ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

IT3160E
Introduction to Artificial Intelligence

Chapter 3 – Problem solving

Part 3: problem-solving by searching using more advanced search strategies

Lecturer:
Muriel VISANI
Acknowledgements:
Le Thanh Huong
Tran Duc Khanh
Department of Information Systems
School of Information and Communication Technology - HUST

Content of the course

- Chapter 1: Introduction
- Chapter 2: Intelligent agents
- Chapter 3: Problem Solving
  - Search algorithms, adversarial search
  - Constraint Satisfaction Problems
- Chapter 4: Knowledge and Inference
  - Knowledge representation
  - Propositional and first-order logic
- Chapter 5: Uncertain knowledge and reasoning
- Chapter 6: Advanced topics
  - Machine learning
  - Computer Vision

Outline

- Chapter 3 – part 1: un-informed (basic) algorithms
- Chapter3 - part 2: informed search strategies in graphs
- Chapter 3 – part 3: advanced search strategies
  - Memory-bounded heursitic search
    - Introduction
    - IDA*
    - RBFS
    - SMA*
  - Local search algorithms
    - Hill-climbing search
    - Simulated annealing search
  - Exercise
  - Summary

Goal of this Lecture

Goal	Description of the goal or output requirement	Output division/ Level (I/T/U)
M1	Understand basic concepts and techniques of AI	1.2

Memory-bounded heuristic search

Introduction

Memory-bounded heuristic search

- Recall from the last lecture:
  - A* is a very good algorithm, but its main drawback is the space cost (exponential in the path length)
    - In large search spaces, might be impossible to use on regular compters
- Memory-bounded heuristic search are solutions to this problem, maintaining completeness and optimality
  - Iterative-deepening A* (IDA*)
    - Close to IDS, but with cutoff based on the the *f*-cost (*g*+*h*) instead of depth
  - Recursive best-first search (RBFS)
    - Recursive algorithm that attempts to mimic standard best-first search with linear space
  - (simple) Memory-bounded A* ((S)MA*)
    - Drop the worst-leaf node when memory is full

Memory-bounded heuristic search

IDA*

Iterative Deeping A* (IDA*)

- IDA* is a **tree-search** algorithm
  - Can be applied to graphs only after they're converted to trees
    - An example of that is given in an exercise, at the end of this lecture

- It is a variant of Iterative Deepening (depth-first) Search (IDS) that borrows the idea of A* to use an evaluation function in the form $f(n)=g(n)+h(n)$
  - $g(n)$: true cost from the root to node n
  - $h(n)$: heuristic cost from the node n to the goal

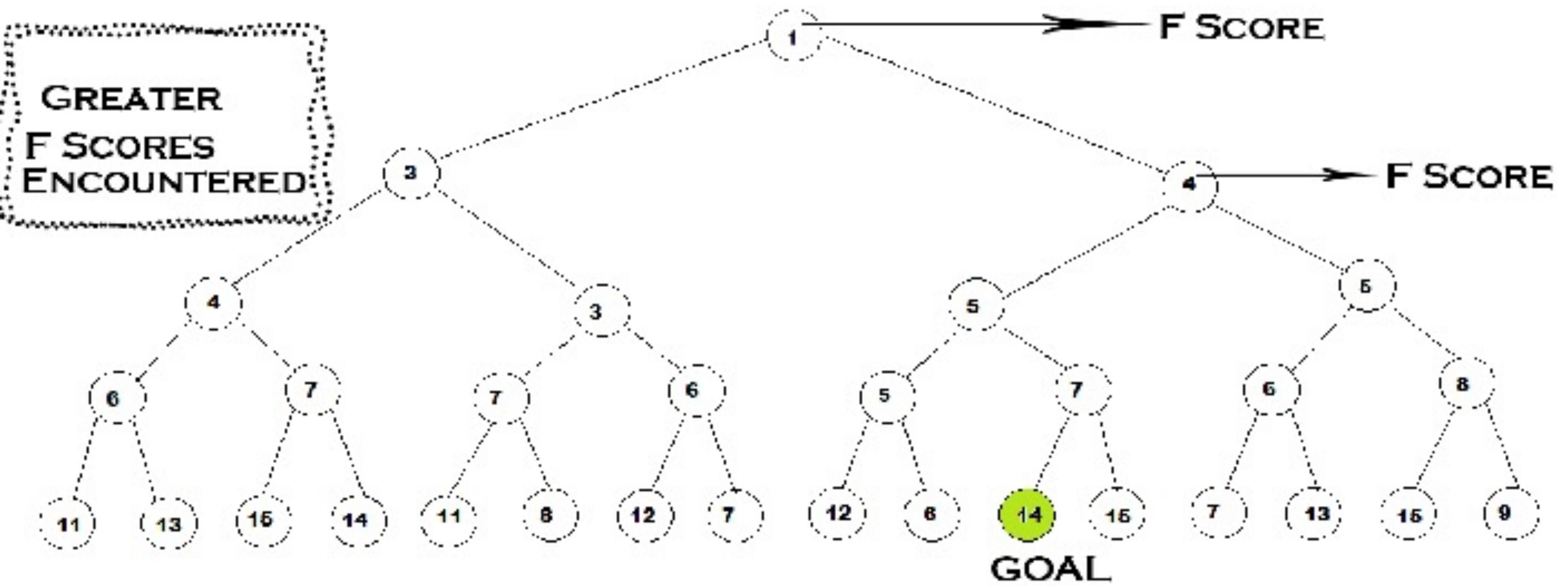
- Combines advantages of IDS and A*:
  - It's a depth-first search algorithm=> space cost lower than A*
  - Concentrates on exploring the most promising nodes => doesn't go to the same depth everywhere in the search tree => more time efficient than IDS

Iterative Deeping A* (IDA*)

- Iterative Deeping version of A* (general idea)
  - Define a threshold, θ_1 = *f(root_node)*
  - *t<-1*
  - Then, at each iteration *t*
    - Expand the non-goal nodes *n* in a depth-first way, with stopping criterion *f()* > θ_t
      - *I.e.* If *f(n)* > θ_t then **prune** the node *n* (do not expand it), and put this node in the fringe
      - If any goal node in the fringe, return the goal node with lowest value of *f()*
        - Simple strategy in case of ties: pick randomly; other strategies are possible
      - Set θ_{t+1} = minimum *f()* of all nodes in the fringe
      - t <- t+1

Iterative Deeping A* (IDA*)

- **To save space**, between iterations, IDA* retains only a single number: θ
  - ⇒ Suffers from too many node re-generations in general
  - ⇒ Costly, in terms of **time complexity**
- Example:
  - N.B. in this example, the f() values have been pre-computed from *f(n)=g(n)+h(n)*
  - N.B. Initially, threshold=1



Iterative Deeping A* (IDA*)

- **Recall**:
  - *b:* branching factor; *d:* depth of the **shallowest** goal node
- IDA* is complete and optimal under the same conditions as A* **tree** search:
  - *h* is admissible
  - All arc costs > epsilon > 0
  - Finite branching factor
- Time complexity: same as IDS: $O(b^d)$ – in the worst case
  - In practice, it strongly depends on the number of different values that the *f()* value can take on
- Space complexity is decreased to O(bd) – in the worst case
  - More precisely, If δ is the smallest cost, and f^* is the optimal solution cost, then IDA* will require bf^*/δ nodes.

Memory-bounded heuristic search

Recursive best-first search

Recursive best first search example

- Algorithm: similar to recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the attribute *f-limit*
  - *f-limit* keeps track of the best alternative path, **starting from any ancestor** of the current node
    - So that the algorithm can backtrack there, if needed
    - The whole alternative paths **are not kept in memory**, only their *f_limit*

