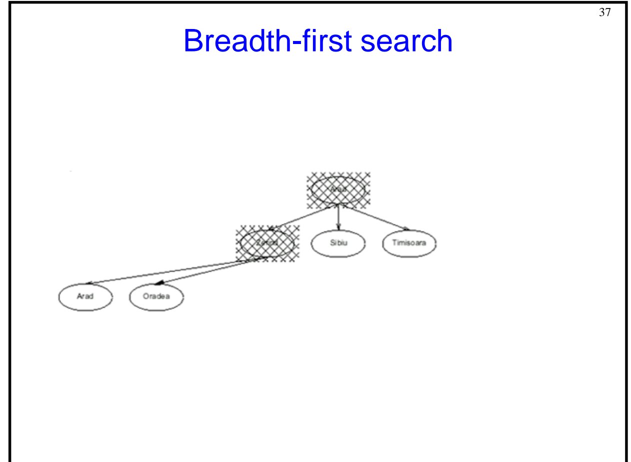
- Formulate problem:
  - States: various cities
  - Operators: drive between cities
- Goal: be in Bucharest
- Start: in Arad
- Solution is a sequence of cities, such that *total driving distance* (Arad → Bucharest) *is minimized.*
- Search – try breadth-first, depth-first and best-first

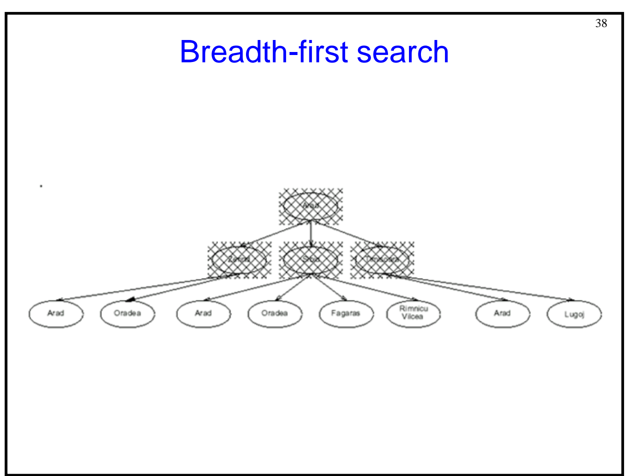


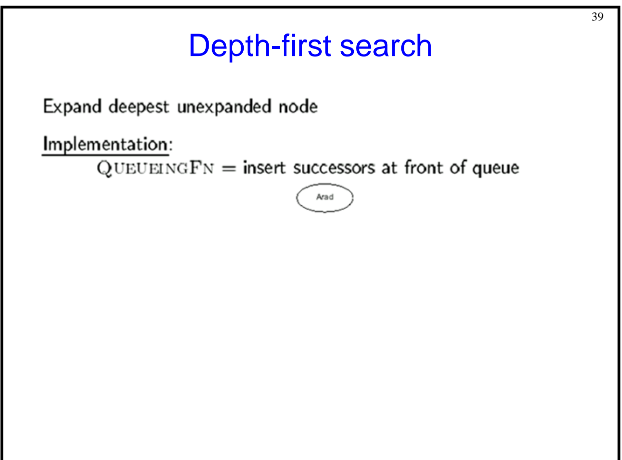


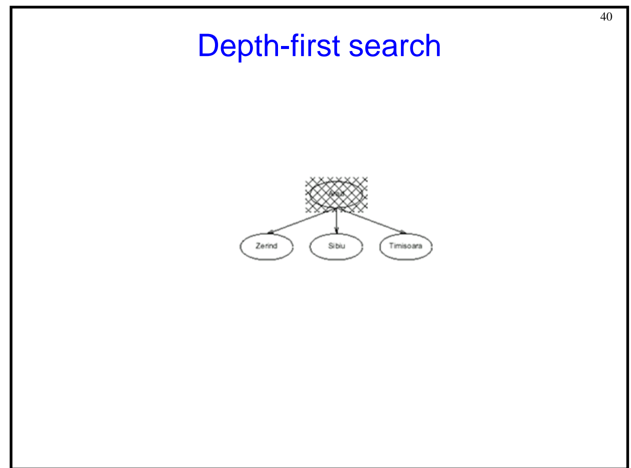


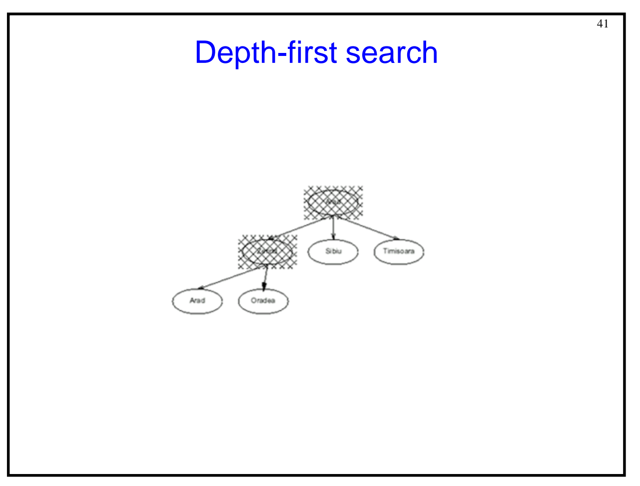


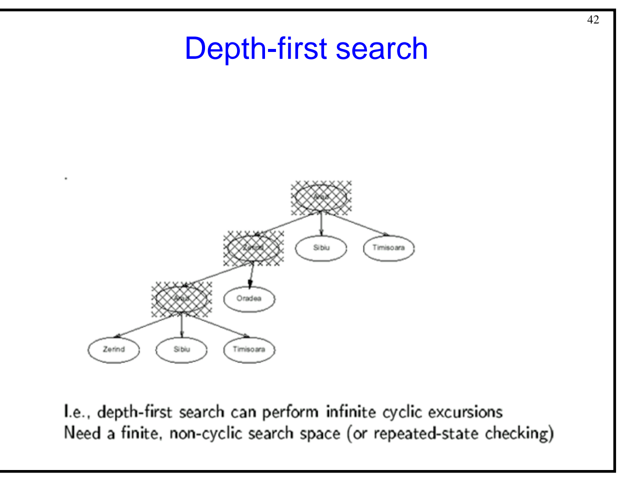

37
Breadth-first search
Sibiu
Timisoara
Arad
Oradea


38
Breadth-first search
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu Vilcea
Arad
Lugoj


39
Depth-first search
Expand deepest unexpanded node
Implementation:
QUEUEINGFN = insert successors at front of queue
Arad


40
Depth-first search
Zerind
Sibiu
Timisoara


41
Depth-first search
Sibiu
Timisoara
Arad
Oradea


42
Depth-first search
Sibiu
Timisoara
Oradea
Zerind
Sibiu
Timisoara
I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

Avoiding repeated states

In increasing order of effectiveness and computational overhead:

- do not return to state we come from, i.e., expand function will skip possible successors that are in same state as node's parent.
