Lecture 4: Informed Search

Artificial Intelligence

CS-6364

Outline

- *Informed (heuristic) search*
  - Greedy Best-first search
  - A* search
  - Memory Bounded heuristic search:
    - Recursive Best First Search (RBFS)
- Heuristic Functions

Informed search strategy

- When selecting what node to expand (in TREE-SEARCH of GRAPH-SEARCH), it is informed by an *evaluation function f(n).* It is a cost (to the solution) estimate, such that the node with the lowest cost is selected!!!
- *The choice of the function f determines the search strategy!!!!*
- *Most evaluation function include some heuristic function h(n)*
- *Some also include the cost of getting to the node from START, g(n)*

Greedy Best-first search

Greedy BEST-FIRST-SEARCH tries to expand a node that is closest to the goal, on the grounds that it is likely to lead to a solution *quickly!!*

* **Evaluates** nodes by using just the heuristic function *f(n)=h(n)*

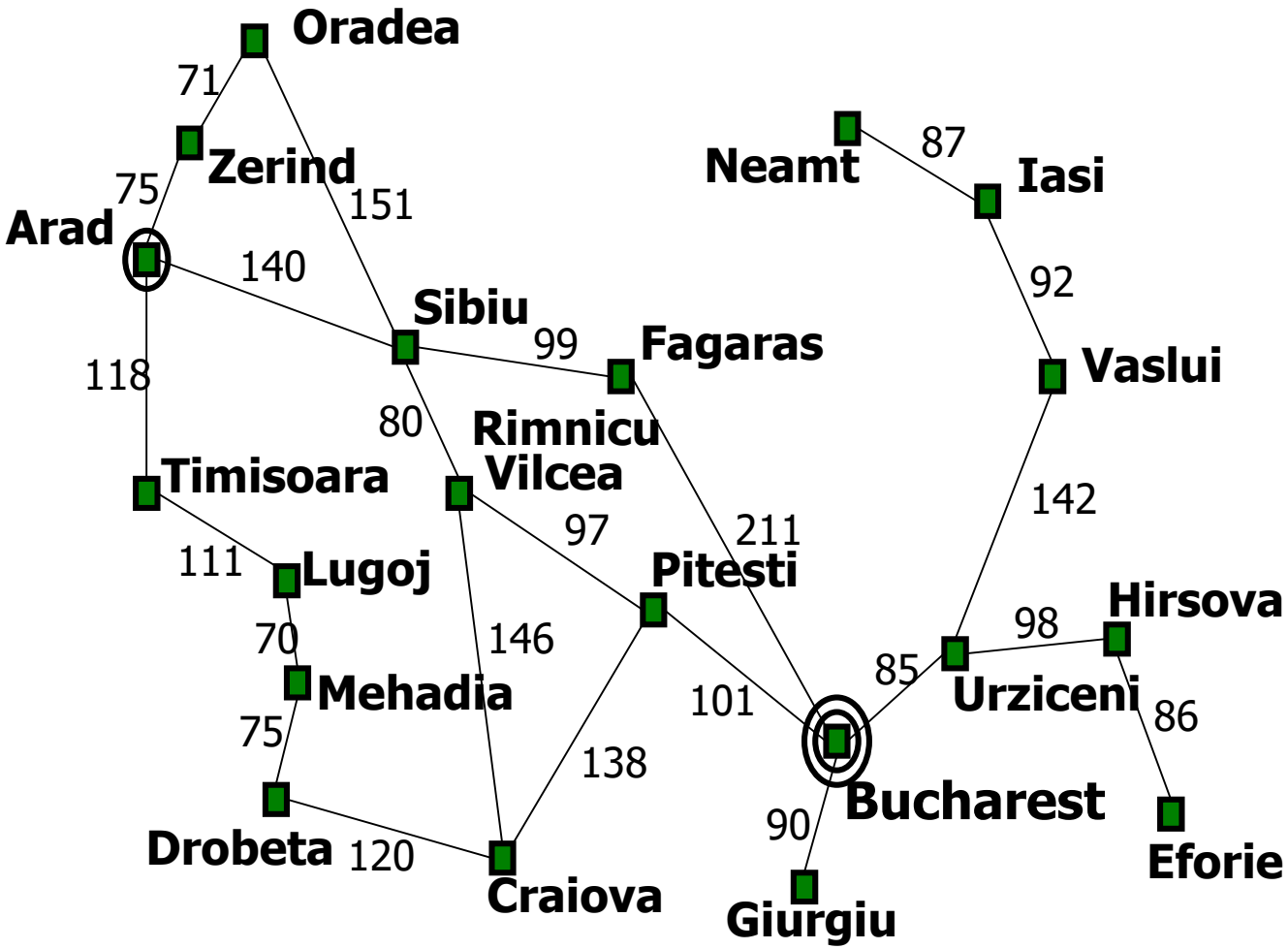
Example of heuristic: *straight-line distance* h^{SLD}

GOAL = Bucharest

$h^{SLD}(In\ (Arad))$ = 366

Important : the values of h^{SLD} cannot be computed from the problem formulation directly.