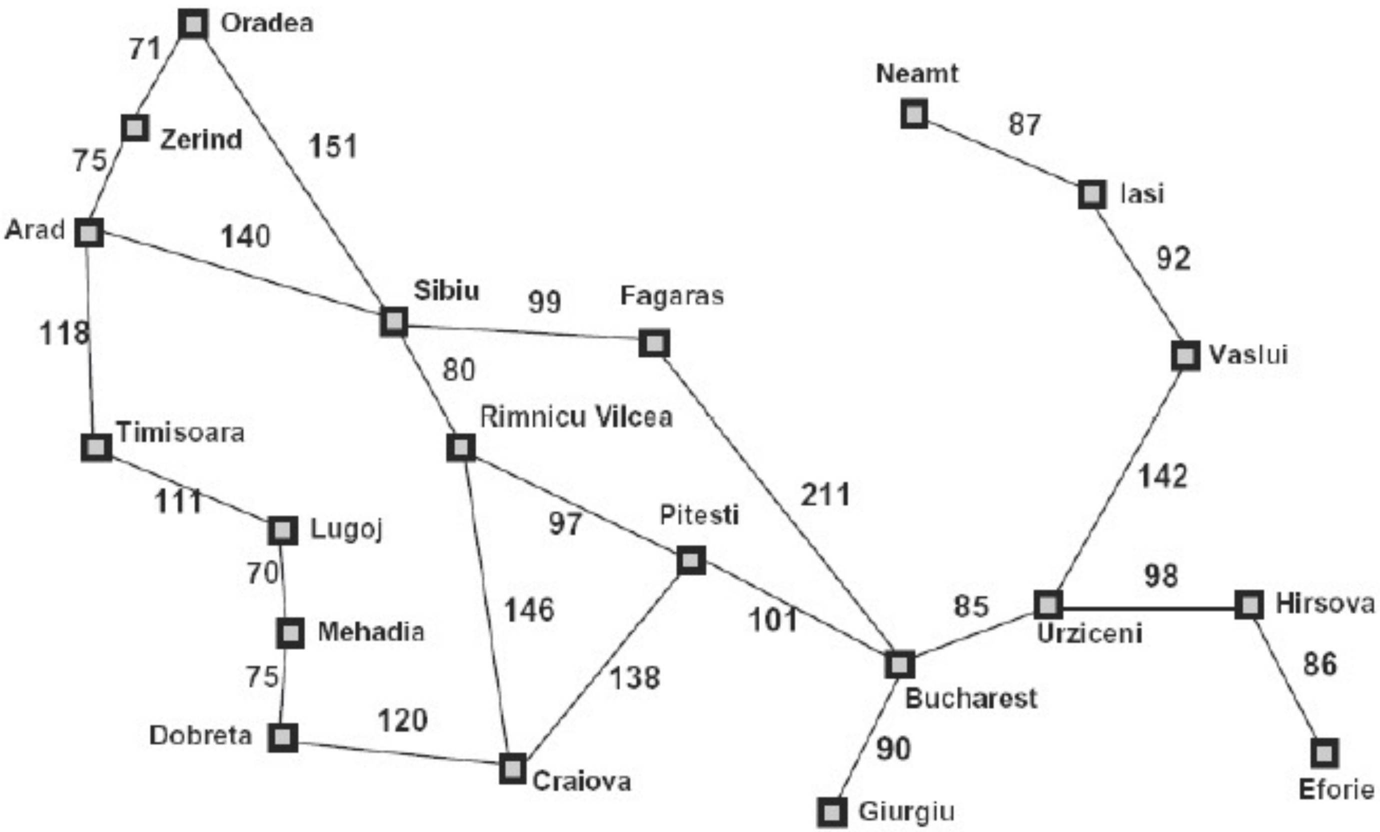
    => When the algorithm backtracks there, then it has to **regenerate** the alternative path
      - Costly, in terms of time complexity
  - For the root, the value of *f-limit* is initially set to +infinity

Recursive best-first search algorithm

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **return** a solution or failure
    return RFBS(*problem*,MAKE-NODE(INITIAL-STATE[*problem*]),∞)

function RFBS(*problem, node, f_limit*) **return** a solution or failure and a new *f-cost* limit
    if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*
    successors ← EXPAND(*node, problem*)
    if *successors* is empty then return failure, ∞
    for each *s* **in** *successors* **do**
        f [*s*] ← max(*g(s)* + *h(s)*, *f* [*node*])
    repeat
        best ← the lowest *f*-value node in *successors*
        if *f* [*best*] > *f_limit* **then return** failure, *f* [*best*]
        alternative ← the second lowest *f*-value among *successors*
        result, *f* [*best*] ← RBFS*(problem, best,* min(*f_limit, alternative*))
        if *result* ≠ failure **then return** *result*

Recursive best first search example (with A*)

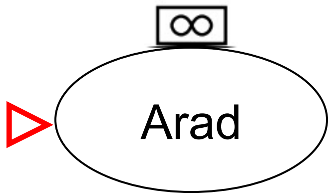


Straight−line distance to Bucharest

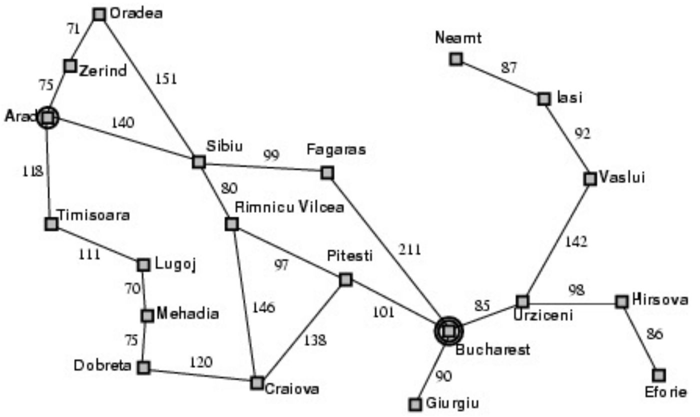
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	~~178~~ 176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	~~98~~ 100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Recursive best first search example (with A*)

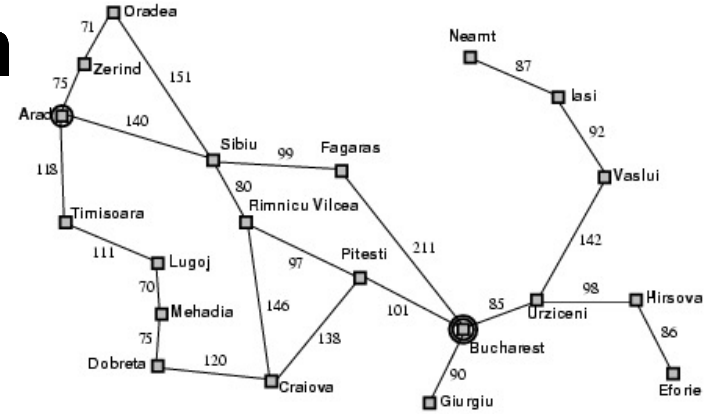
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	~~178~~ 176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	~~98~~ 100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



366 = 0 + 366

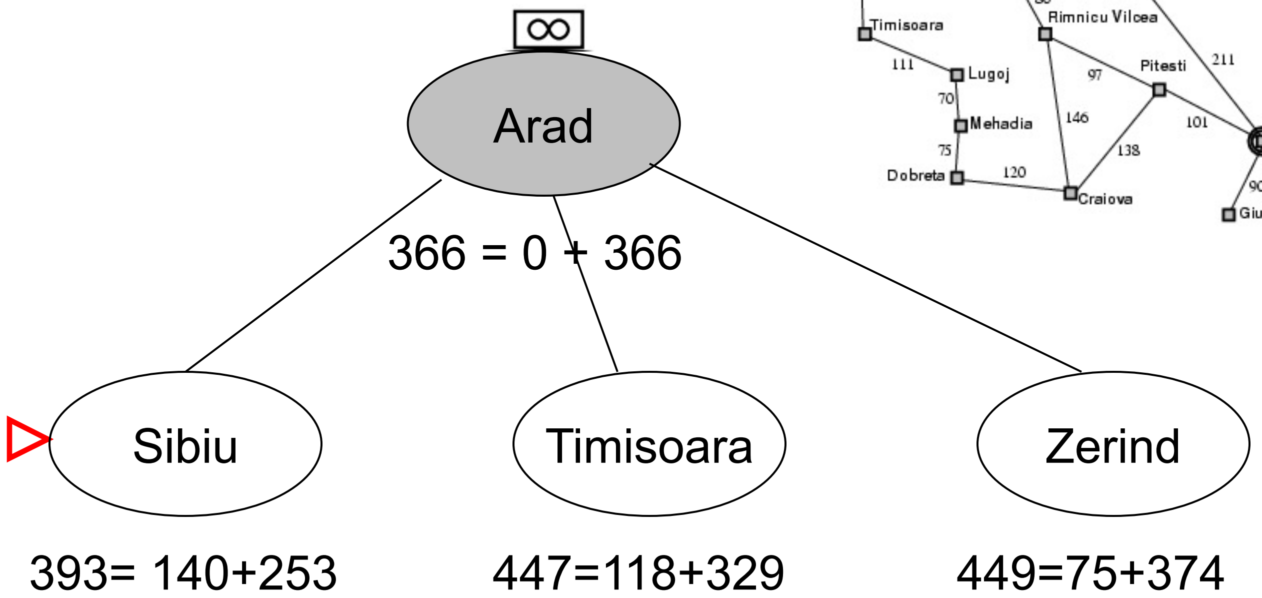


Recursive best first search



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	~~178~~ 176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	~~98~~ 100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



