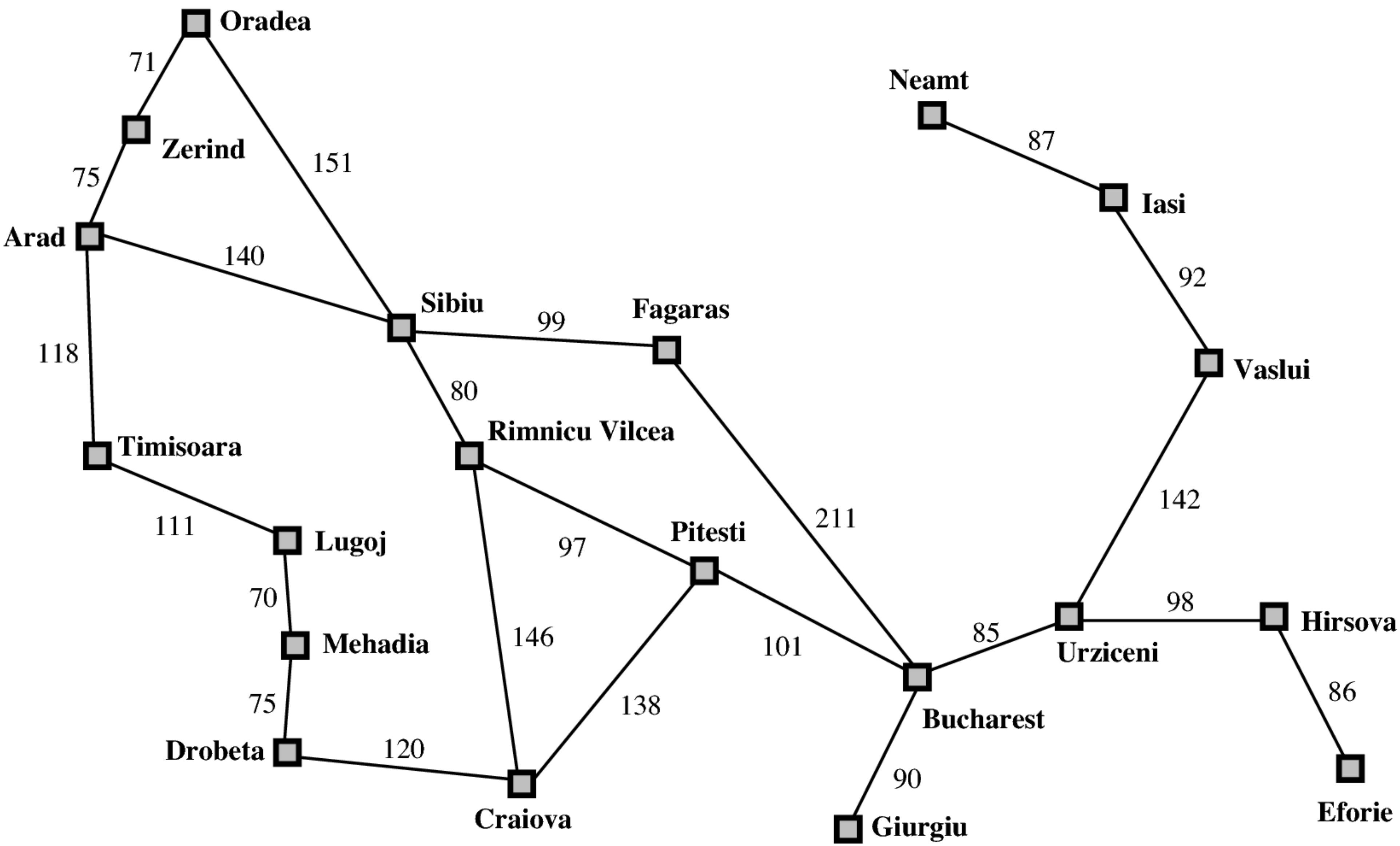
Artificial Intelligence

Information Systems and Machine Learning Lab (ISMLL)
Tomáš Horváth

16rd November, 2011

Informed Search and Exploration

Example (again)



Informed strategy

- we use a problem-specific knowledge beyond the definition of a problem itself
- evaluation function f(n)
  - the node with <u>the lowest f(n)</u> will be selected <u>first</u>
  - BEST-FIRST search
- <u>heuristic function</u> h(n)
  - the estimated cost of the cheapest path from the node n to a goal node
  - somehow imparts an additional knowledge
  - if n is a goal node, then h(n) = 0

An example heuristic function

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	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

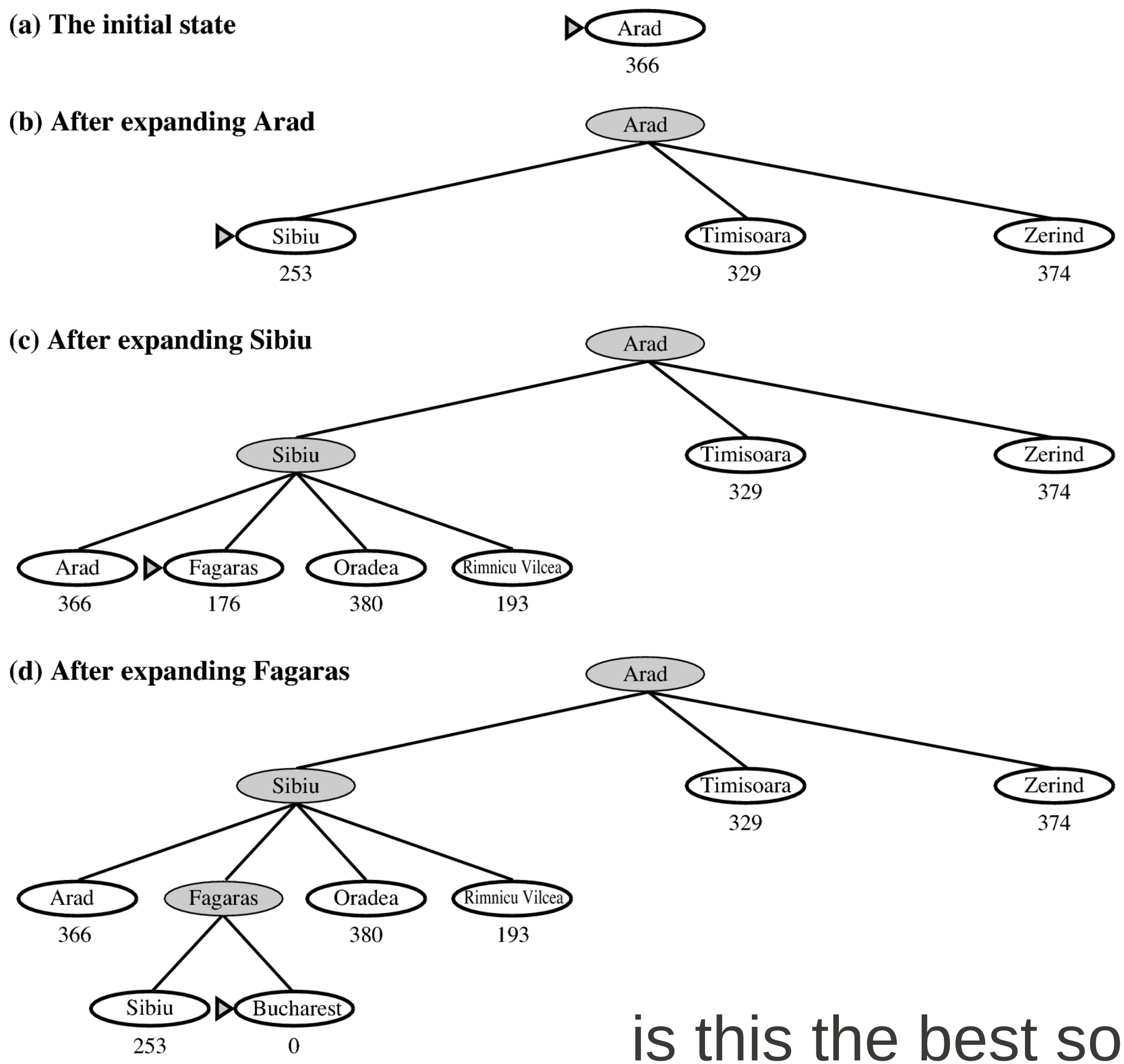
- if it correlates with actual road distances then it is a useful heuristic.