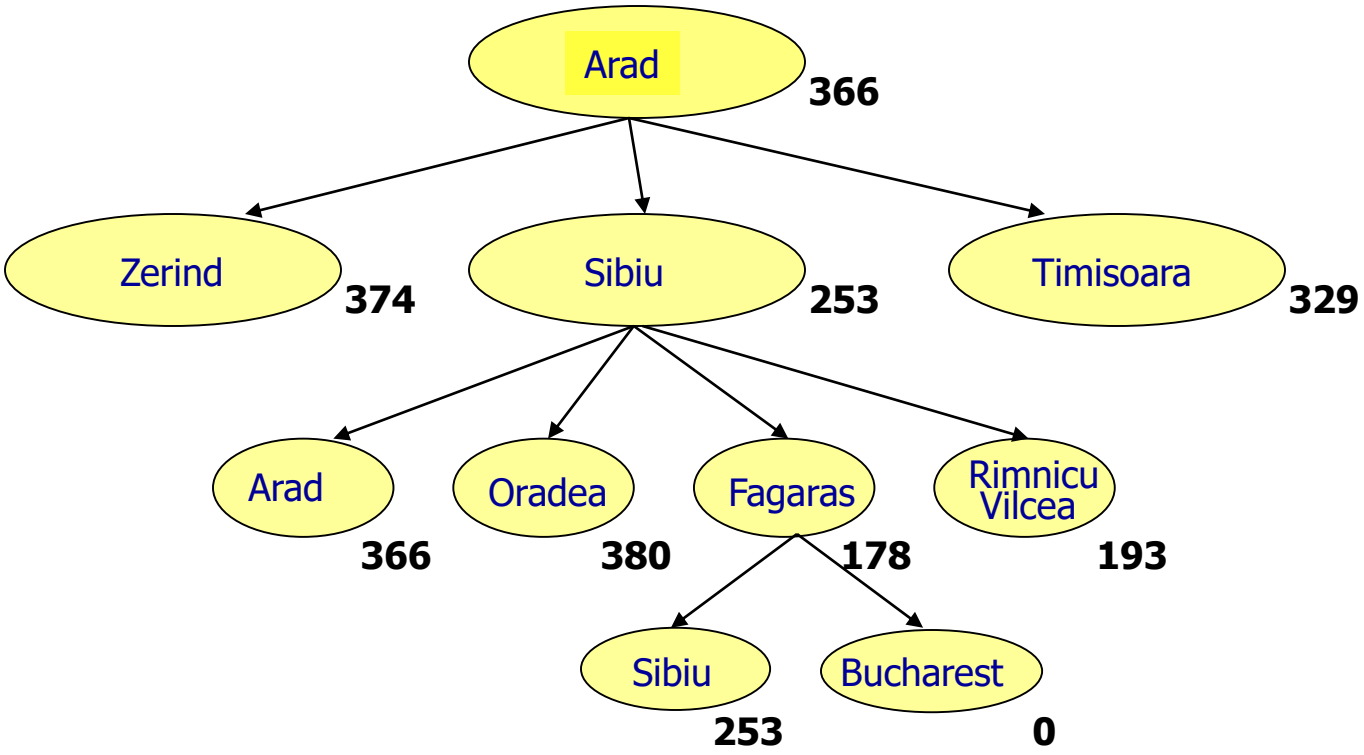
Romania with step costs in km



Straight-line distance to Bucharest

Arad	**366**
Bucharest	**0**
Craiova	**160**
Drobeta	**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**

Greedy search example

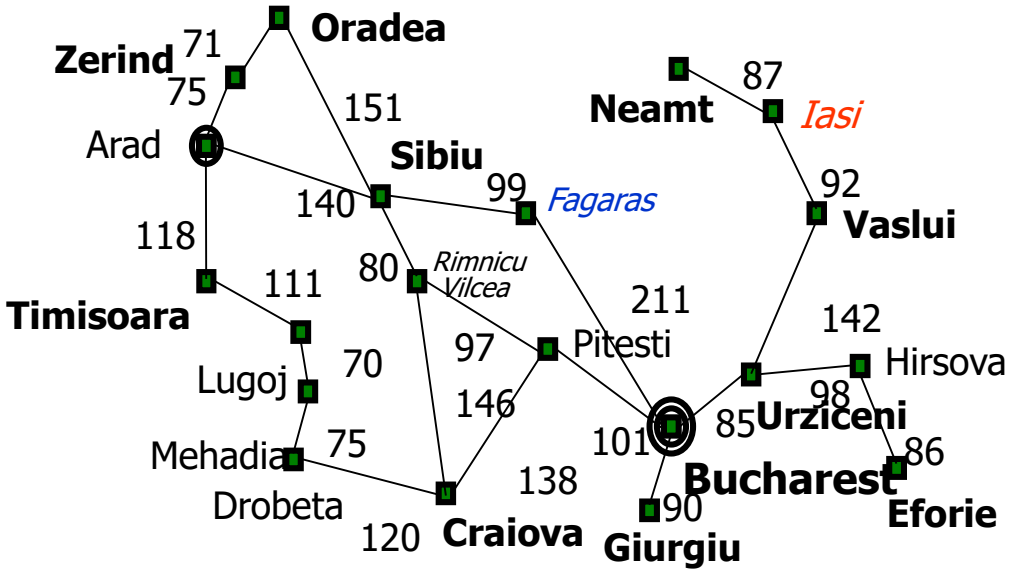


Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Drobeta	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

Is all that good?

- Minimizing *h(n)* is susceptible to false starts
  - *Example: Go from Iasi to Fagaras*



- Greedy best-first search resembles depth-first search because it prefers to follow a single path all the way to the goal, but it will back up when it hits a dead-end.
  - Worse case time and space complexity: $O(b^m)$, where *m* is the maximum depth of the search space

Greedy best-first search – to remember!!!