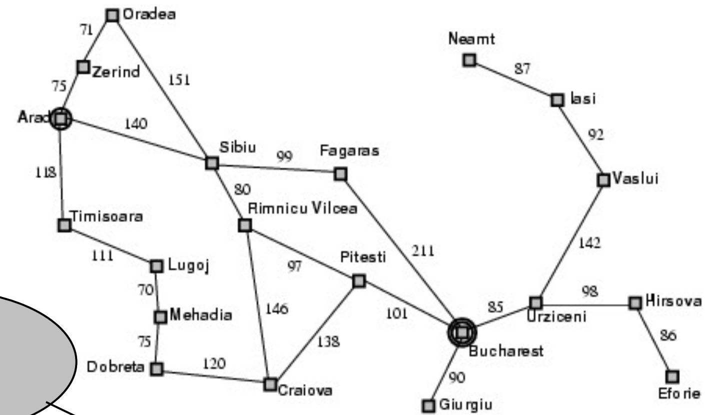
F_limit=+infinity
Sibiu is now best, with f(best)=393
Alternative is Timisoara, with value=447
-> f_limit becomes 447

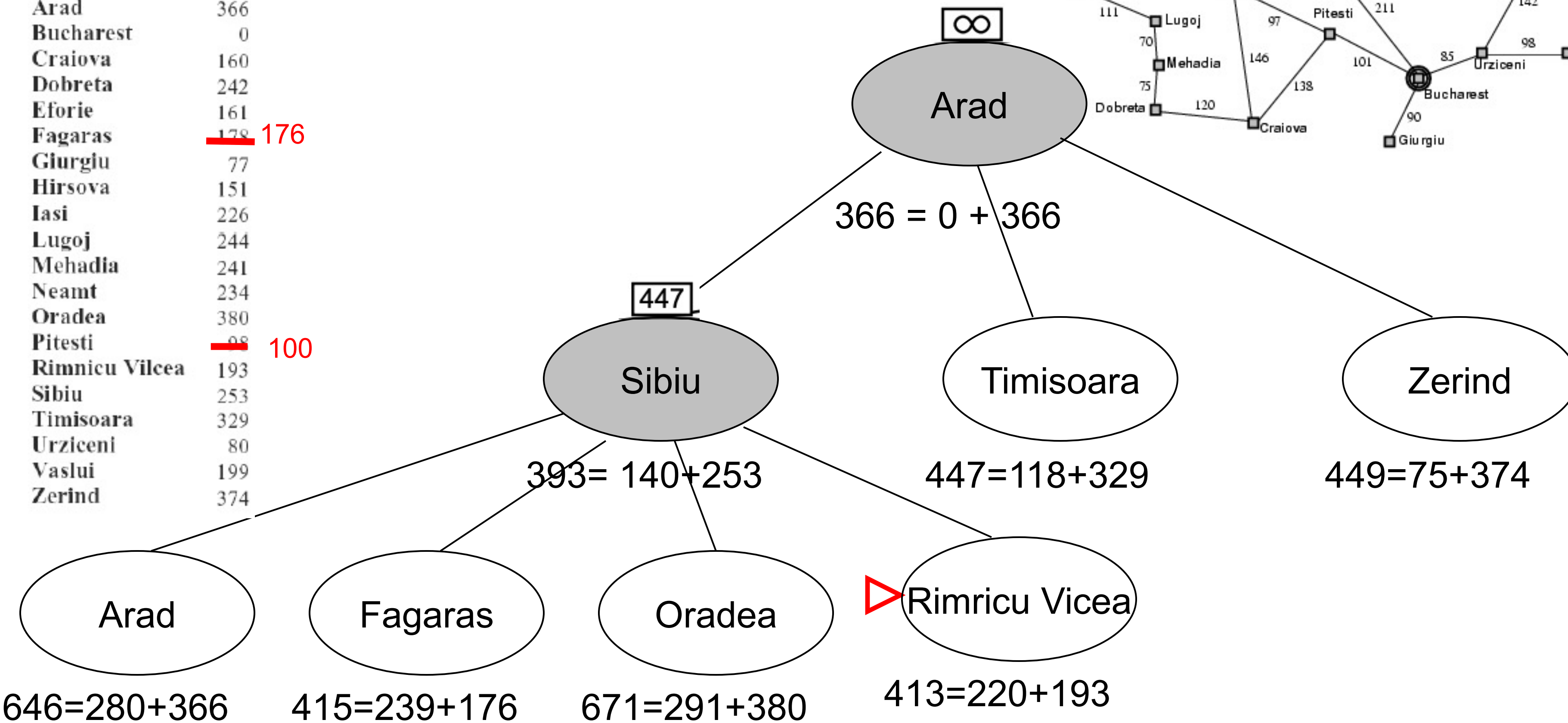
(*result*, *f* [*best*] ← RBFS*(problem, best,* min(*f_limit, alternative*)))

Recursive best first search

Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	~~178~~ 176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	~~98~~ 100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

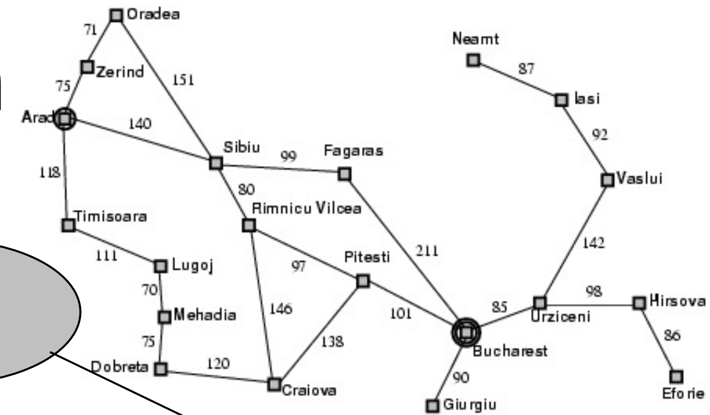




F_limit= 447
Rimnicu-Vicea is now best, with f(best)=413
Alternative is Fagaras, with value=415
-> f_limit becomes 415

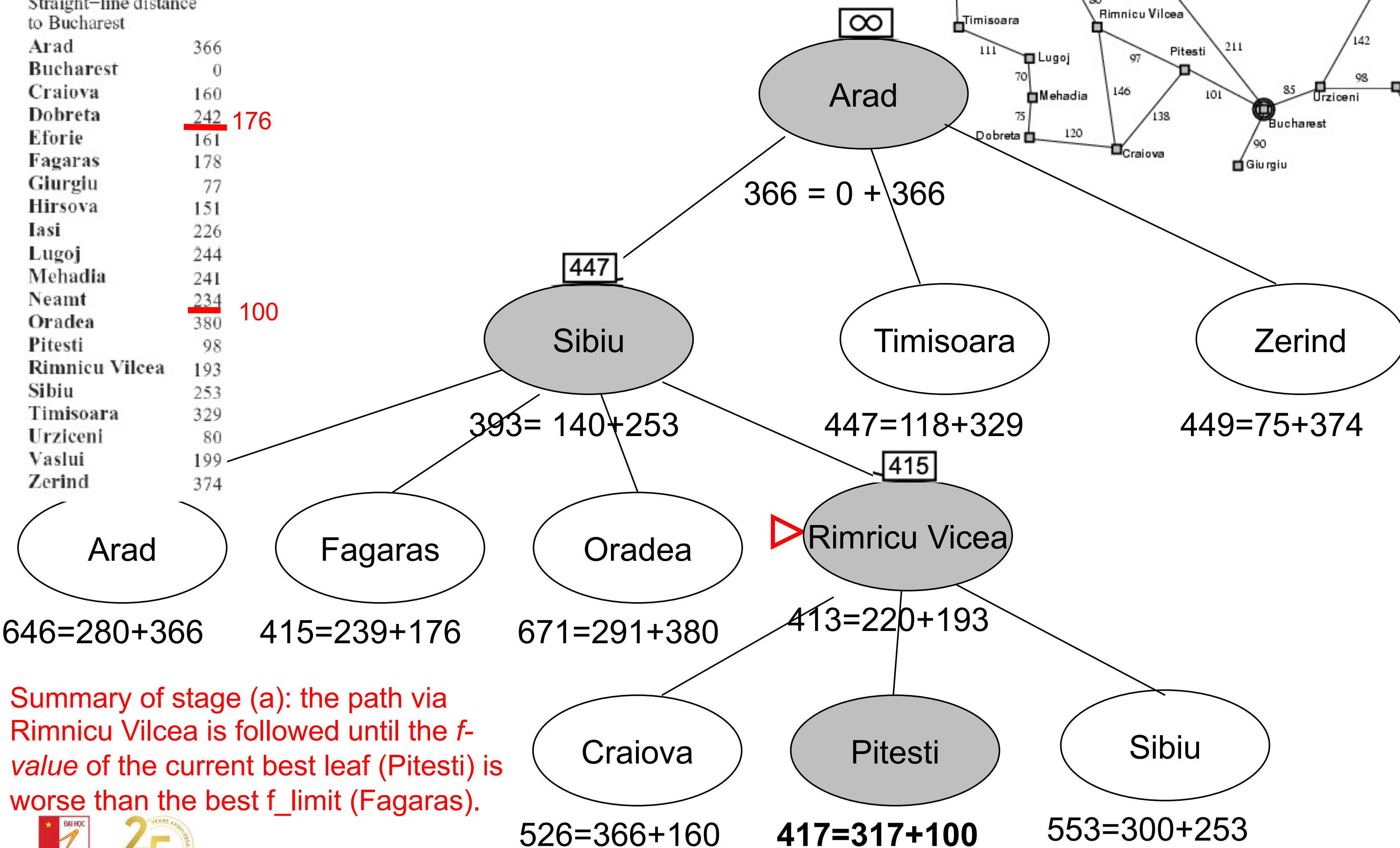
(*result*, *f* [*best*] ← RBFS(*problem, best,* min(*f_limit, alternative*)))