Greedy best-first search

- expand the node that is closest to the goal
- $f(n) = h(n)$

- After seeing an example, try to answer
  - Is this search optimal?
  - What are the drawbacks?
  - What complexity does it have?

Greedy best-first search



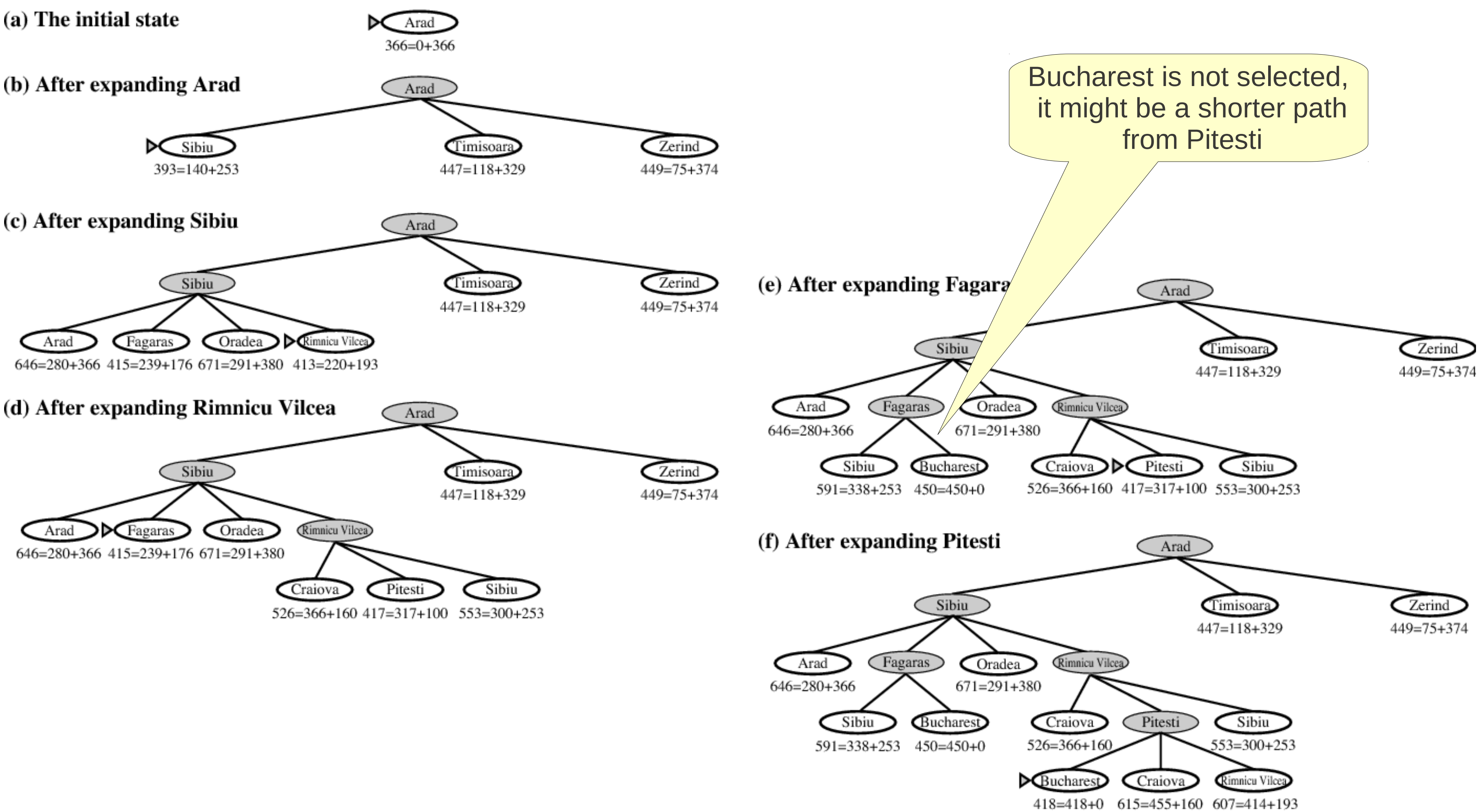
is this the best solution?

- what about the way from Iasi to Fagaras?