- Evaluation function $h(n)$ (heuristic)
  = estimate of cost from n to *goal*
- E.g. $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that *appears* to be closest to goal

Properties of greedy search

b - maximum branching factor of the search tree
d - depth of the least-cost solution
m - max depth of the state space (may be ∞)
C^* - the cost of the optimal solution

- Complete ?? No - can get stuck in loops
  - e.g. Iasi→Neamt →Iasi →Neamt

  Complete in finite space with repeated-state checking
- Time ?? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space ?? $O(b^m)$ (keeps all nodes in memory)
- Optimal ?? No

A* search

- Idea: *avoid* expanding paths that are already expensive
- Evaluation function $f(n)=g(n)+h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated cost to goal from n
  - $f(n)$ = estimated cost of path through n to goal

➢ A* search uses an *admissible* heuristic

- Admissible heuristics never overestimate the cost of reaching a goal
  - i.e. $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n *to the goal*
- E.g. $h_{SLD}(n)$ never overestimates the actual road distance
- Theorem: A* search is optimal

A* heuristic properties

- A* search uses an *admissible* heuristic
- A* uses a *consistent* heuristic

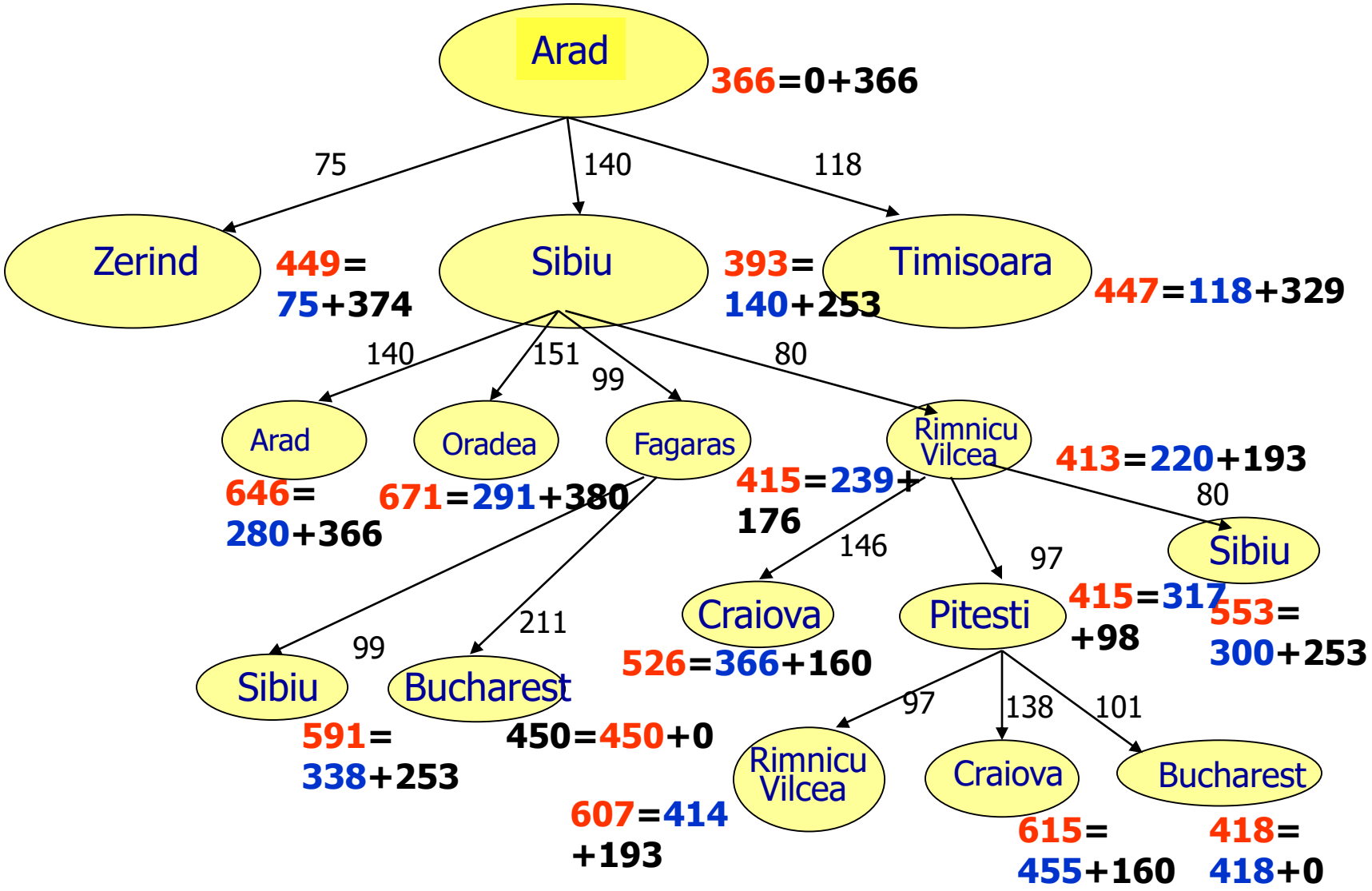
  Consistency = monotonicity

  – A heuristic is consistent if for every node *n* and for every successor *n′* (generated by action *a), the* estimated cost of reaching *the goal* from *n* is no greater than the estimated cost of reaching *the goal* from *n′ :*

    - $h(n) \leq c(n,a,n') + h(n')$

- Theorem: A* search is optimal

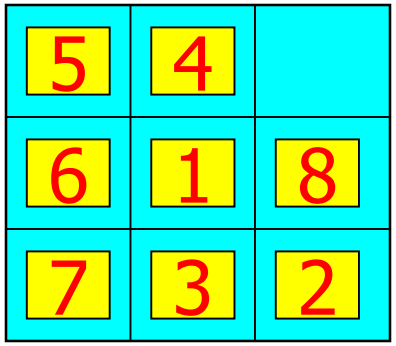
A* search example



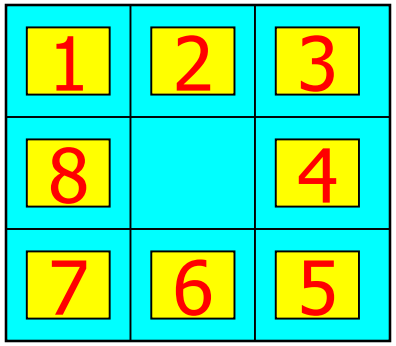
Consistent heuristics

E.g. for the 8-puzzle:

- *h1*(*n*) = number of misplaced tiles
- *h2*(*n*) = total Manhattan distance (i.e. # of squares from desired location of each tile)



Start state



Goal state

h1(*S*) = ?? 7

h2(*S*) = ?? 2+ 3+ 3+ 2+ 4+2+0+2=18

Why is A optimal?*

How do we search?

