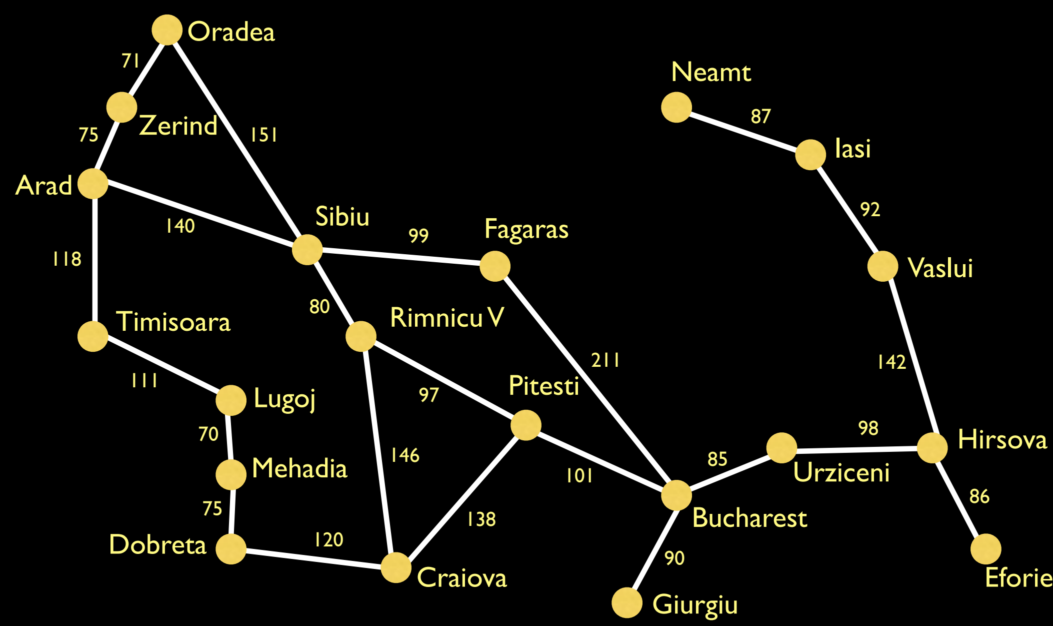
Uninformed Search

Traveling through Romania

- currently in Arad
- flight leaves tomorrow from Bucharest
- goal: be in Bucharest
- problem:
  - states = various cities
  - actions = drive between cities
- solve:
  - find the sequence of cities to drive through: Arad, Sibiu, Fagaras, Bucharest