A* search

- $f(n) = g(n) + h(n)$
  - cost for reach the node + cost to get to the goal
  - estimated cost of the cheapest solution through n
- <u>admissible heuristic</u> h(n)
  - never overestimates the cost to reach the goal
  - **Is the straight-line distance admissible?**
- A* is optimal
  - if it is used with TREE-SEARCH and
  - if h(n) is admissible
    - **How can it be proved?**

A* example



(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
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
(d) After expanding Rimnicu Vilcea
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
(e) After expanding Fagaras
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
(f) After expanding Pitesti
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
Bucharest is not selected, it might be a shorter path from Pitesti

A* proof (tree-search)

- *since g(n) is the exact cost and h(n) is admissible, f(n) never overestimates*
- suboptimal goal G2, cost C* for optimal solution
  - h(G2) = 0
    - f(G2) = g(G2) + h(G2) = g(G2) > C*
- consider a node n on an optimal path
  - if a solution exists, n exists too
  - h(n) does not overestimate
    - f(n) = g(n) + h(n) <= C*
      - f(n) <= C* < f(G2)
        - G2 will not be expanded and A* must return an optimal solution

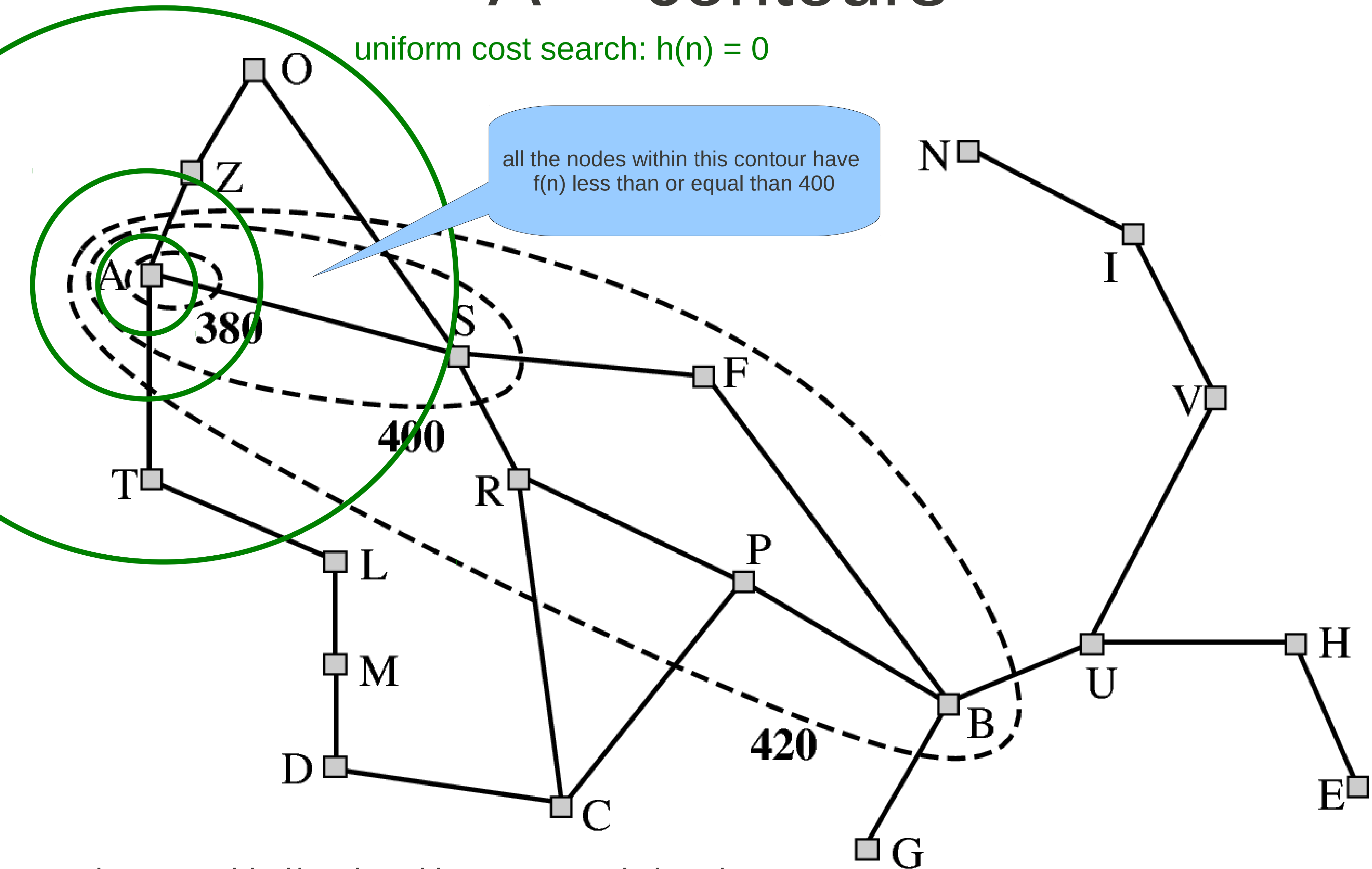
A* (graph-search)

- *graph-search can discard the optimal path to a repeated state if it is not the first one generated*
  - discarding the more expensive of any two paths found to the same node
    - such an extra bookkeeping is messy, even if guarantees optimality
  - ensuring that the optimal path to any repeated state is always the first one followed
    - as is in the case of uniform-cost search
    - h(n) needs to be <u>consistent</u> (monotone)
      - for every *n* and every successor *n'* of *n* generated by any action *a*
        - $h(n) <= c(n,a,n') + h(n')$

A* (graph-search)

- *n*, *n'* and the closest goal to *n* form a triangle (triangle inequality)
  - every consistent heuristic is also admissible
- *if h(n) is consistent then the values of f(n) along any path are nondecreasing*
  - $g(n') = g(n) + c(n,a,n')$
  - $h(n) <= c(n,a,n') + h(n')$
    - $f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') >= g(n) + h(n) = f(n)$
- *A* using graph-search is optimal if h(n) is consistent*
  - sequence of nodes expanded by A* using graph-search is in nondecreasing order of f(n)
    - the first goal node selected for expansion must be optimal since all later nodes will be at least as expensive

A* - contours



nodes are added/explored in a concentric bands

A* - large-scale problems

- expand no nodes with $f(n) > C^*$
  - such nodes are <u>pruned</u>
- however, the number of nodes within the goal contour is for most problems still exponential
  - unless $|h(n) - h^*(n)| <= O(\log h^*(n))$
    - $h^*(n)$ is the true cost of getting from n to the goal
  - keeps all generated nodes in the memory
    - as all graph-search algorithms
  - *impractical to insist on finding an optimal solution*
    - variants of A* for finding suboptimal solutions quickly