- If we are Using TREE-SEARCH, then the heuristic h(n) has to be admissible!!!
- If we are Using GRAPH-SEARCH, then the heuristic h(n) has to be consistent!!!

- We will consider only the consistency property.

→ if *h(n)* is consistent, then the values of *f(n)* along the search path are non-decreasing.

PROOF: suppose *n´* is a successor of *n*, → $g(n') = g(n) + c(n,a,n')$

- $f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n)$

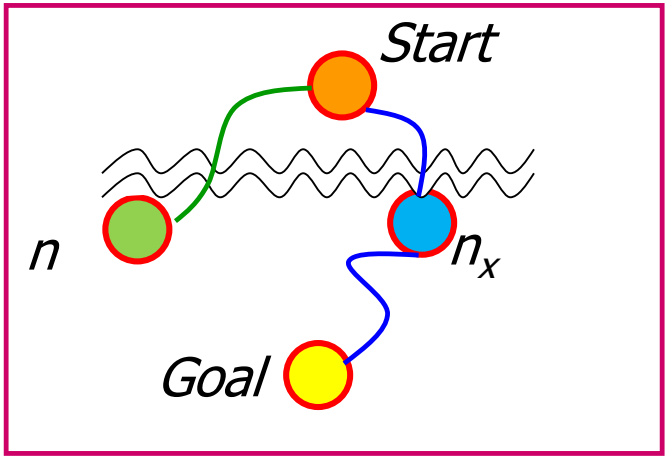
When A selects a node n for expansion, the optimal path to that node has been found.*

PROOF: by contradiction. If this would not be the case, there would have to be another node n_x on the frontier, such that it belongs to the optimal path from the start node to the goal.

Optimality of A* (standard proof)

- In this case, n_x *would have to have a lower value of the evaluation function* $f(n_x) < f(n)$.

$\rightarrow$ The sequence of nodes expanded by GRAPH-SEARCH is in non-decreasing order of *f(n).*



A* always selects for expansion the node with the lowest f(n) from the frontier (Open list)

Consistent heuristics

- Heuristics that ensure that the optimal paths is selected by GRAPH-SEARCH when repeated states are reached are called *consistent heuristics*
- *CONSISTENCY=MONOTONICITY*
- *How do we see if heuristic h is consistent?*
  - Let *n* be a node and *n´* his successor created by action *a.* Then the estimated cost of reaching *the goal* from *n* is no greater than the cost of getting to *n´* plus the estimated cost of reaching *the goal* from *n´ :*
    - $h(n) \leq c(n,a,n') + h(n')$

Triangle inequality

Consequences of consistency

- Every *consistent* heuristic is also *admissible*
- GRAPH-SEARCH is optimal if *h* is consistent
- If *h(n)* is consistent, then the values of *f(n)* along any path are non-decreasing
  - Proof:

Suppose *n′* is a successor of *n*, then

$$f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq$$
$$\geq g(n) + h(n) = f(n)$$

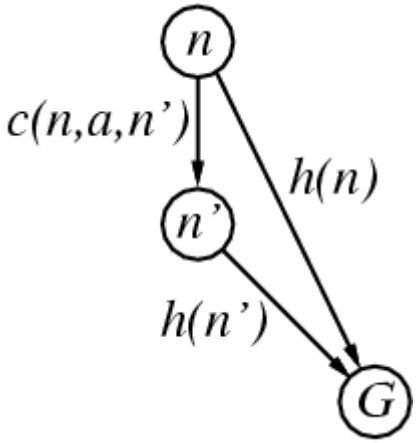
Admissible heuristics

- A heuristic h*(n) is admissible if for every node n,
  h*(n) ≤ 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
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If h*(n) is admissible, A^* using `TREE-SEARCH` is optimal

Consistent heuristics

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

  $h(n) \leq c(n,a,n') + h(n')$



- If *h* is consistent, we have

  f(n') = g(n') + h(n')
  = g(n) + c(n,a,n') + h(n')
  ≥ g(n) + h(n)
  = f(n)