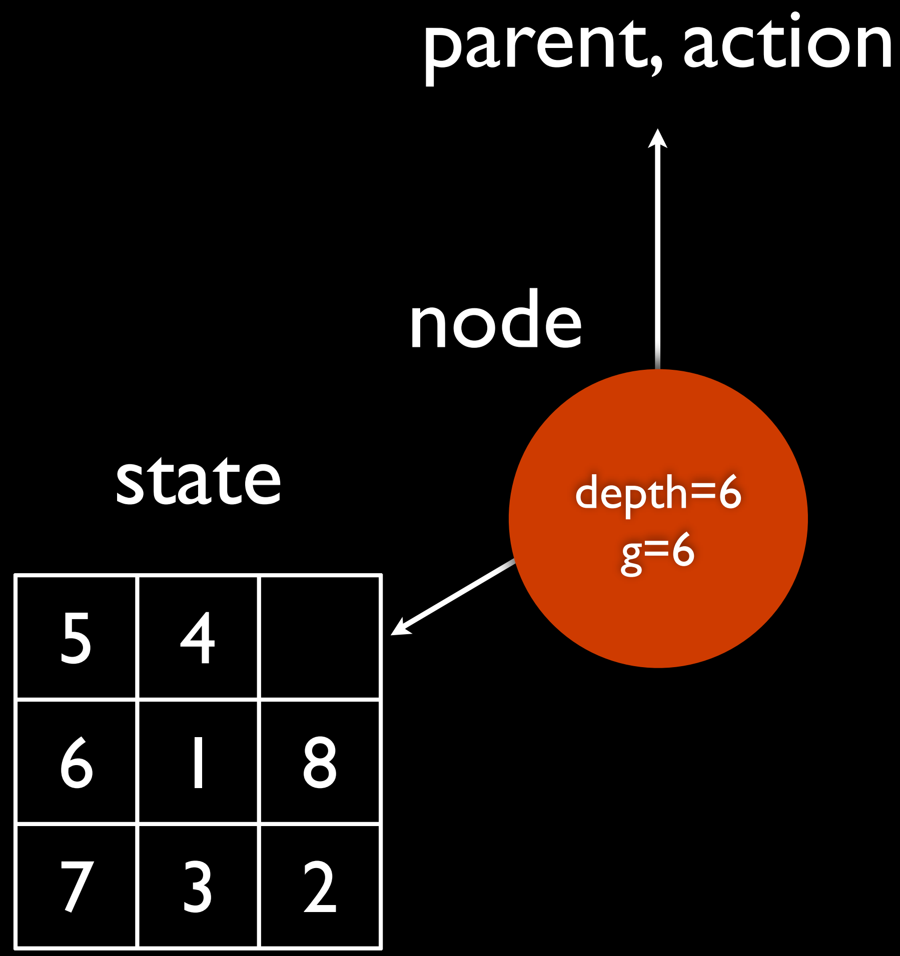


Stating the Problem

- initial state: at Arad
- successor function: succ(state) = { (action,state,cost), ... }
- goal test: goalp(state) = True if state is goal
- path cost: sum of step costs c(x,a,y) = g(n)
- solution: sequence of actions from initial to goal state

States vs Nodes

- state: representation of a problem configuration
- node: data structure in search graph including
  - state
  - parent node
  - action
  - path cost – g(n)
  - depth



```
(defun make-node (&key state (parent nil)
                             (action nil)
                             (path-cost 0) (depth 0))
  (list state parent action path-cost depth))

(defun node-state (node) (car node))
(defun node-parent (node) (cadr node))
(defun node-action (node) (caddr node))
(defun node-path-cost (node) (cadddr node))
(defun node-depth (node) (car (cddddr node)))
```

OR

```
(defstruct node state
           (parent nil) (action nil) (path-cost 0) (depth 0))
```

Search Strategies

- defined by picking the order of node expansion
- strategies evaluated by:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - 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 (graph)
  - d: depth of least-cost solution
  - m: maximum depth of the state space (may be ∞)

Uninformed Search

- breadth first
- uniform cost
- depth first
- depth limited
- iterative deepening

```lisp
(defun general-search (initial-state successor goalp
                       &key (enqueue #'enqueue-LIFO)
                            (key #'identity))
  (let ((fringe (make-q :enqueue enqueue :key key)))
    (q-insert fringe (list (make-node :state initial-state)))
    (tree-search fringe successor goalp)))
```

```
(defun tree-search (fringe successor goalp)
  (unless (q-emptyp fringe)
    (let ((node (q-remove fringe)))
      (if (funcall goalp (node-state node))
        (action-sequence node)
        (tree-search (q-insert fringe (expand successor node))
                     successor goalp))
      )))
```

```
(defun expand (successor node)
  (let ((triples (funcall successor (node-state node))))
    (mapcar (lambda (action-state-cost)
              (let ((action (car action-state-cost))
                    (state (cadr action-state-cost))
                    (cost (caddr action-state-cost)))
                (make-node :state state
                           :parent node
                           :action action
                           :path-cost (+ (node-path-cost node)
                                         cost)
                           :depth (1+ (node-depth node)))
                ))
            triples)
    ))
```