Memory-bounded heuristic search

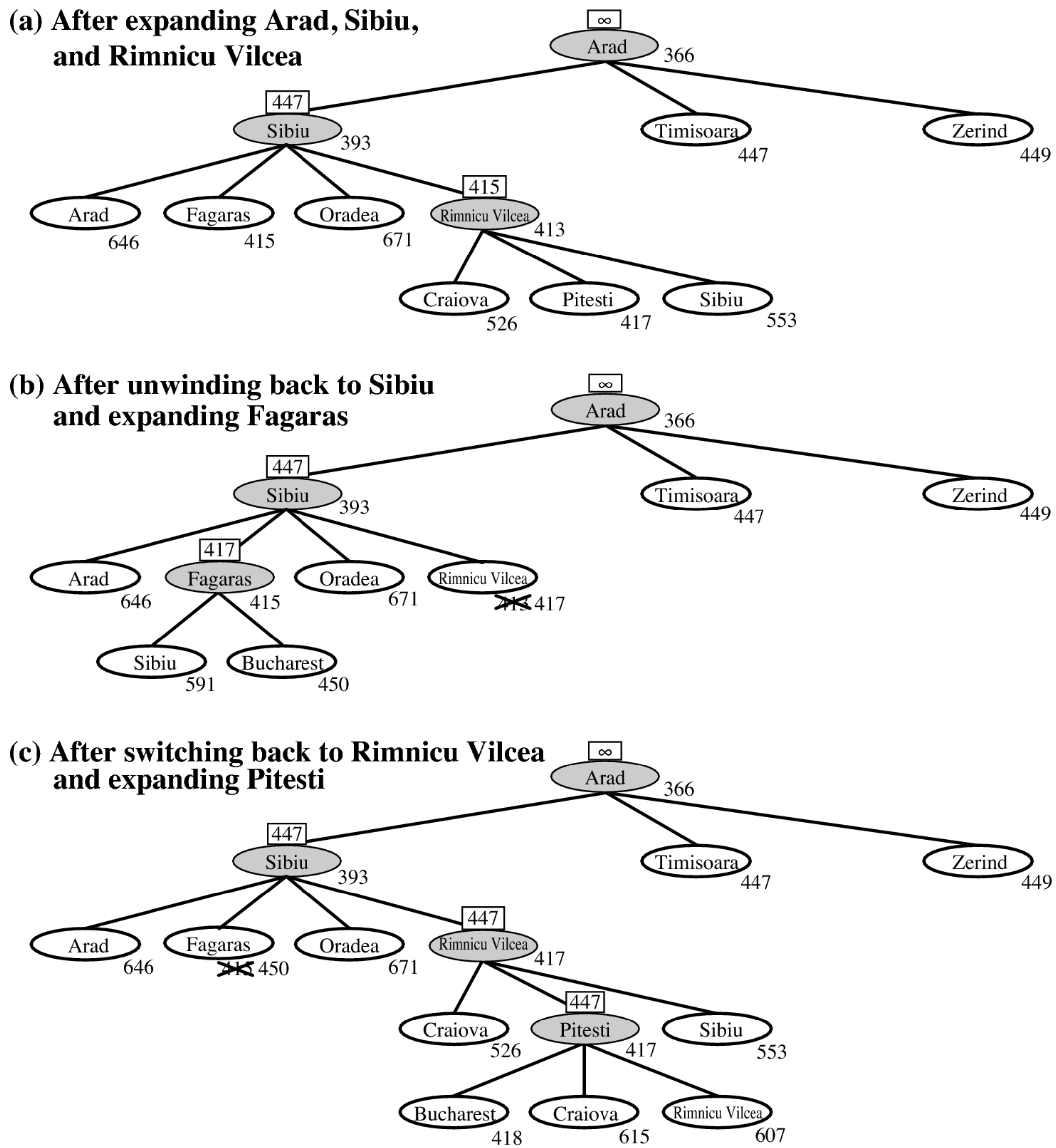
- we can simply adapt the idea of iterative-deepening (IDA*)
  - use the smallest f-cost of any node that exceeded the cutoff in the previous iteration as a new cutoff

- RBFS
- MA*

Recursive best-first search

- a simple recursive algorithm but
  - it keeps track of the f-value of the best alternative path available from any ancestor of the current node
  - if the current node exceeds the limit the recursion <u>unwinds</u> back to the alternative path
    - replaces the f-value of each node along the path with the best f-value of its children
- remembers the f-value of the best leaf in the forgotten subtree