- do not create paths with cycles, i.e., expand function will skip possible successors that are in same state as any of node's ancestors.
- do not generate any state that was ever generated before, by keeping track (in memory) of every state generated.

Depth-limited search

Is a depth-first search with depth limit *l*

Implementation:
Nodes at depth *l* have no successors.

Complete: if cutoff chosen appropriately then it is guaranteed to find a solution.

Optimal: no, it does not guarantee finding the least-cost solution

Iterative deepening search

```
Function Iterative-deepening-Search(problem) returns a solution, or failure
  for depth = 0 to ∞ do
      result ← Depth-Limited-Search(problem, depth)
      if result succeeds then return result
  end
  return failure
```

Combines the best of breadth-first and depth-first search strategies.

- Completeness: Yes,
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality: Yes, if step cost = 1

Summary of Un-Informed Search Strategies

Method / Criterion	Breadth-first	Depth-first	Depth-limited	Iterative deepening	Bidirectional (if applicable)
Time	b^d	b^m	b^l	b^d	$b^{(d/2)}$
Space	b^d	bm	bl	bd	b^d
Optimal?	Yes	No	No	Yes	Yes
Complete?	Yes	No	Yes if l≥d	Yes	Yes

- *b* – max branching factor of the search tree
- *d* – depth of the least-cost solution
- *m* – max depth of the state-space (may be infinity)
- l – depth cutoff