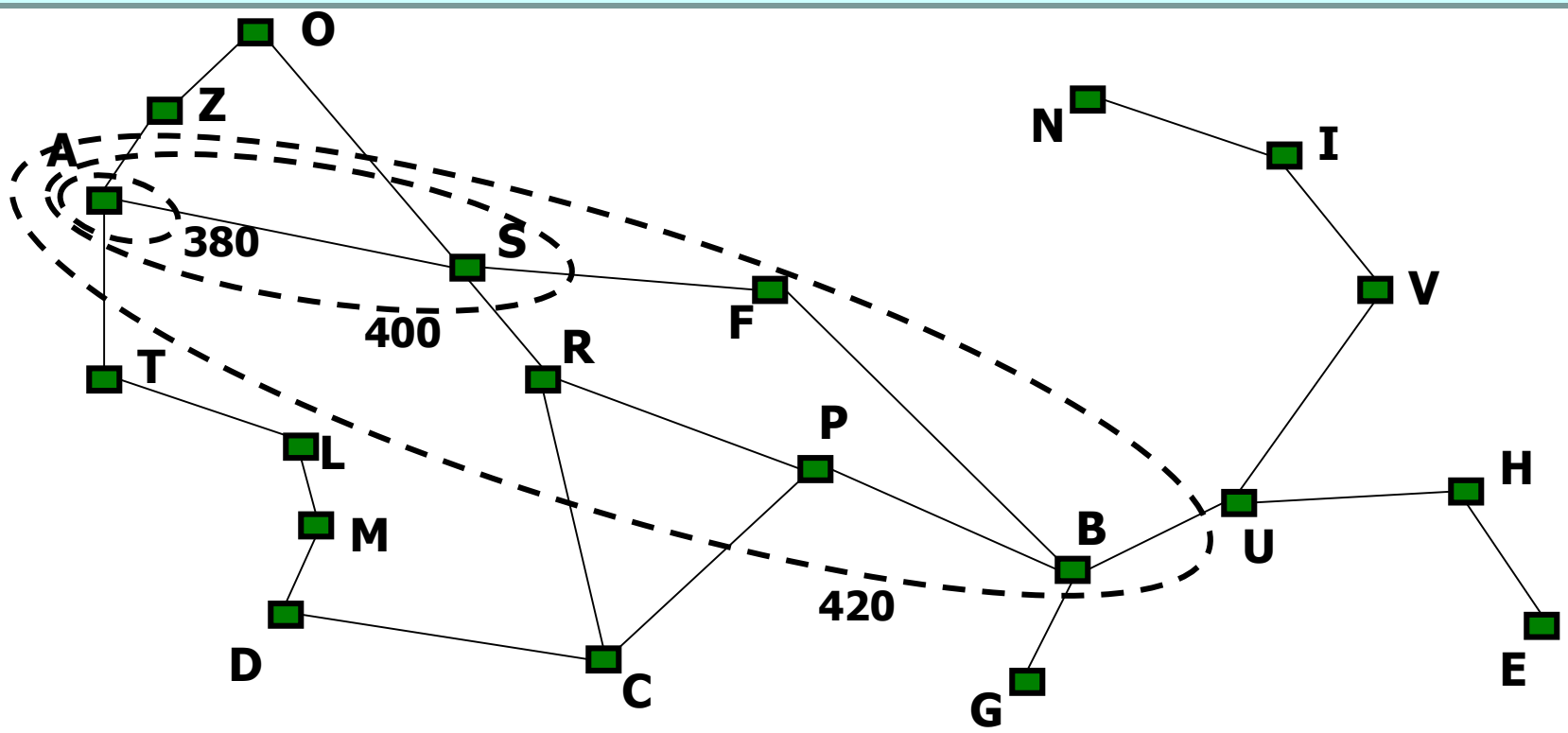
- i.e., *f(n)* is non-decreasing along any path.
- Theorem: If *h(n)* is consistent, A* using `GRAPH-SEARCH` is optimal

Optimality of A* (more useful)

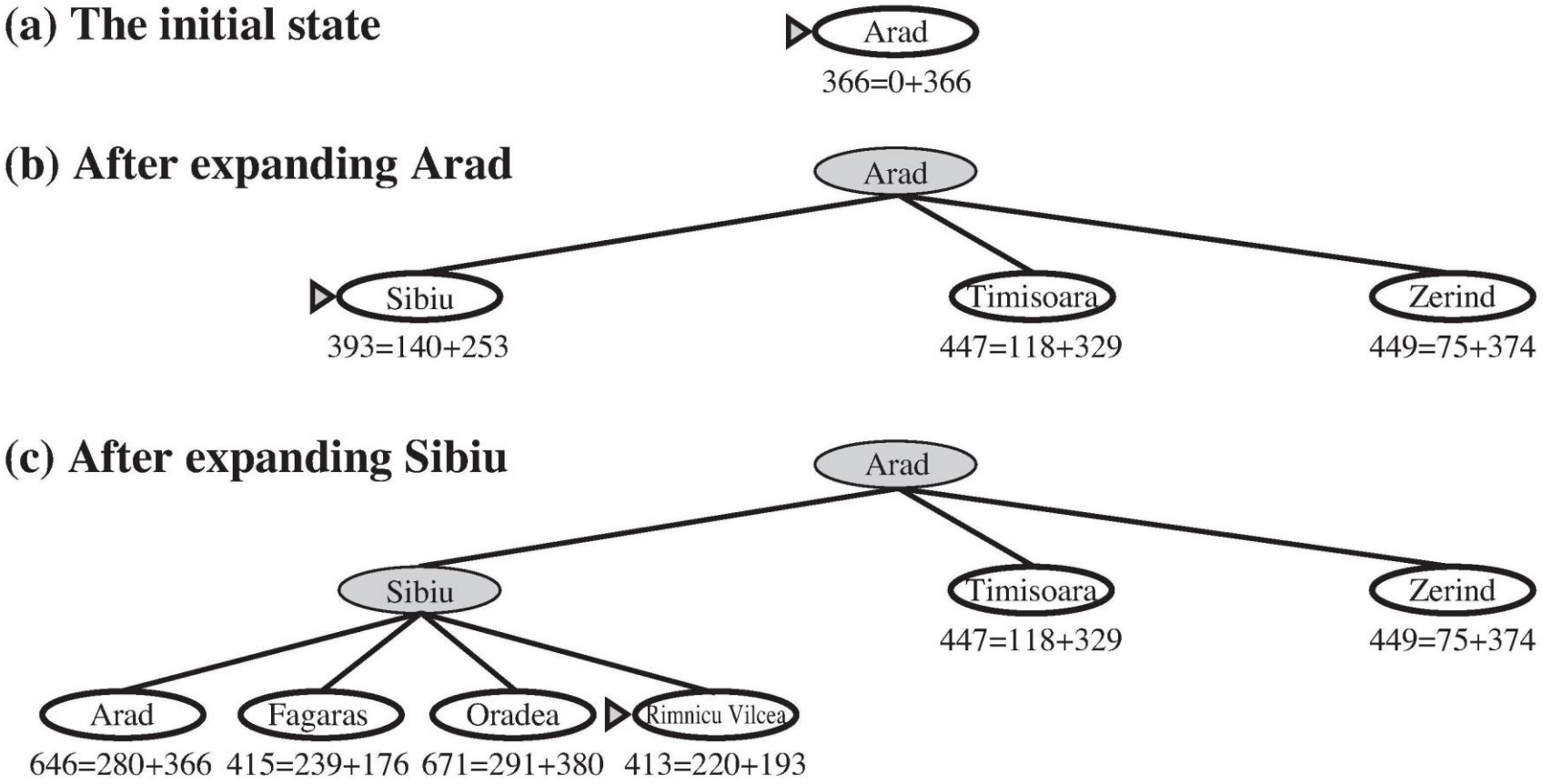
Lemma: A* expands nodes in order of increasing *f* value