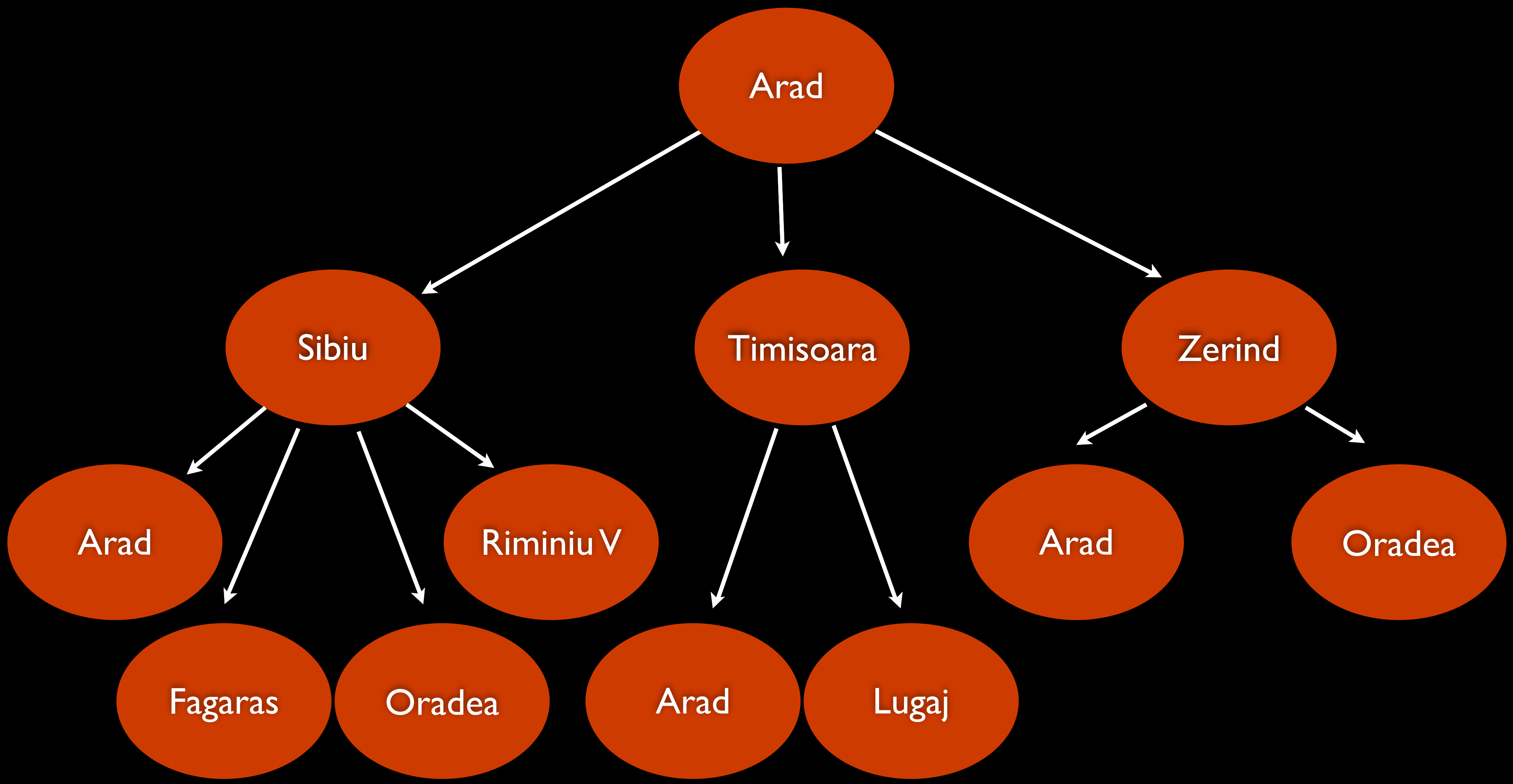
```
(defun action-sequence (node &optional (actions nil))
  (if (node-parent node)
    (action-sequence (node-parent node)
                     (cons (node-action node) actions))
    actions
    ))
```

```lisp
(defstruct node
  (state nil)
  (parent nil)
  (action nil)
  (path-cost 0)
  (depth 0))

; implicitly defines
; (defun make-node (&key (state nil) (parent nil) (action nil)
;                        (path-cost 0) (depth 0))...
; (defun node-state (node)...
; (defun node-parent (node)...
; (defun node-action (node)...
; (defun node-path-cost (node)...
; (defun node-depth (node)...
; (defun node-p (x)...

; also, accessors can be used with setf
; (setf (node-path-cost n) 15)
```

Breadth-First Search

- fringe is FIFO queue
- complete (if **b** is finite)
- time complexity = $O(\mathbf{b}^{d+1})$
- space complexity = $O(\mathbf{b}^{d+1})$ — keeps all nodes in memory
- optimal if unit cost

Searching the Tree (Graph)