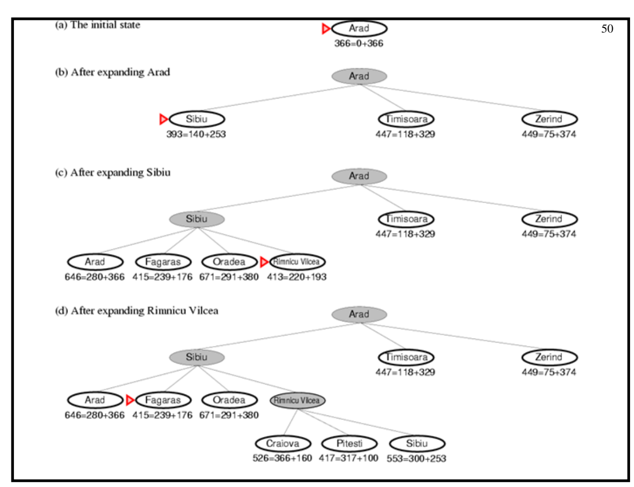
Best-First/Heuristic Search (Informed)

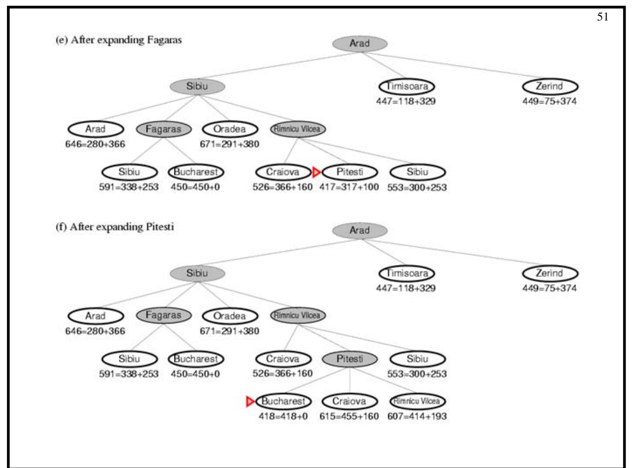
- Uses a function to estimate the cost of finding a goal from current state F=G+H
- Fn(node) = Cost(start-to-current-node) + HeuristicEst(current-node-to-goal)
- Priority queue - nodes ordered by estimated cost of path to reach goal
- Expand node with lowest cost first
- Guarantee of optimal solution if H never overestimates true cost to goal
- If H = 1 => breadth first!

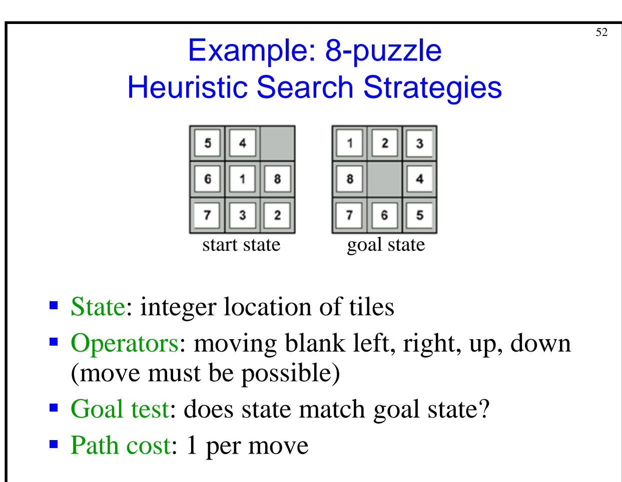
Romania Straight-Line Distances to Bucharest = h(n) heuristic function for distance left to goal

City	h	City	h
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

This is the heuristic value for remaining cost







53

Encapsulating *state* information in *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!

parent
State
Node
depth = 6
g = 6
state
children

The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

54

;;; HEURISTIC evaluation functions for the n-puzzle problem

```
; This evaluation function can be used in the absence of a better one
; It looks at the elements of a state and counts how many are the same
; in that state and the goal.
; HEURISTIC-EVAL-0 - Return the number of tiles out of place.
(defun heuristic-eval-0 (state goal)
  (cond
    ((null state) 0)                 ; all tiles processed
    ((equal (car state) *blank*)  ; the blank doesn't count as a tile
       (heuristic-eval-0 (cdr state) (cdr goal))) ; Return value of next
    ((equal (car state) (car goal))   ; the tiles are the same
         (heuristic-eval-0 (cdr state) (cdr goal))) ; Return value of next
    (t                          ; otherwise, Add 1 (for this tile) to
       (1+ (heuristic-eval-0 (cdr state) (cdr goal)))))) ; cost of rest
```