Recursive best first search



Straight−line distance to Bucharest

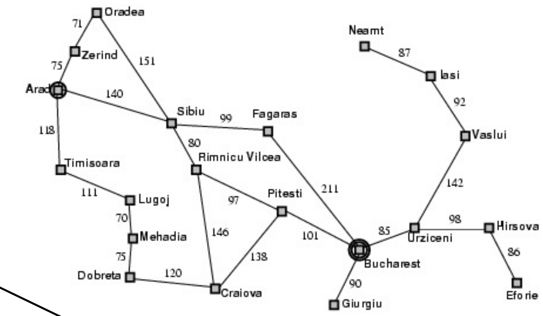
Arad	366
Bucharest	0
Craiova	160
Dobreta	~~242~~ 176
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	~~234~~ 100
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

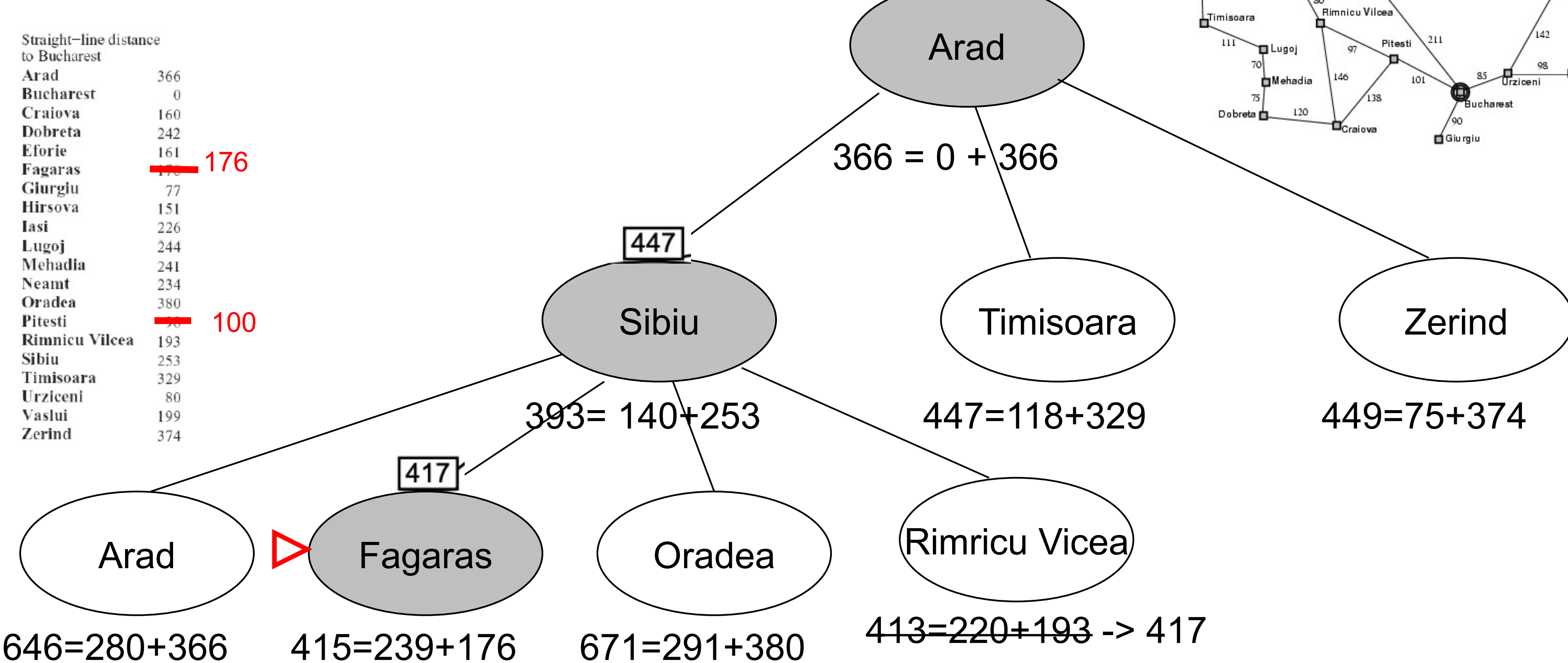


Summary of stage (a): the path via Rimnicu Vilcea is followed until the *f-value* of the current best leaf (Pitesti) is worse than the best f_limit (Fagaras).

Recursive best first search

Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	~~178~~ 176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	~~98~~ 100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



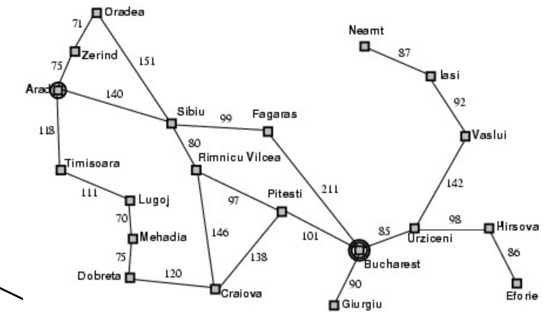


if *f* [*best*] > *f_limit* **then return** failure, *f* [*best*]

Unwind recursion and store best *f*-value for current best leaf Pitesti
best is now Fagaras, with f_limit 415, and alternative is Rimricu Vicea, with f_limit 417

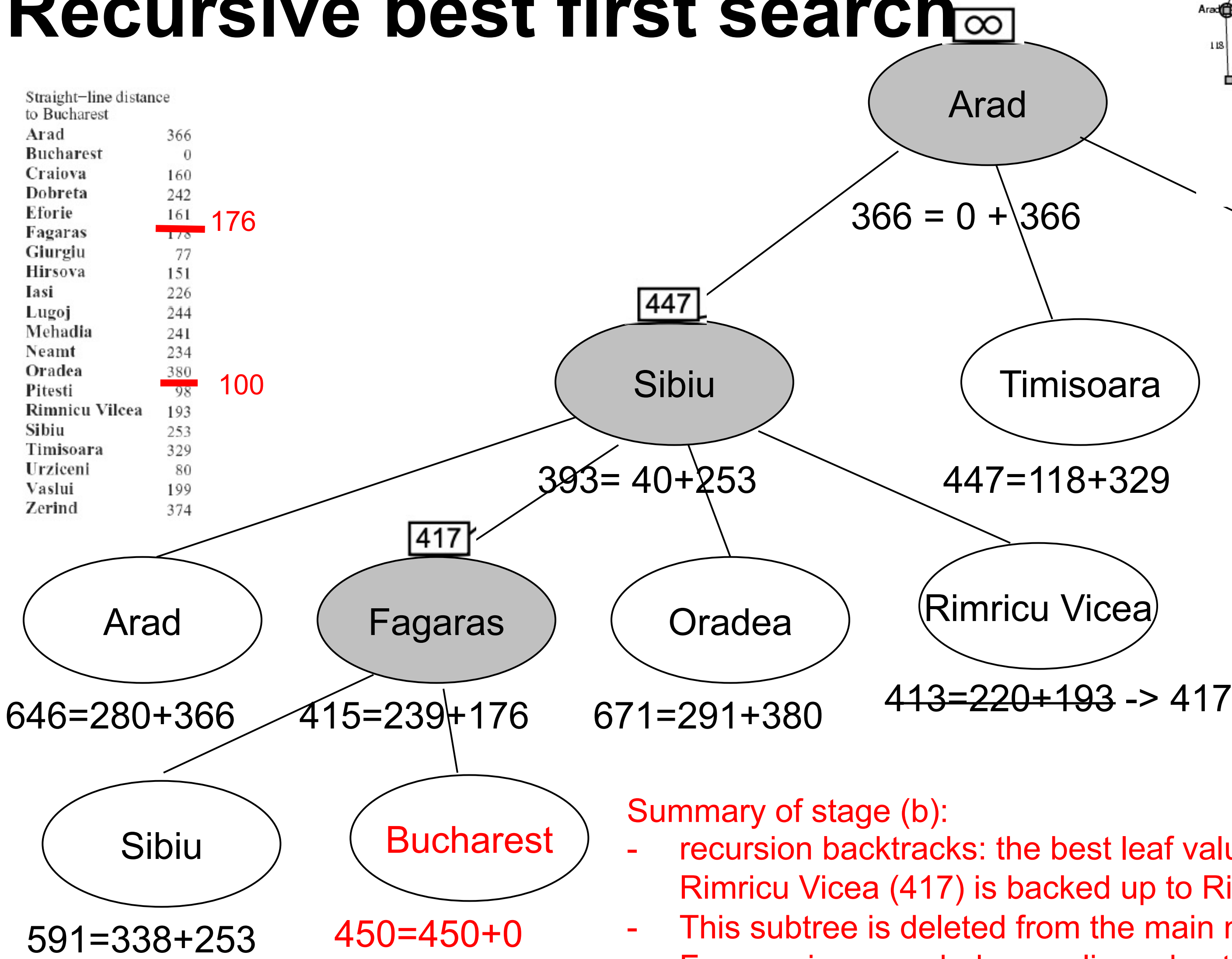
result, *f* [*best*] ← RBFS(*problem, best,* min(*f_limit, alternative*))