Gradually adds "*f*-contours" of nodes (cf. breadth-first adds layers)

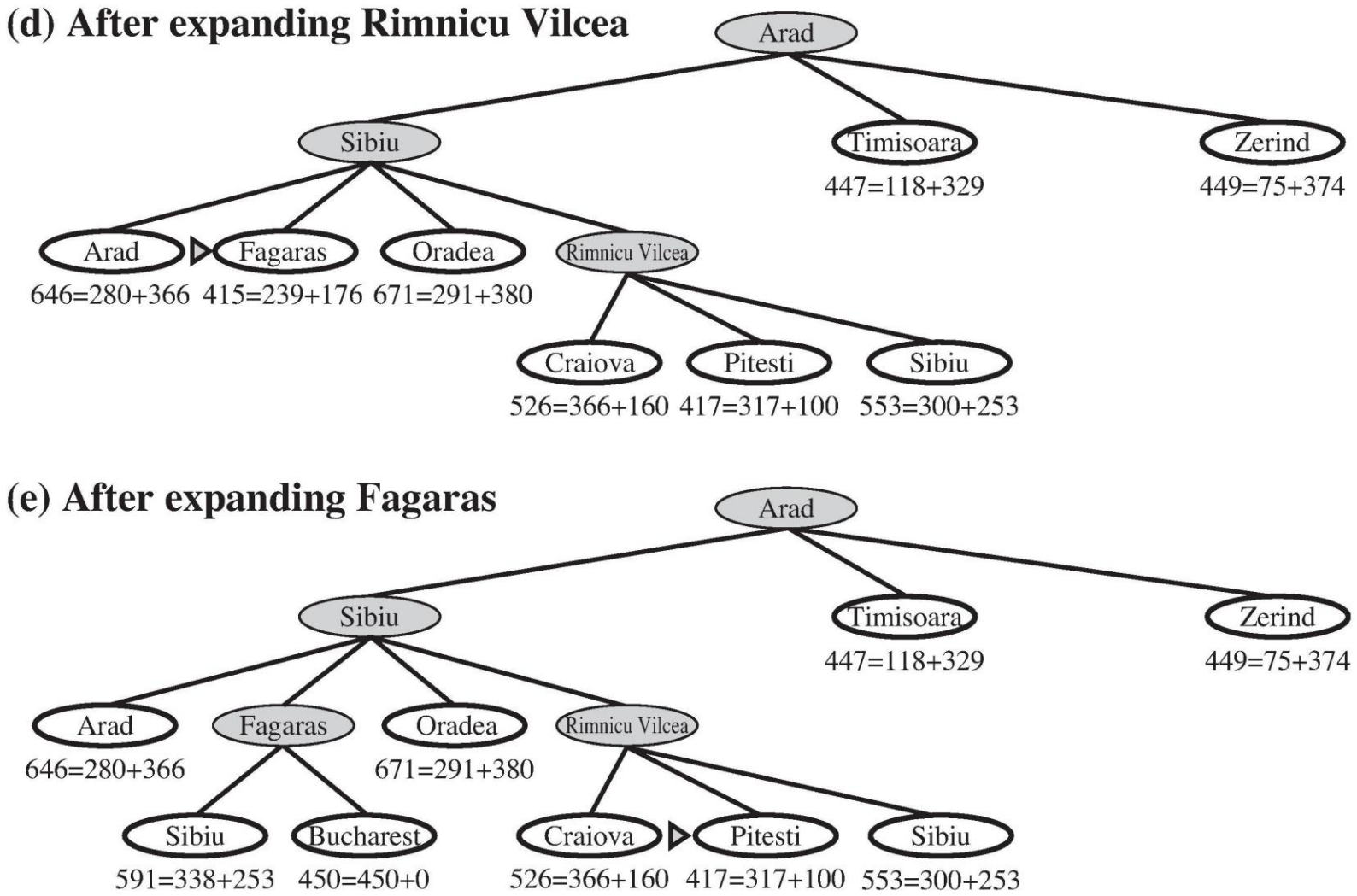
Contour *i* has all nodes with $f = f_i$ where $f_i < f_{i+1}$