Recursive best-first search



Recursive best-first search

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action)  into successors
  if successors is empty then return failure, ∞
  for each s in successors do  /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min( f_limit, alternative))
      if result ≠ failure then return result
```

IDA* and RBFS

- like A*, is optimal if h(n) is admissible
- excessive node regeneration
- space complexity is linear in depth of the deepest optimal solution
- hard to characterize it's time complexity
  - they may explore the same state many times
- IDA* and RBFS suffers from <u>too little memory</u>
  - it seems sensible to use all available memory

Simplified memory-bounded A*

- proceeds like A* until the memory is full
- if the memory is full SMA* drops the worst leaf node (with the highest value)
  - however, the ancestor of a forgotten subtree knows the value of the best path in that subtree
  - SMA* regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten
- SMA* is complete if the depth of the shallowest goal is less than the memory size
- extra time needed for repeated regeneration

Simplified memory-bounded A*

- **What if all the leaf nodes have the same value?**
  - it might select the same node for deletion and expansion
  - expanding the newest best leaf and deleting the oldest worst leaf
  - the same node if there is only 1 leaf
    - the current search tree is a single path from root to leaf that fills all of the memory

A short note on heuristics

- 8-puzzle example
  - start: (7, 2, 4, 5, null, 6, 8, 3, 1), goal: (null, 1, 2, 3, 4, 5, 6, 7, 8)
  - h1 = the number of misplaced tiles
  - h2 = the sum of the distances of the tiles from their goal position (Manhattan distance)
  - **Which one is better?**
- effective branching factor b*
  - that a uniform tree of depth d would have when containing N+1 total nodes generated
  - $N+1 = 1 + (b^*) + (b^*)^2 + \ldots + (b^*)^d$
  - a well-designed heuristic would have b* close to 1
    - test it on a small set of of problems generated which gives us a good guide of the usefulness of a given heuristic

A short note on heuristics

	Search Cost			***Effective branching factor***		
d	***IDS***	***A*(h1)***	***A*(h2)***	***IDS***	***A*(h1)***	***A*(h2)***
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
10	47127	93	39	2.79	1.38	1.22
14	--	539	113	--	1.44	1.23
20	--	7276	676	--	1.47	1.27
24	--	39135	1641	--	1.48	1.26

A short note on heuristics

- h2 is better than h1 for an 8-puzzle problem
  - **Is it always better?**
- h2 <u>dominates</u> h1
  - if for any node n, h2(n) >= h1(n)
    - using h2 will never expand more nodes than h1
      - every node with f(n) < C* will surely be expanded
      - every node with h(n) < C* - g(n) will surely be expanded
      - since h2 is at least as big as h1 for all nodes, every node surely expanded with h2 will also be surely expanded with h1
  - it is always better to use heuristic with higher values
    - just if the heuristic does not overestimate

Local search and Optimization