Recursive best first search



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	~~178~~ 176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	~~98~~ 100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



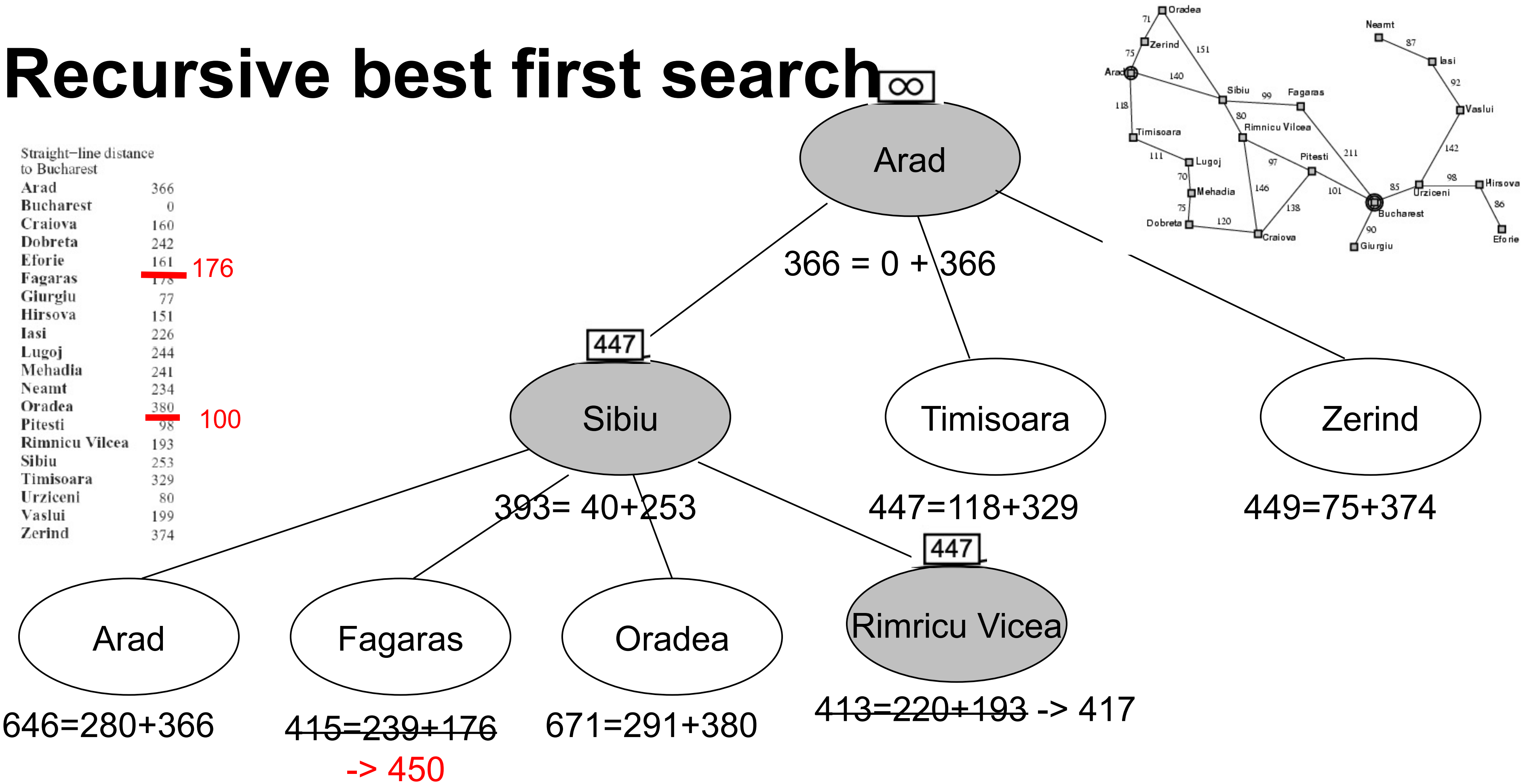
Summary of stage (b):

- recursion backtracks: the best leaf value of the subtree starting at Rimricu Vicea (417) is backed up to Rimnicu Vicea
- This subtree is deleted from the main memory
- Fagaras is expanded, revealing a best value of 450, for a goal node
- But the f_limit (417) is < 450 so the algorithm continues

Recursive best first search

Straight−line distance to Bucharest

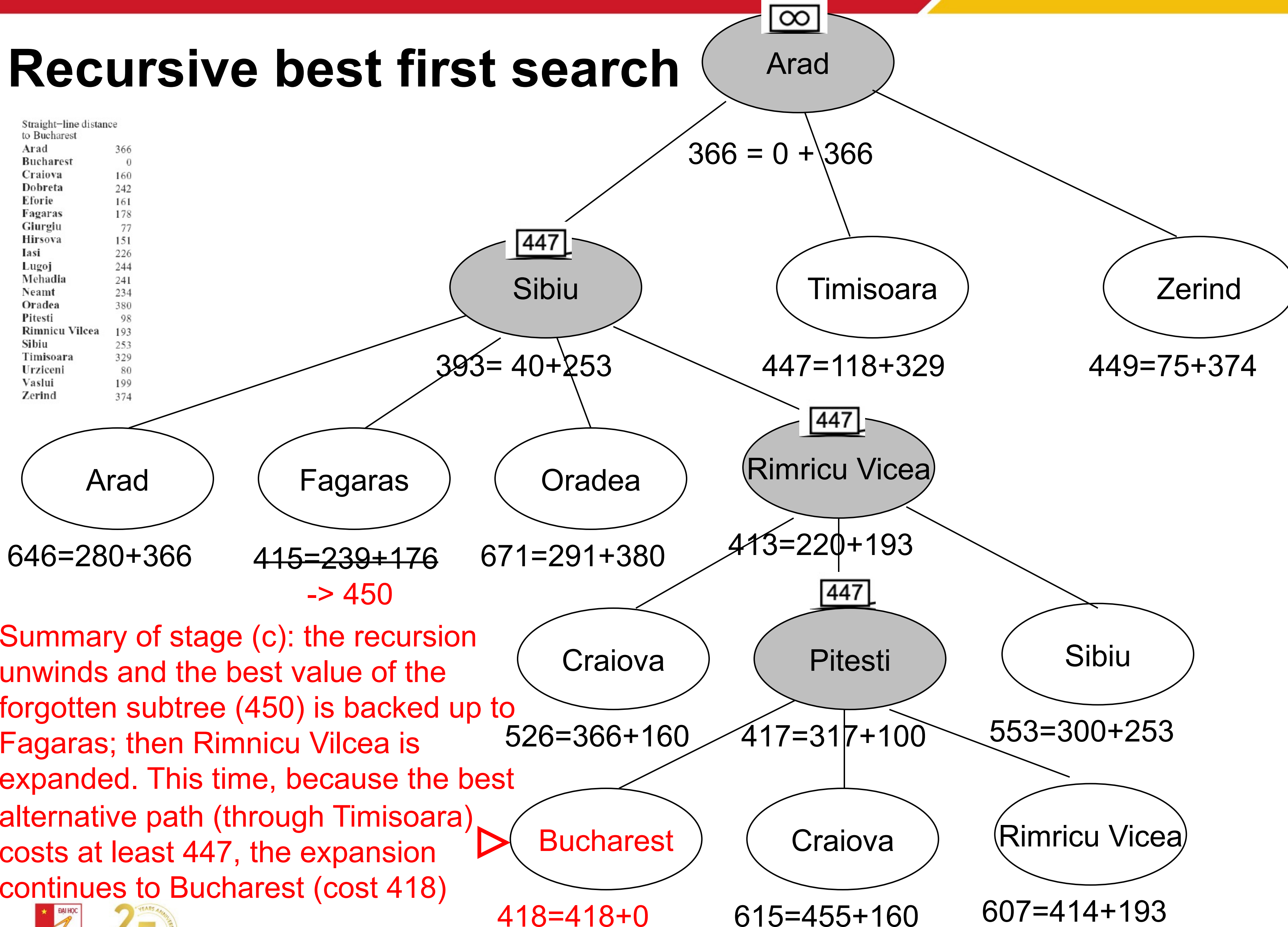
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	~~178~~ 176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	~~98~~ 100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



if *f* [*best*] > *f_limit* **then return** failure, *f* [*best*]