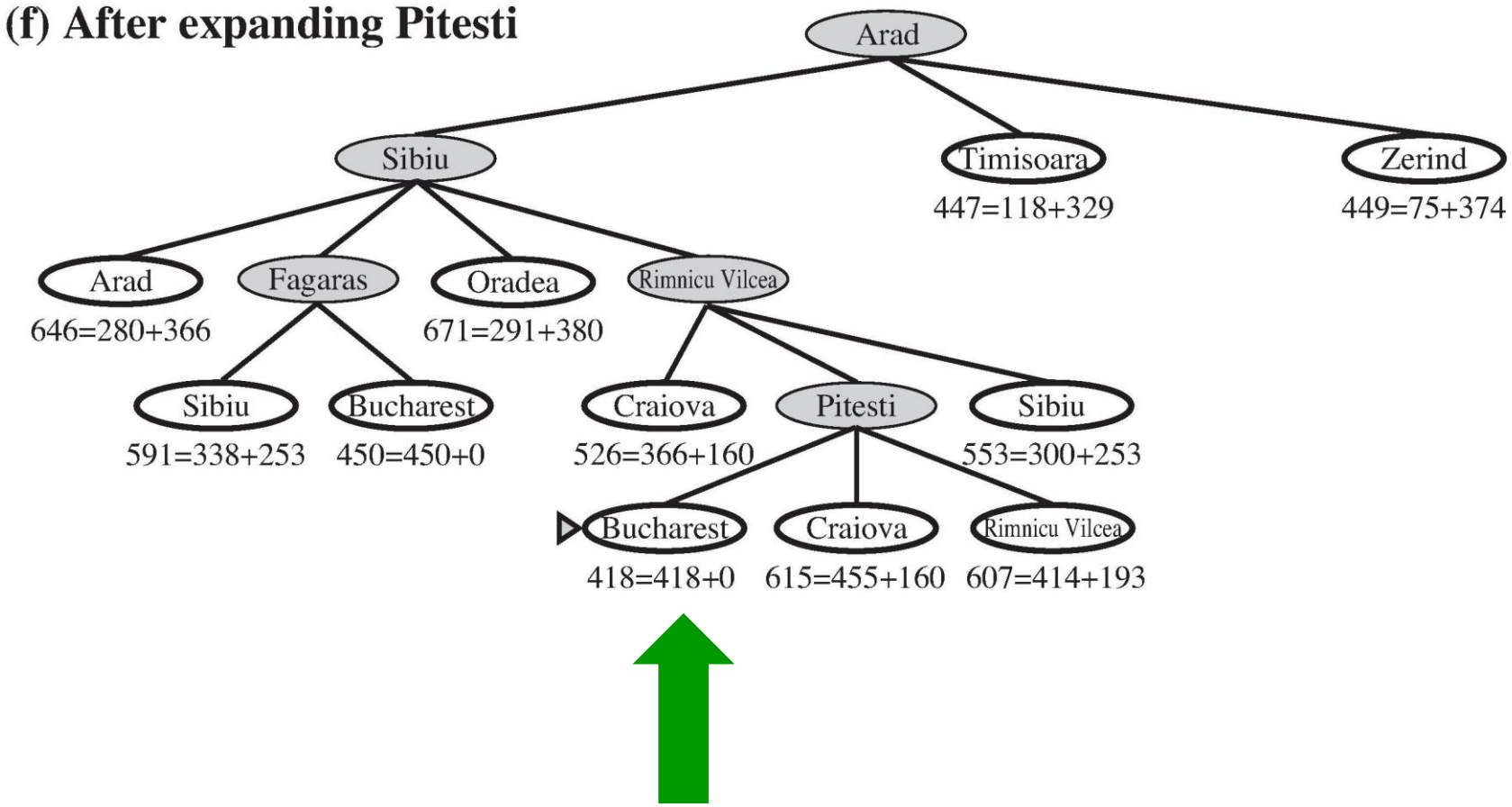
Stages of A* search from Bucharest



Stages of A* search from Bucharest



Stages of A* search from Bucharest



*Properties of A**

- Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time ?? Exponential in [relative error in $h \times$ length of solution path.]
- Space ?? Keeps all nodes in memory
- Optimal ?? Yes – it is optimally efficient for any consistent heuristic

Memory bounded heuristic search

- Reduce the memory requirements for A* ??

- Adapt the idea of iterative deepening to heuristic search: the cut-off is the *f-cost*, not the *depth*
  - *IDA*: At each iteration, the cutoff is the smallest f-cost of each node that exceeded the cutoff of the previous iteration*