- sometimes the path to the goal is irrelevant
  - e.g. 8-queen
- Local search algorithms
  - not systematic
    - use a single current state
    - generally, move only to neighbors
    - typically, the paths are not retained
  - use very little memory
  - often find reasonable solutions in large spaces
- Optimization problems
  - find the best state according to an objective function

Hill-climbing search

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

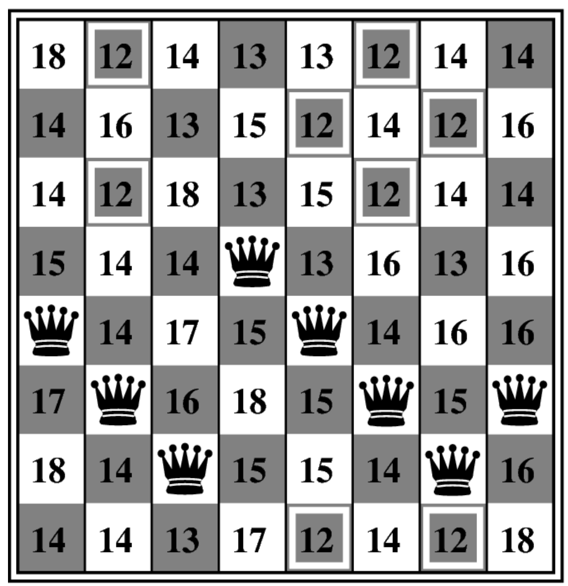
    current ← MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor ← a highest-valued successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
        current ← neighbor

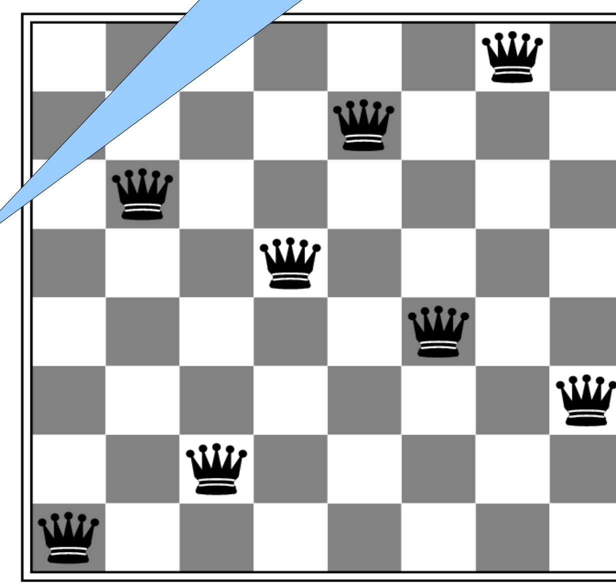
heuristic function: number of pairs of queens attacking each other

complete-state formulation:

- each state has 8 queens on the board

- successor function returns all possible states generated by moving a single queen to another position in the same column (8 x 7 = 56 successors)

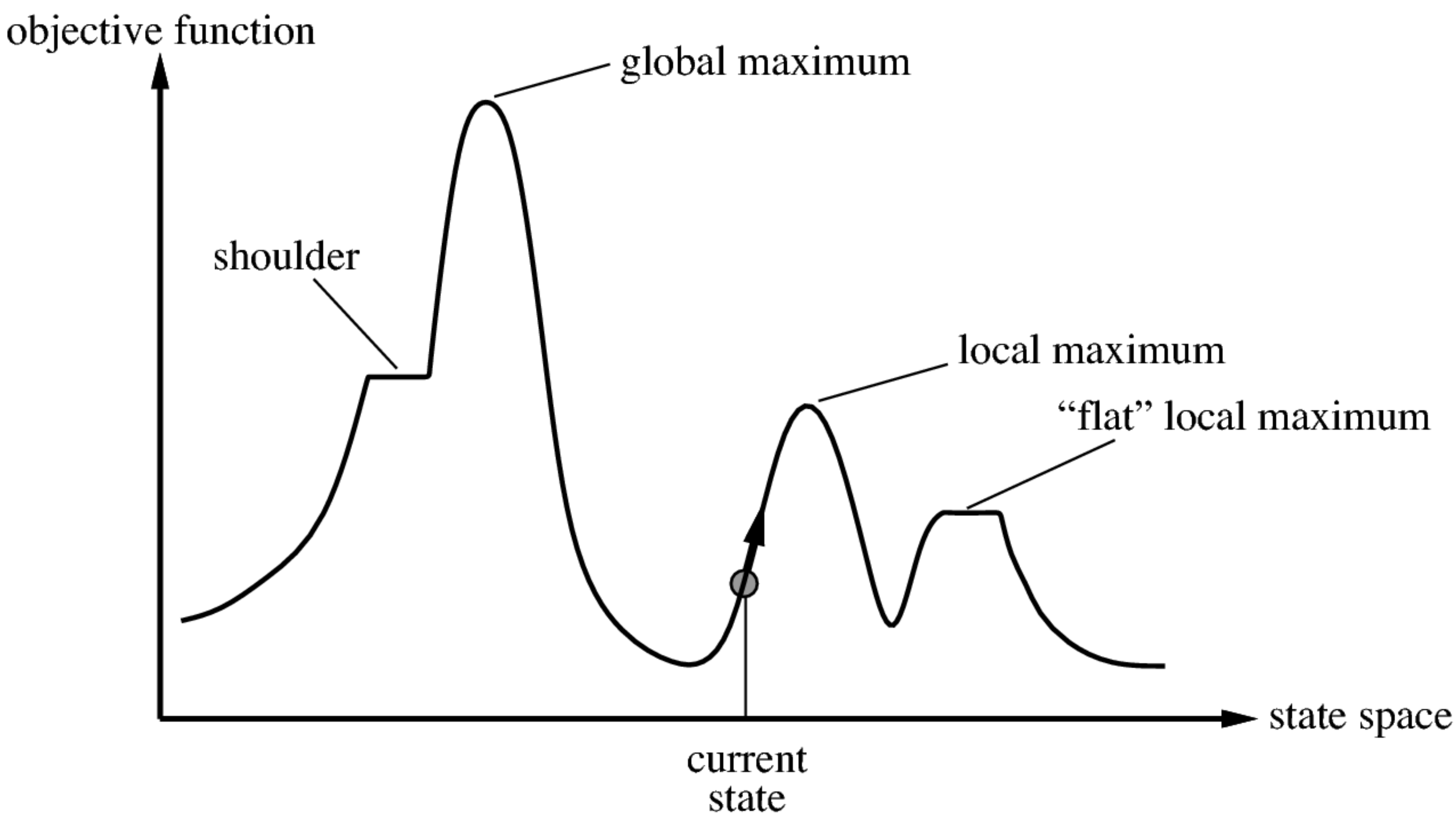




- **What are the drawbacks of this algorithm?**

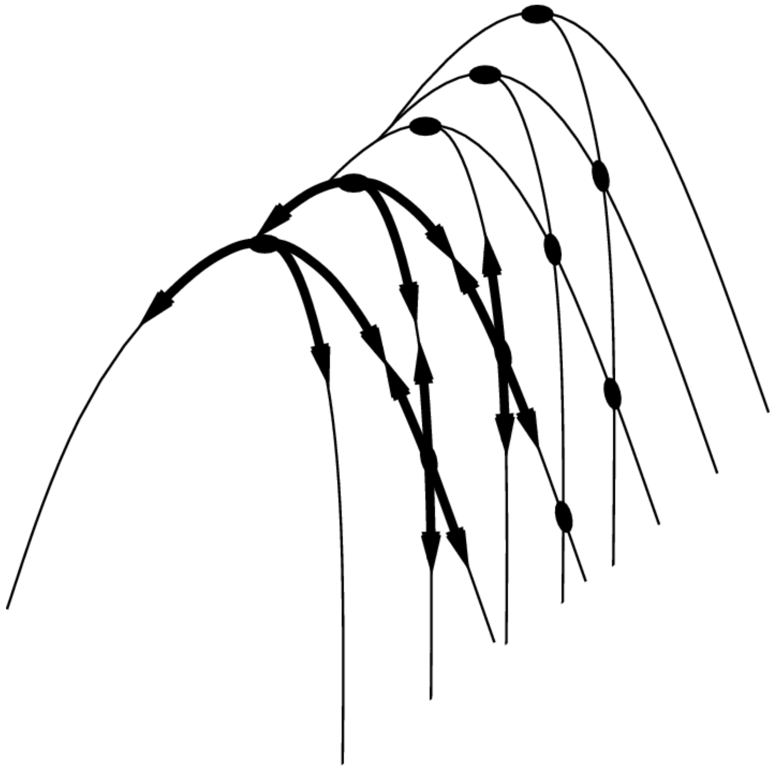
Hill-climbing search

- The state space landscape



Hill-climbing search

- Sideways moves
  - allow when a plateau is reached



Hill-climbing search

- variations
  - stochastic
    - chooses at random from among the uphill moves
  - first-choice
    - stochastic HC by generating successors randomly until one is generated that is better than the current state
  - random-restart
    - perform a series of HC with randomly generated initial states

Simulated Annealing

- HC never makes “downhill” move
  - it can stuck in the local maximum
- random-walk
  - choosing a successor uniformly at random from the set of successors
  - complete but inefficient
- it seems reasonable to combine HC and RW
  - simulated annealing
    - motivated by a process of annealing in metallurgy which is a process to temper or harden metals and glass

Simulated Annealing

for lowering T

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"

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

if the move improves the current situation it is always accepted

otherwise, the move is accepted with some probability exponentially decreasing in time (as the temperature decreases)

if T is lowered slowly enough then the algorithm will find the global optimum

Local Beam Search

- keep track on k states
- begin with k random states
- at each step
  - all the successors of the k states are generated
    - if any is a goal, then halt
    - select the k best successors and repeat
- **how it differs from running k random restarts in sequence?**

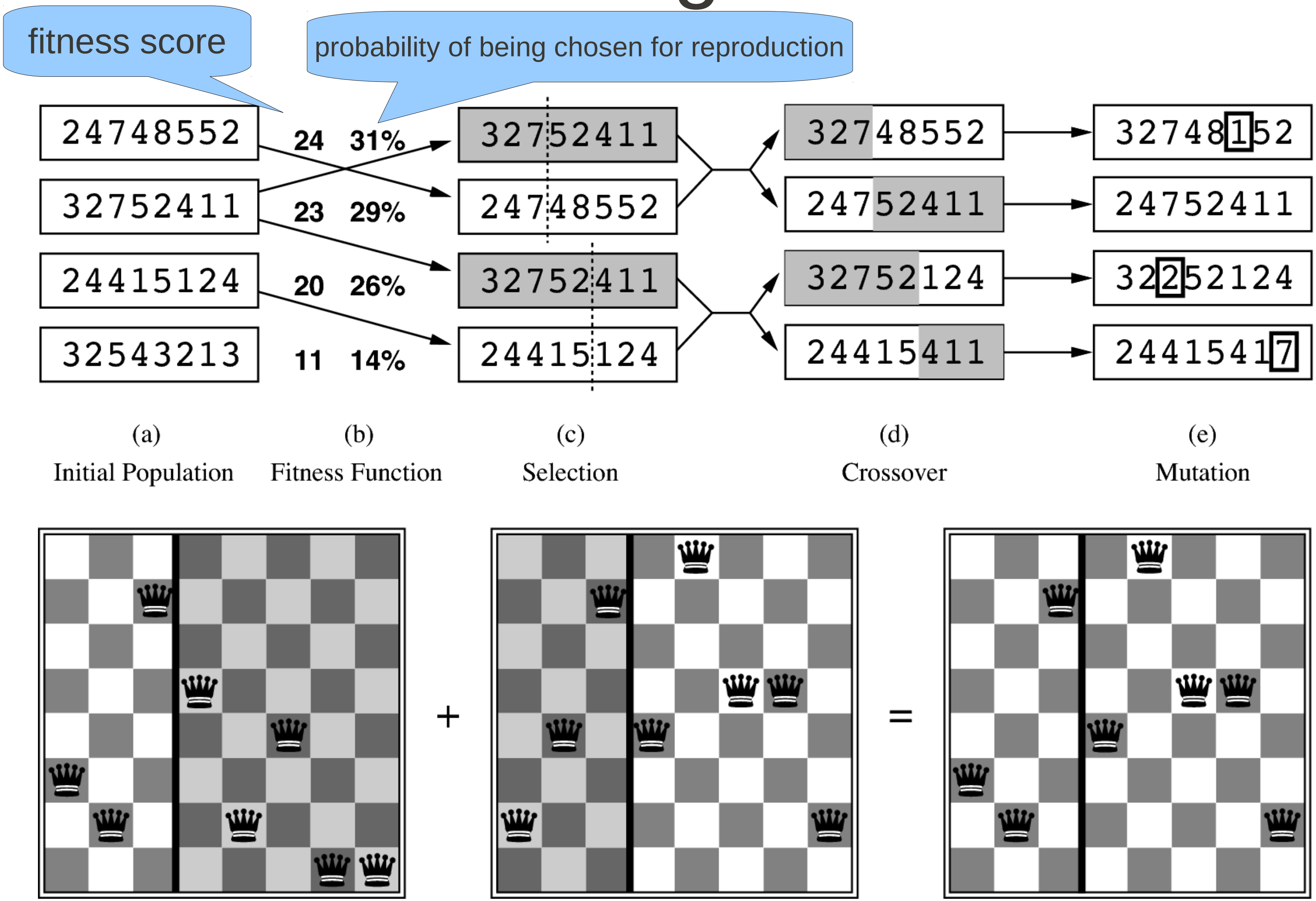
Local Beam Search

- useful information is passed among the k parallel search threads
  - e.g. 1 state generates several good successors while other states generates bad successors
  - moves the resources to prospective areas of the search space
- the k successors can quickly become concentrated in a small area of the space
  - stochastic beam search