Recursive best first search

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



Summary of stage (c): the recursion unwinds and the best value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest (cost 418)

We found a goal node, less costly than the alternative path -> **this node is returned as solution**

Recursive best first search evaluation

- RBFS is a bit more efficient than IDA*
  - But, still has excessive node re-generation (backtrackings)
- Like A*, optimal if *h(n)* is admissible
- Space complexity is *O(bd)*
- Time complexity difficult to characterize
  - Depends on accuracy of *h(n)* and how often best path changes
  - In practice:
    - If the costs grow monotically, then RBFS expands less nodes (in total, considering also re-expansions) than IDA*
    - Else, RBFS and IDA* are difficult to compare, as they traverse entirely different search trees
      - Need to compare both in practice
  - Time complexity is $O(b^d)$ in the worst case

Recursive best first search evaluation

=> IDA* and RBFS can suffer from ***too little*** memory, leading to re-expanding many nodes (thus, higher time complexity)

- However, they are great in the sense that the space complexity is linear instead of exponential!
  - Good in the presence of big search spaces
- Could be more efficient if they could use a bit more memory, instead of having to re-expand so many nodes

Memory-bounded heuristic search

(Simplified) memory-bounded A*

(Simplified) memory-bounded A* (SMA*)

- Use all available memory.
  - *i.e.* expand best leaves until available memory is full
  - When full, SMA* drops the worst leaf node (with highest *f*-value)
  - Like RBFS, we remember the value for the best descendant in the branch we prune (f_limit in RBFS)

- What if all leaves have the same *f*-value?
  - Problem: the same node could be selected for expansion <u>and</u> deletion
  - SMA* solves this by expanding *newest* best leaf and pruning *oldest* worst leaf
    - In short, 2 strategies for ties: “LIFO for the Fringe, FIFO for deletion”

- The pruned branch is regenerated when all other candidates look worse than its best value

(Simplified) memory-bounded A* (SMA*)

- SMA* is complete if solution is reachable, optimal if optimal solution is reachable
  - *i.e.* if the depth of the shallowest goal node is less than the memory size (expressed in nodes)
- But, time complexity is still $O(b^d)$ in the worst case
- SMA* is a good choice for finding solutions, particularly when:
  - the state space is a graph
  - step costs are **not** uniform
  - node generation is expensive compared to the overhead of keeping the fringe nodes, and the explored nodes, in memory
    - Tradeoff RAM / CPU…

Local search algorithms

Introduction

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
  - Example: the 8-puzzle

start

1	2	5
	7	4
8	6	3

goal

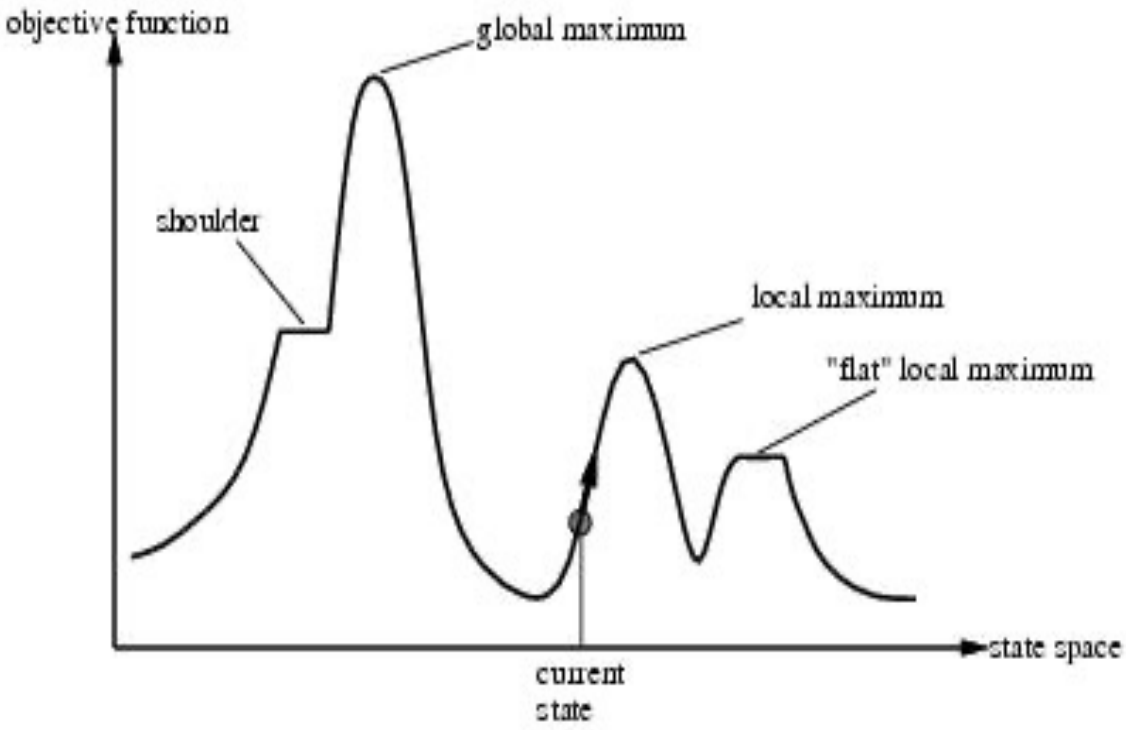
1	2	3
8		4
7	6	5

- In such cases, we can use local search algorithms
  - Standard mathematical **optimization techniques**, belong to numerical analysis
  - Can be used for AI (but **do not belong to AI** algorithms)

- Local search= use single current state and move to neighboring states

Local search and optimization

- In local search algorithms (general optimization algorithms)
  - Sometimes, it can happen that the objective function has to be maximized
  - Example: hill-climbing algorithm
  - So, different from the previous search algorithms, negative cost values (rewards) can be considered
- But, **do NOT** consider negative costs with the search algorithms previously introduced (uniform cost search, greedy best-first search, A* search, IDA*, RBFS, SMA*)!!!
  - Otherwise, you would need to modify the algorithms (out of the scope of this lecture)



Local search algorithms

Hill-climbing search

Hill-climbing search

- Also called “greedy local search” as it only looks to its best **immediate** neighbor state and <u>not beyond that</u>
  - In case of ties, selects the neighbor randomly

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit

- Types of Hill Climbing Algorithm: https://www.javatpoint.com/hill-climbing-algorithm-in-ai
  - Simple hill Climbing: examines only children nodes
  - Steepest-Ascent hill-climbing: examines also neighbor nodes
  - Stochastic hill Climbing: select randomly some neighbor nodes to consider
  - Many other variants…

Simple hill-climbing search

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

  input: *problem*, a problem

  local variables: *current*, **a node.**

   neighbor, **a node.**

  current $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])

  loop do

    neighbor $\leftarrow$ a highest valued successor of *current*

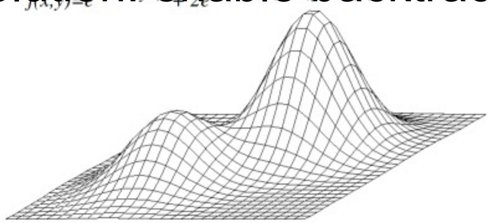
    if VALUE [*neighbor*] ≤ VALUE[*current*] **then return** STATE[*current*]

    current $\leftarrow$ *neighbor*

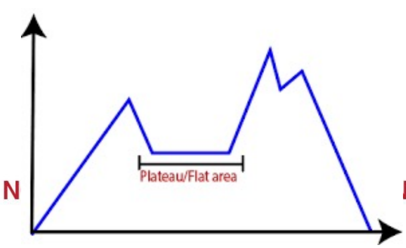
Hill-climbing search evaluation

- Advantages:
  - Use very little memory
  - Find often reasonable solutions in large or infinite state spaces
- But, 3 main problems:

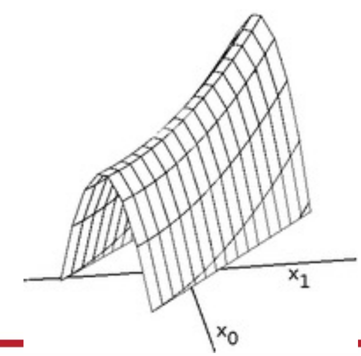
Local maximum:
Hill climbing might be stuck in a local maximum
Possible solution: enable backtracking



Plateau:
Hill climbing might not find the best direction to move to in a plateau (random walk)
Possible solution: enable bigger steps