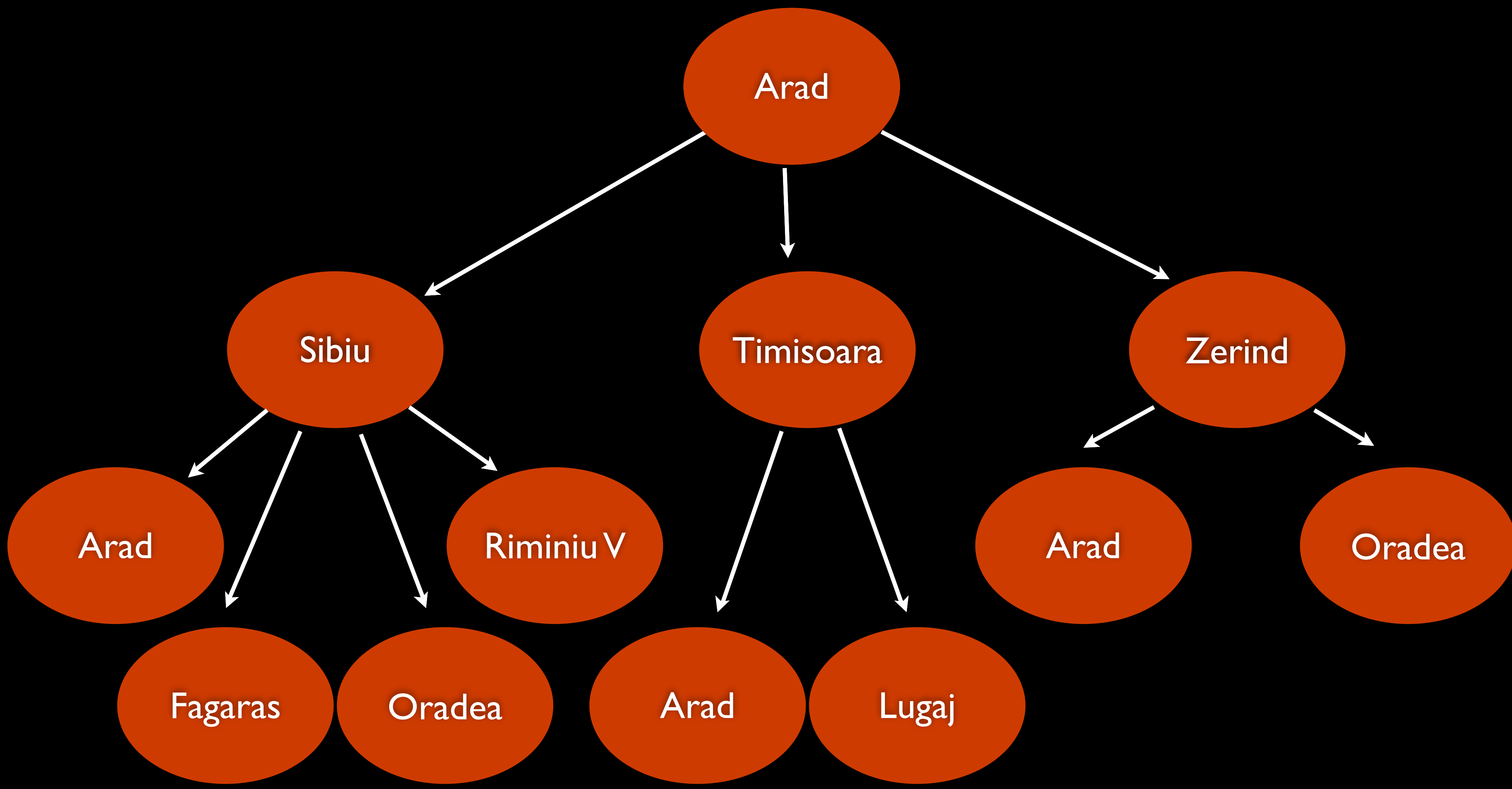
Uniform-Cost Search

- fringe = queue ordered by path cost, g(n)
- equivalent to breadth-first if step costs are all equal
- complete if step costs are non-negative
- time & space complexity: similar to breadth-first
- optimal because nodes are expanded in increasing order of path cost, g(n)

Depth First Search

- fringe is LIFO queue (stack)
- incomplete — fails in infinite-depth spaces or spaces with loops
  - modify code to avoid repeated states
  - complete in finite spaces
- time complexity is $O(b^m)$ — terrible if m >> b, but often much faster than breadth-first
- space complexity is $O(bm)$ — linear!!!
- optimal — no. why?

Searching the Tree (Graph)



Other Uniformed Searches

- depth-limited is depth-first with a cutoff (nodes at maximum depth have no successors)
- iterative deepening search is iterative calls to depth-limited search, increasing depth cutoff each time
- what's the overhead?
- is it complete? what are the time and space complexities? is it optimal?

Eliminating Repeated States

- failure to detect repeated states turns linear problem into exponential one
- fixing this turns tree search into graph search

```
(defun general-search (initial-state successor goalp
                       &key (samep #'eql)
                            (enqueue #'enqueue-LIFO)
                            (key #'identity))
  (let ((fringe (make-q :enqueue enqueue :key key)))
    (q-insert fringe (list (make-node :state initial-state)))
    (graph-search fringe nil successor goalp samep)
    ))
```

```
(defun graph-search (fringe closed successor goalp samep)
  (unless (q-emptyp fringe)
    (let ((node (q-remove fringe)))
      (cond ((funcall goalp (node-state node))
             (action-sequence node))
            ((member (node-state node) closed
                     :test samep :key #'node-state)
             (graph-search fringe closed
                           successor goalp samep))
            (t (let ((successors (expand successor node)))
                 (graph-search (q-insert fringe successors)
                               (cons node closed)
                               successor goalp samep)))
            ))
    ))
```