    - choose successors at random
    - the probability of choosing a successor grows with its value
      - a “natural” selection

Genetic Algorithms

- a variant of stochastic beam search
  - successor states are generated by combining two parent states
  - analogy to natural selection
- begins with the randomly generated <u>population</u>
  - an <u>individual</u> is represented by a string over a finite alphabet
    - 0-1 or digits (the two encodings behave differently)
    - <u>fitness function</u>
      - e.g. the number of nonattacking pairs of queens

Genetic Algorithms



Genetic Algorithms

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Genetic Algorithms

as in SA, at the beginning larger steps are taken

the population is quite diverse at the beginning

- the crossover operation has the ability to combine large blocks that have evolved independently
  - doing crossover in a random order, however, makes no advantage
  - <u>schema</u>
    - for example 246*****
    - instances of the schema
    - makes sense, if adjacent bits are related each other, i.e. when schemas correspond to meaningful components of a solution

if the average fitness of instances of a schema is above the mean, then the number of instances of the schema within the population will grow over the time

Local search in continuous spaces

- none of the algorithms before can handle continuous spaces
  - the successor function would return infinitely many cases
  - example - we have to place 3 airports on “our” map such that the sum of squared distances from each city to the closest airport is minimized
    - coordinates (x1,y1), (x2,y2), (x3,y3)
      - six <u>variables</u> (six dimensional space)
      - objective function f(x1,y1,x2,y2,x3,y3) is tricky to express
      - **how could we apply e.g. hill climbing?**
        - can we discretize the neighborhood of the states (move only one airport in x or y direction by +- delta)?

can we apply SA directly by generating random vectors?

Local search in continuous spaces

What does the gradient represent?

- gradient ascent algorithms
  - $\nabla f = (\partial f/\partial x1, \partial f/\partial y1, \partial f/\partial x2, \partial f/\partial y2, \partial f/\partial x3, \partial f/\partial y3)$
    - we can compute the gradient only locally
  - perform steepest-ascent hill climbing
    - $x\_new = x\_old + \alpha \nabla f(x)$
    - α is a “small” constant
      - if too small, many steps are needed
      - if too large, it can overshoot the maximum
    - line search
      - doubling α until f starts to decrease
      - this point becames the new state

Local search in continuous spaces

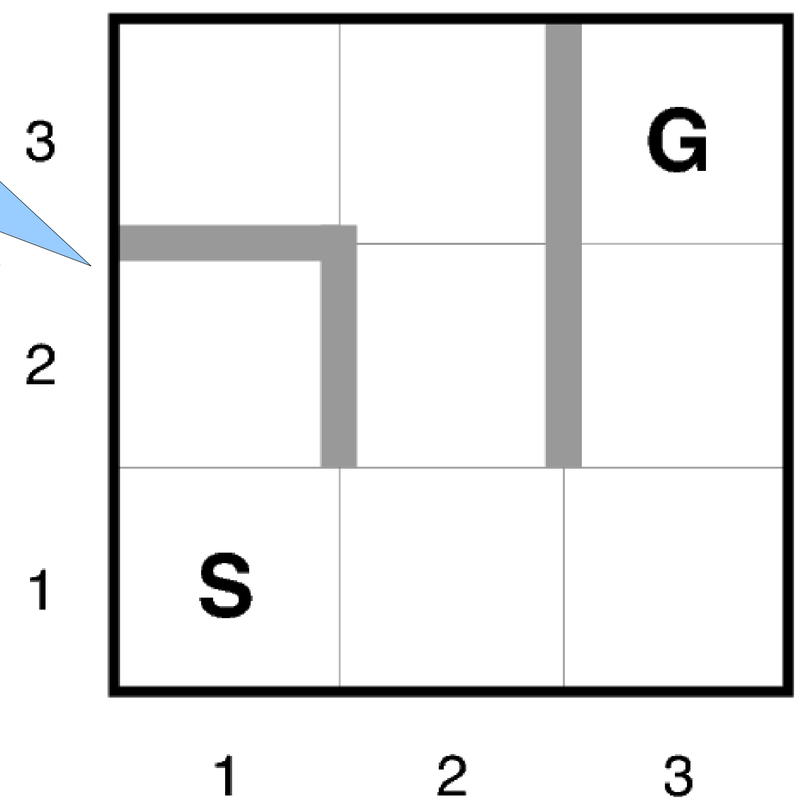
- sometimes an objective function is not available in a differentiable form
  - for example is computed by some other (external) tools
  - in this case use empirical gradient
    - evaluating the response to small increments and decrements in each coordinate
- there are several variations of the gradient ascent algorithm

On-line search

- can be solved only by an agent executing actions rather than by a purely computational process