Ridge:
Flat like a plateau, but with dropoffs to the sides. *E.g.* steps to the North, East, South and West may go down, but a combination of two steps (e.g. N, W) may go up
https://en.wikipedia.org/wiki/Hill_climbing
Possible solution: use a gradient method instead if the function is differentiable



Hill-climbing search

- Some problem spaces are great for hill climbing and others are terrible
- For example, hill climbing can be applied to the travelling salesman problem
  - Initialization: solution visiting all the cities (even if very long)
  - The algorithm makes small improvements to it, such as switching the order in which two cities are visited
  - Eventually, the algorithm will likely return:
    - a much shorter path than the original path
    - a longer path than the optimal solution (local optimum)

Local search algorithms

Simulated annealing search

Simulated Annealing

- **What is annealing? -> in metallurgy**
  - Process of heating, then slowly cooling down a compound or a substance (often, metals)
  - Heating makes the substance "unstable"
  - Then, slow cooling let the substance "flow around" → thermodynamic equilibrium
  - Molecules get optimum conformation
  - Example: annealing silver makes it softer

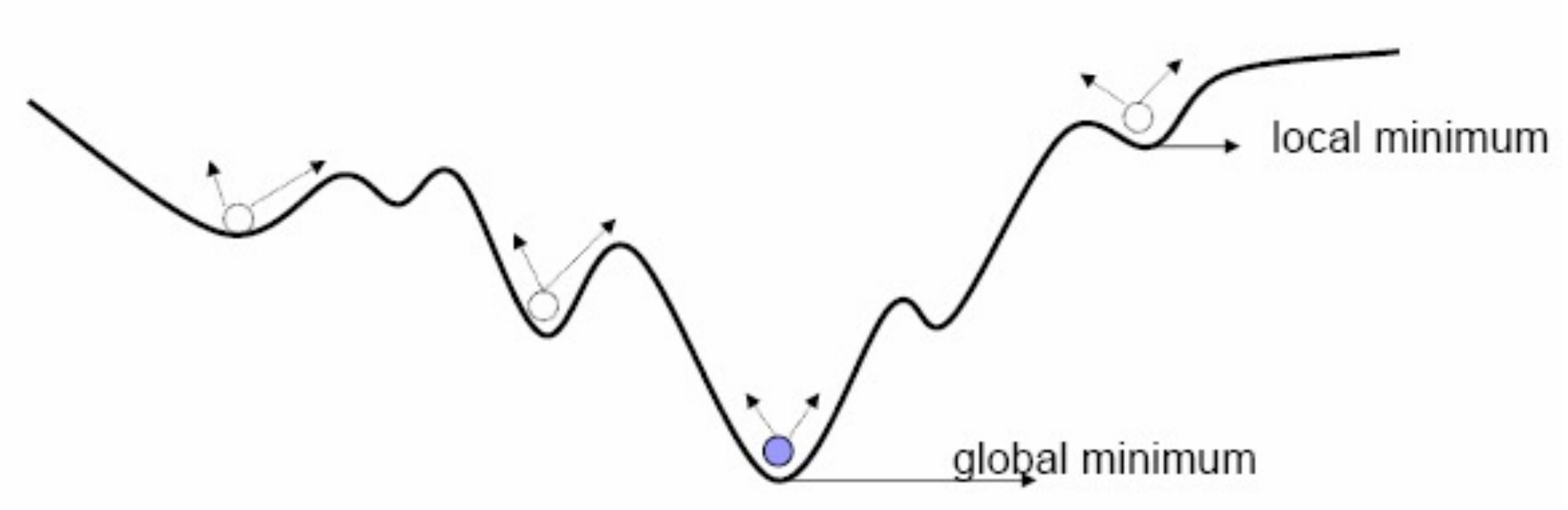
Simulated Annealing

- Simulates the slow cooling of the annealing process

- Applied for combinatorial optimization problem for the first time by S. Kirkpatric (‘83)

- N.B. In this part, we’ll use (as usual except for hill climbing) a cost function to minimize ☺

Simulated annealing



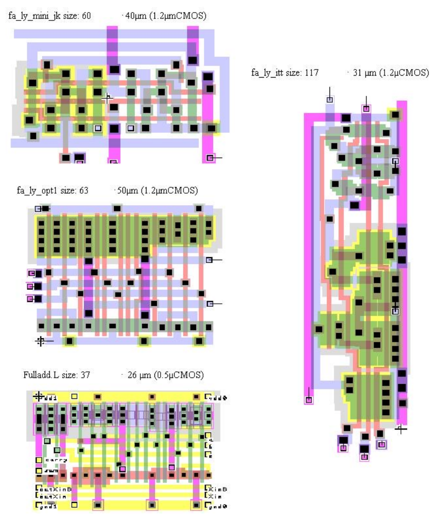
gradually decrease “shaking” to make sure the ball escape from local minima and fall into the global minimum

Simulated annealing

- Idea: Escape local minima by allowing "bad" moves….
  - … but gradually decrease their loss amplitude, and frequency

- Implement:
  - Not always select the best move
  - Accept a "worse" move with probability less than 1 (p<1)
    - The probability of accepting a move decreases inversely to how bad the move is
  - p decreases with time, as the (temperature) parameter T decreases

- If T decreases slowly enough and with a good "schedule", best state is reached
- Applied for travelling salesman problem, VLSI layout, airline scheduling, etc



Simulated annealing

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

input: *problem*, a problem

schedule, a mapping from time to temperature

local variables: *current***,** a node; *next***,** a node.

*T***,** a "temperature" controlling the probability 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*

ΔE ← VALUE[*next*] - VALUE[*current*]

if $\Delta E > 0$ **then** *current* ← *next*

else *current* ← *next* only with probability $e^{\Delta E / T}$

Similar to hill climbing, but a **random** move instead of best move

case of improvement, make the move

Otherwise, choose the move with probability that decreases exponentially with the "badness" of the move.

What's the probability when: T → +∞?

What's the probability when: T → 0?

What's the probability when: Δ=0?

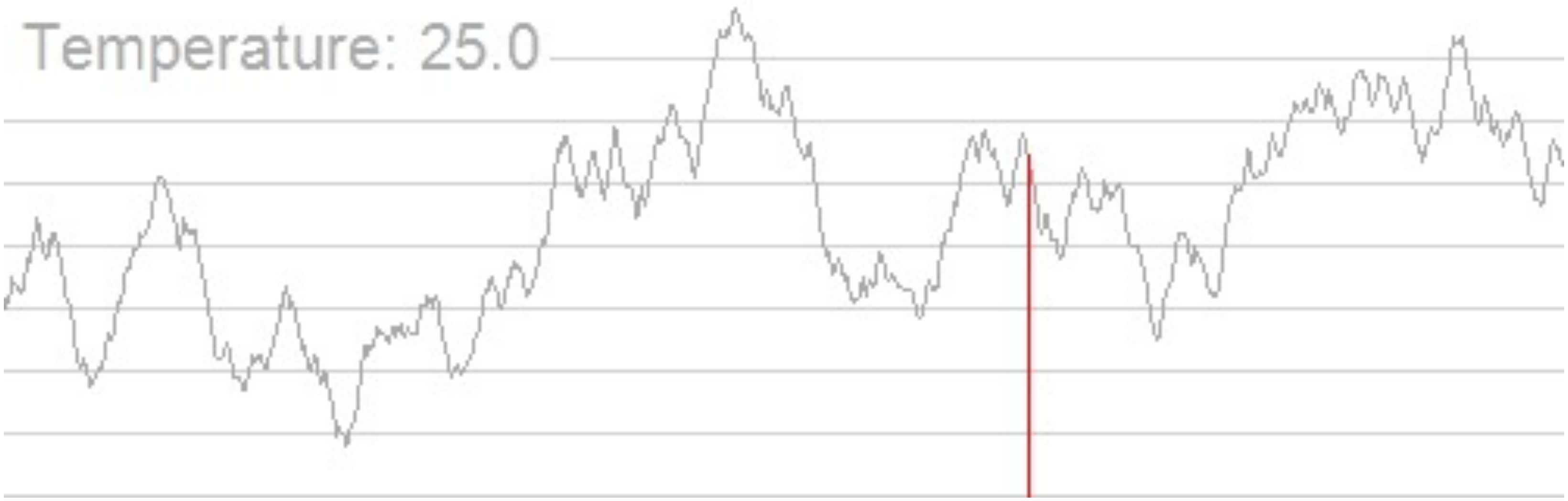
What's the probability when: Δ→-∞?