Heuristic Evaluation Function #1

```
; HEURISTIC-EVAL-1 - Return the sum of the distances tiles are out of place.
; A better heuristic for the n-puzzle problem than # of tiles out of place.

(defun heuristic-eval-1 (state goal)
  "Sum the distance for each tile from its goal position."
  (do
   ((sum 0) (tile 1 (+ tile 1))) ;loop adding dist for tiles 1-(n^2 - 1)
   ((equal tile (* *n* *n*)) sum)  ; return sum when done
   (setq sum (+ sum (distance tile state goal)))) )
```


Heuristic Evaluation Function #2

```
; HEURISTIC-EVAL-2 - Return the sum of the distances out of place
; plus two times the number of direct tile reversals.

(defun heuristic-eval-2 (state goal)
  "Sum of distances plus 2 times the number of tile reversals"
  (+
   (heuristic-eval-1 state goal)
   (* 2 (tile-reversals state goal)) ))

; Call the desired heuristic here from heuristic-eval
(defun heuristic-eval (state goal)
  (heuristic-eval-2 state goal))
```



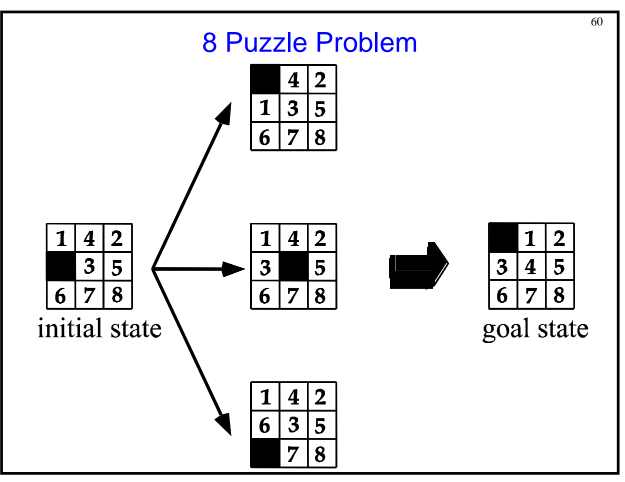
```
; TILE-REVERSALS - Return the number of tiles directly reversed between  state and goal.
(defun tile-reversals (state goal)
  "Calculate the number of tile reversals between two states"
  (+
   (do
    ((sum 0) (i 0 (1+ i)))       ; loop checking row adjacencies
    ((equal i (- *n* 1)) sum)    ; until i = n-2, return sum
    (do
     ((j 0 (1+ j)))
     ((equal j *n*) )            ; until j = n-1, nothing to return
     (setq sum
           (+ sum
              (tile-reverse (+ (* i *n*) j) (+ (* i *n*) j 1)
                            state goal)))))
   (do
    ((sum 0) (i 0 (1+ i)))       ; loop checking column adjacencies
    ((equal i *n* ) sum)         ; until i = n-1, return sum
    (do
     ((j 0 (1+ j)))
     ((equal j (- *n* 1)) )       ; until j = n-2, nothing to return
    (setq sum
     (+ sum
        (tile-reverse (+ (* i *n*) j) (+ (* i *n*) j 1)
                      state goal)))) )))
```



```
;  TILE-REVERSE - Return 1 if the tiles are reversed in the two states,
else ; return 0.  If one of the tiles is the blank, it doesn't count
(return 0).
(defun tile-reverse (pos1 pos2 state1 state2)
  "Return 1 if the tiles in the two positions are reversed, else 0"
  (cond
   ((or
     (equal (nth pos1 state1) *blank*)      ; blanks don't count in
reversals
     (equal (nth pos2 state1) *blank*))
    0)                                      ; return 0
   ((and                                    ; the tiles are reversed
     (equal (nth pos1 state1) (nth pos2 state2))
     (equal (nth pos2 state1) (nth pos1 state2)))
    1)                                      ; return 1
   (t 0)) )                                 ; else return 0
```


Distance Function

```
; DISTANCE - calculate the distance a tile is from its
goal position.
(defun distance (tile state goal)
  "Calculate the Manhattan distance a tile is from its
goal position."
  (+
   (abs (- (row tile state)
           (row tile goal)))
   (abs (- (column tile state)
           (column tile goal)))) )
```