- an agent knows just
  - ACTION(s)
  - c(s,a,s')
  - GOAL_TEST(s)
- an agent can expand only the node that it physically occupies!

only if s' is known

get from S to G without knowing the environment



On-line DFS agent

works only if the actions are reversible

function ONLINE-DFS-AGENT(s') **returns** an action
inputs: s', a percept that identifies the current state
persistent: $result$, a table indexed by state and action, initially empty
$untried$, a table that lists, for each state, the actions not yet tried
$unbacktracked$, a table that lists, for each state, the backtracks not yet tried
s, a, the previous state and action, initially null

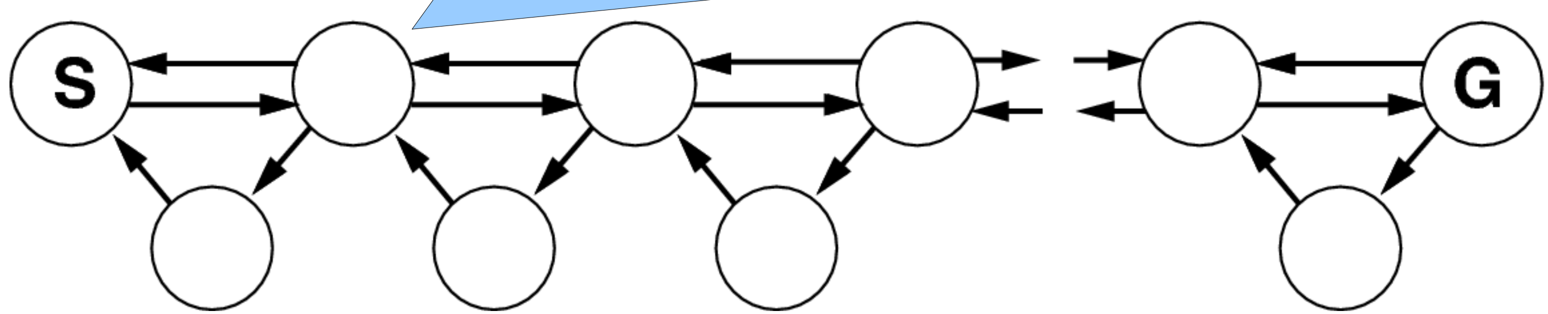
if GOAL-TEST(s') **then return** $stop$
if s' is a new state (not in $untried$) **then** $untried[s'] \leftarrow$ ACTIONS(s')
if s is not null **then**
$result[s, a] \leftarrow s'$
add s to the front of $unbacktracked[s']$
if $untried[s']$ is empty **then**
if $unbacktracked[s']$ is empty **then return** $stop$
else $a \leftarrow$ an action b such that $result[s', b]$ = POP($unbacktracked[s']$)
else $a \leftarrow$ POP($untried[s']$)
$s \leftarrow s'$
return a

We assume a **safely explorable** state space where are no dead-ends, i.e. that some goal state is reachable from any reachable state.

Random Walk

- hill-climbing is already an on-line search (why?)
  - can stuck sitting in the local maximum
    - random restarts cannot be performed since an agent cannot transport itself to a new state
- Random walk
  - selecting at random one of the available actions
    - best if the selected action has been not tried yet
  - very slow

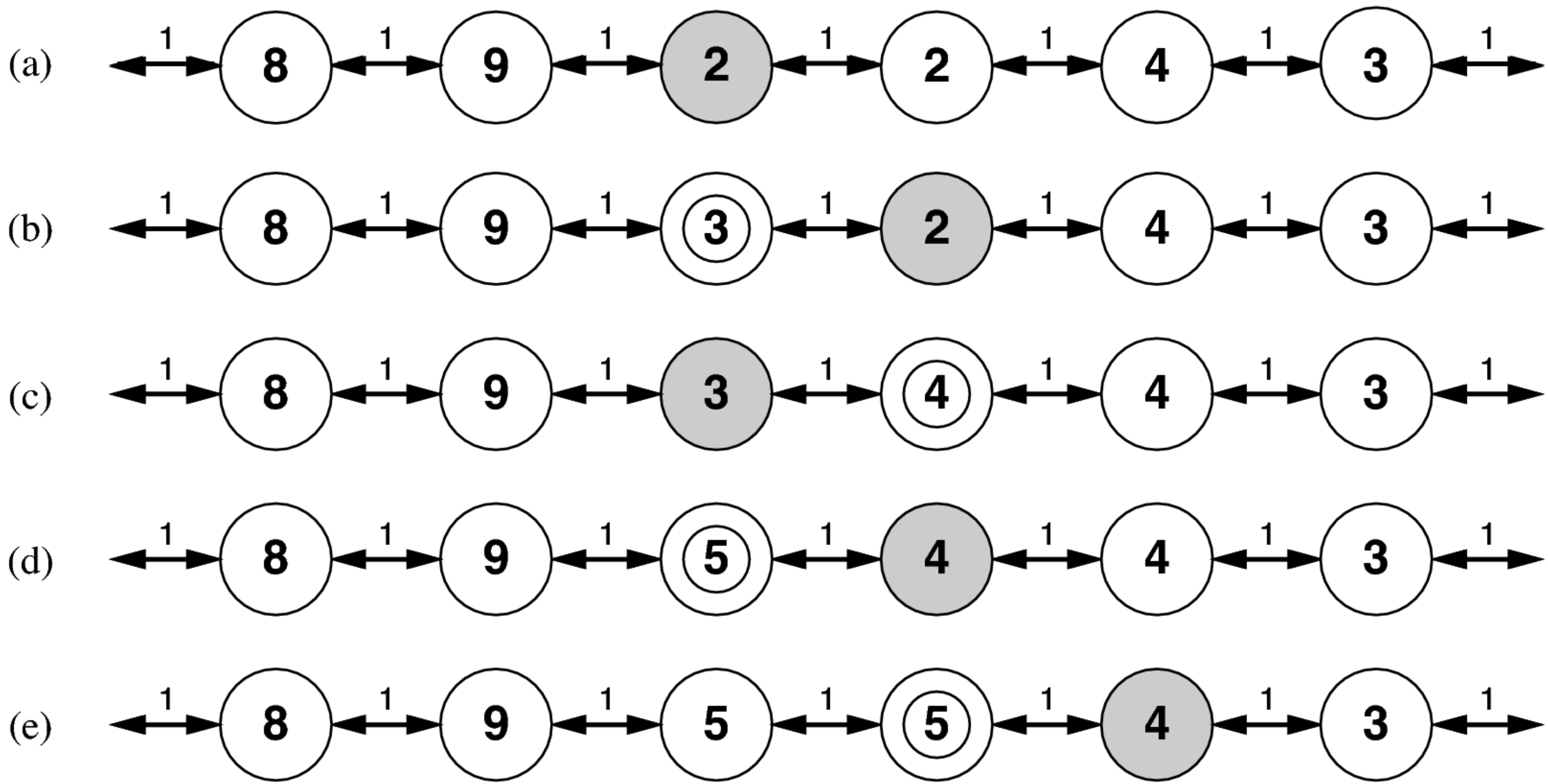
in each step, backward progress is more likely than forward



LRTA* - learning real-time A*

- augmenting HC with memory rather than randomness
  - store the current best estimate H(s) of the cost to reach the goal from each state that has been visited
    - H(s) is updated as the agent gains experience

LRTA*


(a) 1 8 1 9 1 2 1 2 1 4 1 3 1
(b) 1 8 1 9 1 3 1 2 1 4 1 3 1
(c) 1 8 1 9 1 3 1 4 1 4 1 3 1
(d) 1 8 1 9 1 5 1 4 1 4 1 3 1
(e) 1 8 1 9 1 5 1 5 1 4 1 3 1

LRTA*

function LRTA*-AGENT(s') **returns** an action
  inputs: s', a percept that identifies the current state
  persistent: $result$, a table, indexed by state and action, initially empty
    H, a table of cost estimates indexed by state, initially empty
    s, a, the previous state and action, initially null

  if GOAL-TEST(s') **then return** $stop$
  if s' is a new state (not in H) **then** $H[s'] \leftarrow h(s')$
  if s is not null
    $result[s, a] \leftarrow s'$
    $H[s] \leftarrow \min\limits_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(s, b, result[s, b], H)$
  $a \leftarrow$ an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, $result[s', b]$, H)
  $s \leftarrow s'$
  return a

function LRTA*-COST(s, a, s', H) **returns** a cost estimate
  if s' is undefined **then return** $h(s)$
  else return $c(s, a, s') + H[s']$

Thanks for your attention!
Questions?