Simulated Annealing parameters

- Temperature T
  - Used to determine the probability
  - High T : large changes
  - Low T : small changes
- Cooling Schedule
  - Determines rate at which the temperature T is lowered
  - Lowers T "slowly enough", the algorithm will find a global optimum
- In the beginning, aggressive for searching alternatives, become conservative when time goes by

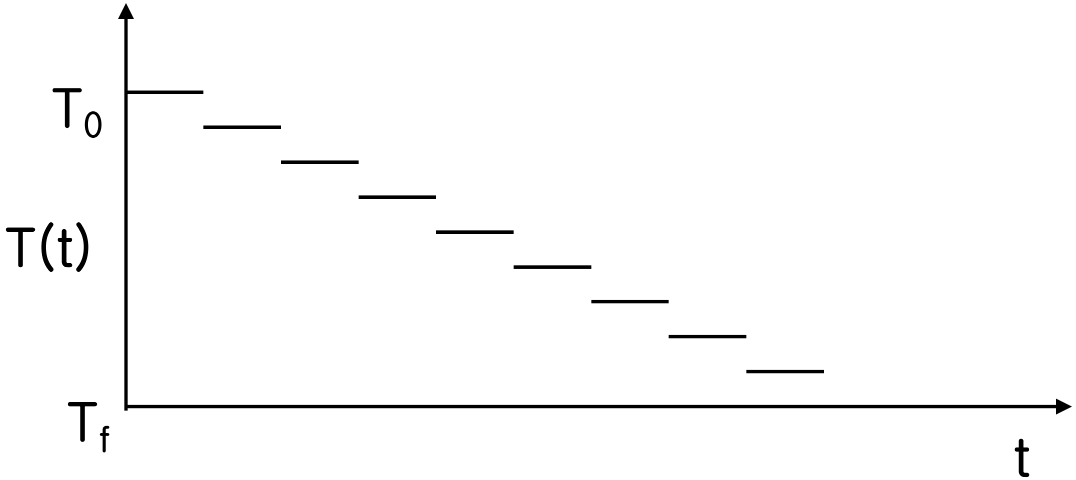
Simulated Annealing parameters

- Visual illustration (from Wikipedia)



Simulated Annealing Cooling Schedule

- The manner in which the temperature T is decreased over the iterations called the cooling **schedule**
  - if Ti is reduced too fast, poor quality
- Many schedules have been proposed, *e.g.*:
  - if Tt >= T(0) / log(1+t) - Geman
    - System will converge to minimun configuration
  - Tt = k/1+t - Szu
  - Tt = a T(t-1) where a is in between 0.8 and 0.99
  - The simple **geometric schedule**, where the temperature is reduced by a constant factor at each chain of moves, and the chain lengths are equal, works quite well on various problems:



Tips for Simulated Annealing

- To avoid getting stuck in local minima
  - Choosing the annealing schedule by trial and error
    - *E.g.*, for the geometric schedule
    - Choice of initial temperature
    - How many iterations are performed at each temperature
    - How much the temperature is decremented at each step as cooling proceeds
- Difficulties
  - Determination of parameters
  - If cooling is too slow → Too much time to get solution
  - If cooling is too rapid → Solution may not be the global optimum

Properties of simulated annealing

- Theoretical guarantee:
  - If T decreases slowly enough (infinite number of plateaux in the schedule, each plateau being “long enough”), it will converge to optimal state!
  - Impossible to reach in practice, but simulated annealing with slow cooling usually produce a good solution

- Advantages of simulated annealing:
  - Easy to implement
  - Can get **un-stuck** from a local optimum
  - Can provide satisfactory results with a relatively low number of iterations => efficient

Summary

Summary

- In this lecture, you have learnt about:

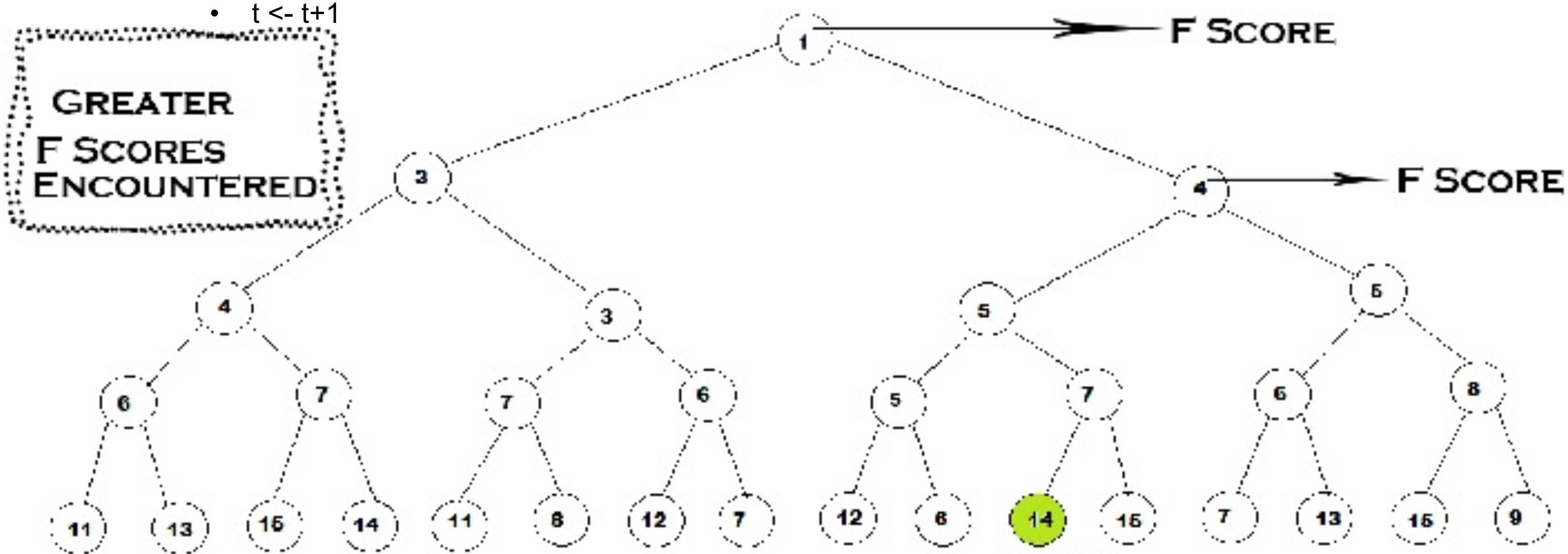
  - Variants of A* search, less memory-consuming
    - Memory-bounded heuristic search: IDA*, RBFS, SMA*

  - Local search algorithms: Hill-climbing search, Simulated annealing search
    - Standard mathematical **optimization techniques**, belong to numerical analysis
    - Can be used for AI (but do not strictly belong to AI)
      - The vocabulary is sometimes a bit different…

Exercise

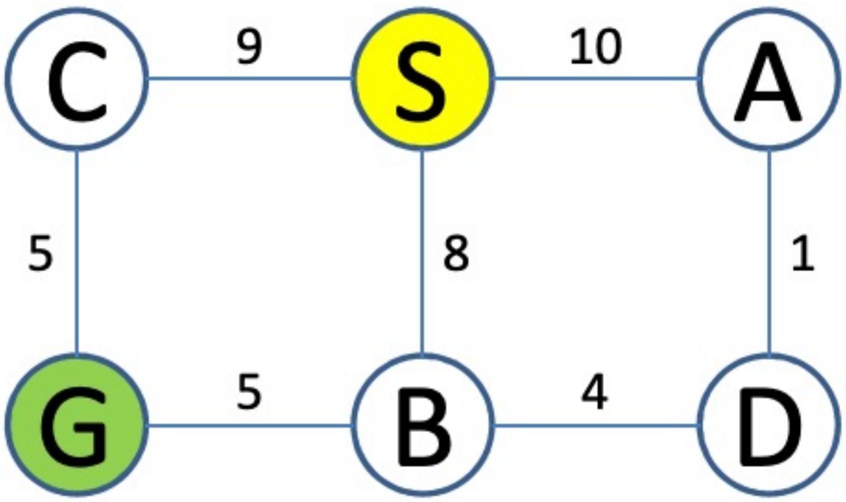
Recall: iterative Deeping A* (IDA*)

- Iterative Deeping version of A* (general idea)
  - If necessary, transform the graph into a tree; remove loopy paths; compute *f* from *g* and *h*
  - Define a threshold, $\theta_1 = f(root\_node)$
  - *t<-1*
  - Then, at each iteration *t*
    - Expand the non-goal nodes *n* in a depth-first way, with stopping criterion $f() > \theta_t$
      - *I.e.* If $f(n) > \theta_t$ then **prune** the node *n* (do not expand it), and put this node in the fringe
      - If any goal node in the fringe, return the goal node with lowest value of *f()*
        - Simple strategy in case of ties: pick randomly; other strategies are possible
      - Set θ_{t+1} = minimum *f()* of all nodes in the fringe
      - t <- t+1



Exercise: IDA* on a graph

- Apply IDA* on this graph search



	S	A	B	C	D	G
heuristic	0	0	4	3	0	0

Chapter 3 – part 3

Questions

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you
for your
attention!