The 8 Puzzle in AIMA

```
;;;; The 8-Puzzle Problem
;;; In this implementation of the 8-puzzle we have a mix of priorities
;;; between efficiency and simplicity.  We restrict ourselves to the
;;; 8-puzzle, instead of the general n-puzzle.  The representation of
;;; states is not the obvious one (a 3x3 array), but it is both
;;; efficient and fairly easy to manipulate.  We represent each tile
;;; as an integer from 0 to 8 (0 for the blank).  We also represent
;;; each square as an integer from 0 to 8, arranged as follows:
;;;
;;;      0 1 2
;;;      3 4 5
;;;      6 7 8
;;;
```

http://aima.cs.berkeley.edu/

http://www2.hawaii.edu/~nreed/aima/

8 Puzzle, cont.

```
;;; Finally, we represent a state (i.e., a complete puzzle) as the sum
;;; of the tile numbers times 9 to the power of the tile's square
;;; number.  For example, the goal state from page 63:
;;;
;;;        1 2 3                          1*9^0 + 2*9^1 + 3*9^2
;;;        8 . 4  is represented by:    + 8*9^3 + 0*9^4 + 4*9^5
;;;        7 6 5                        + 7*9^6 + 6*9^7 + 5*9^8 = 247893796
;;;
;;; We represent actions with the four symbols <, >, ^, V to stand for
;;; moving the blank tile left, right, up and down, respectively.  The
;;; heuristic functions implicitly refer to the goal, *8-puzzle-goal*.
;;; Call the function USE-8-PUZZLE-GOAL to change the goal state if
;;; you wish.
```


8 Puzzle Structures & Methods

```
(defvar *8-puzzle-goal* "To be defined later")
(defstructure (8-puzzle-problem
               (:include problem
                         (initial-state (random-8-puzzle-state))
                         (goal  *8-puzzle-goal*)))
  "The sliding tile problem known as the 8-puzzle.")
(defmethod successors ((problem 8-puzzle-problem) state)
  "Generate the possible moves from an 8-puzzle state."
  (let ((blank (blank-square state))
        (result nil))
     (for each (action destination) in (8-puzzle-legal-moves blank) do
          (push (cons action (move-blank state blank destination))
result))
     result))
```


Simple Heuristic Function

```
(defmethod h-cost ((problem 8-puzzle-problem) state)
  "Manhattan, or sum of city block distances.  This is h_2 on [p 102]."
  (let ((sum 0))
   (for square = 0 to 8 do
        (let ((tile (8-puzzle-ref state square)))
          (unless (= tile 0)
            (incf sum (x+y-distance (8-puzzle-location square)
                                    (8-puzzle-goal-location tile))))))
   sum))
```


Alternative Heuristic Functions (AIMA)

```
(defun misplaced-tiles (state)
  "Number of misplaced tiles.  This is h_1 on [p 102]."
  (let ((sum 0))
   (for square = 0 to 8 do
        (when (misplaced-tile? state square) (incf sum)))
   sum))
(defun misplaced-tile? (state square)
  "Is the tile in SQUARE different from the corresponding goal  tile?
  Don't count the blank."
  (let ((tile (8-puzzle-ref state square)))
    (and (/= tile 0)
         (/= tile (8-puzzle-ref *8-puzzle-goal* square)))))
```


Evaluation of Search Strategies

- A search strategy is defined by picking the order of node expansion.
- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Time complexity:** how long does it take as function of num. of nodes?
  - **Space complexity:** how much memory does it require?
  - **Optimality:** does it guarantee the least-cost solution?
- Time and space complexity are measured in terms of:
  - b – max branching factor of the search tree
  - d – depth of the least-cost solution
  - m – max depth of the state-space (may be infinity)

67

Complexity

- Why worry about complexity of algorithms?
  - because a problem may be solvable in principle but may take too long to solve in practice
- How can we evaluate the complexity of algorithms?
  - through asymptotic analysis, i.e., estimate time (or number of operations) necessary to solve an instance of size n of a problem when n tends towards infinity
- See AIMA, Appendix A

68

Lisp Programming Examples

- State Space Search
  - A "world"
    - Starting state
    - Goal state
  - A set of possible moves
  - Searching strategies
- Code is posted on website
  - http://www2.hawaii.edu/~nreed/lisp/sssearch/

69

Summary

- Uninformed search
  - Breadth-first
  - Depth-first
- Informed search
  - Best-first (e.g. A*)

70

Questions