Recursive Best-First Search

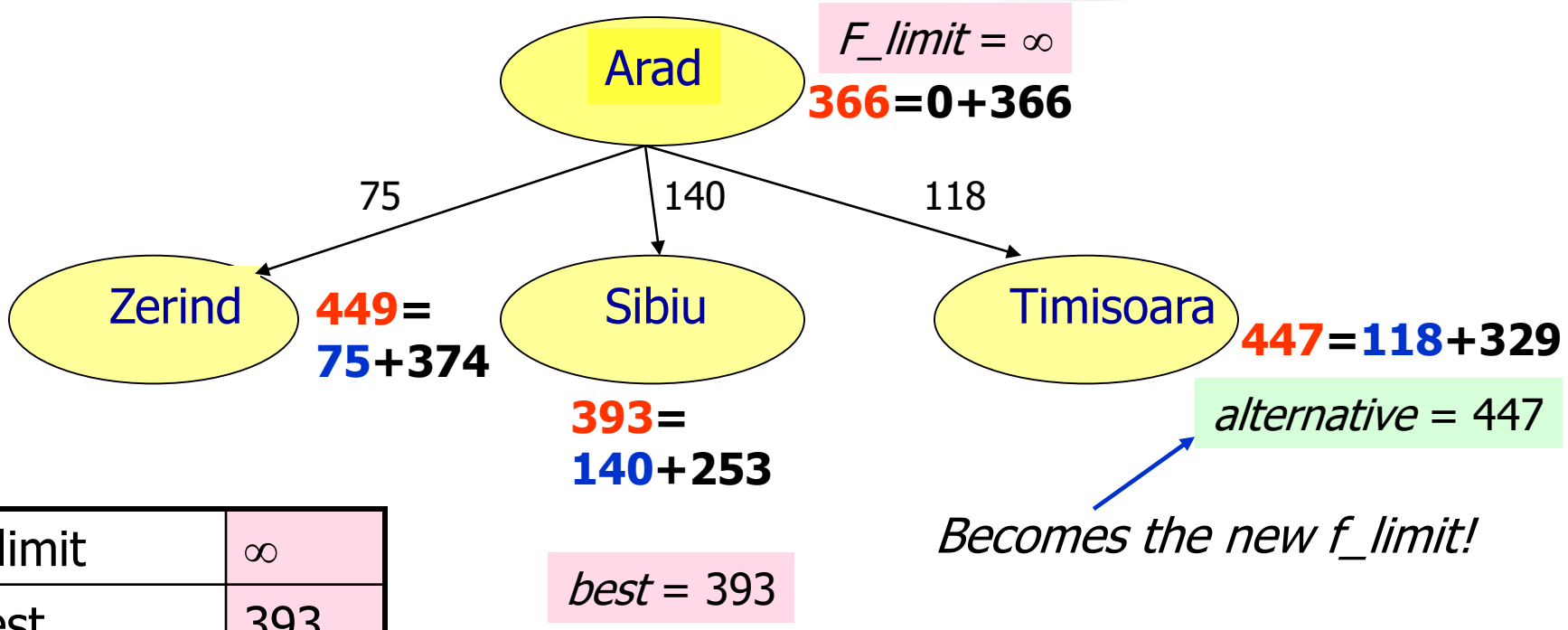
```
function Recursive-Best-First-Search(problem) returns a solution, or failure
  return RBFS( problem, MAKE-NODE (problem.INITIAL-STATE), ∞)

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
```

How does it work???

- We keep track of 5 values:

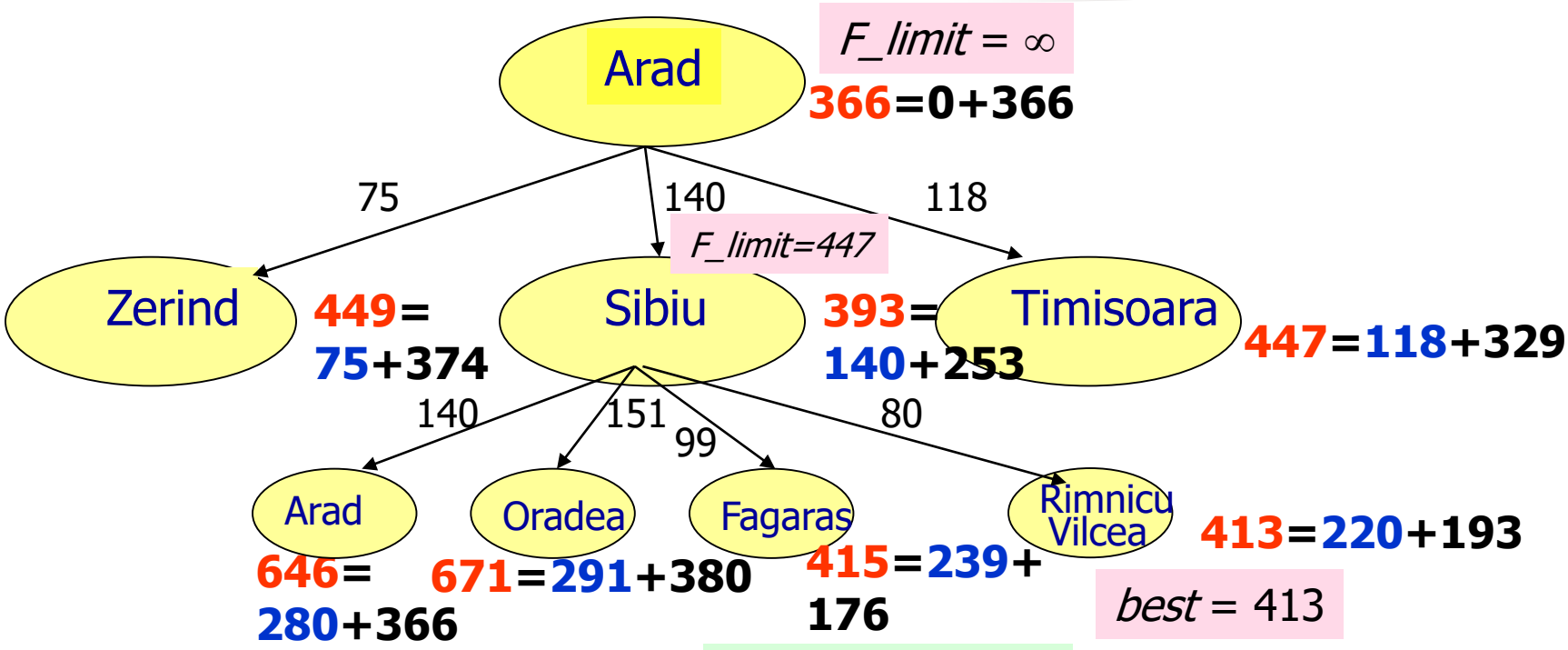
Value/Node	Modified by:
f_limit	f_limit ←min(f_limit,alternative)
best	The minimum *f* value of the successors
alternative	The second minimal *f* value of the successors
Node (current city)	Initial node or the best node or the backtracked node
Best successor (next city)	The node having the f=best

RBFS example -1



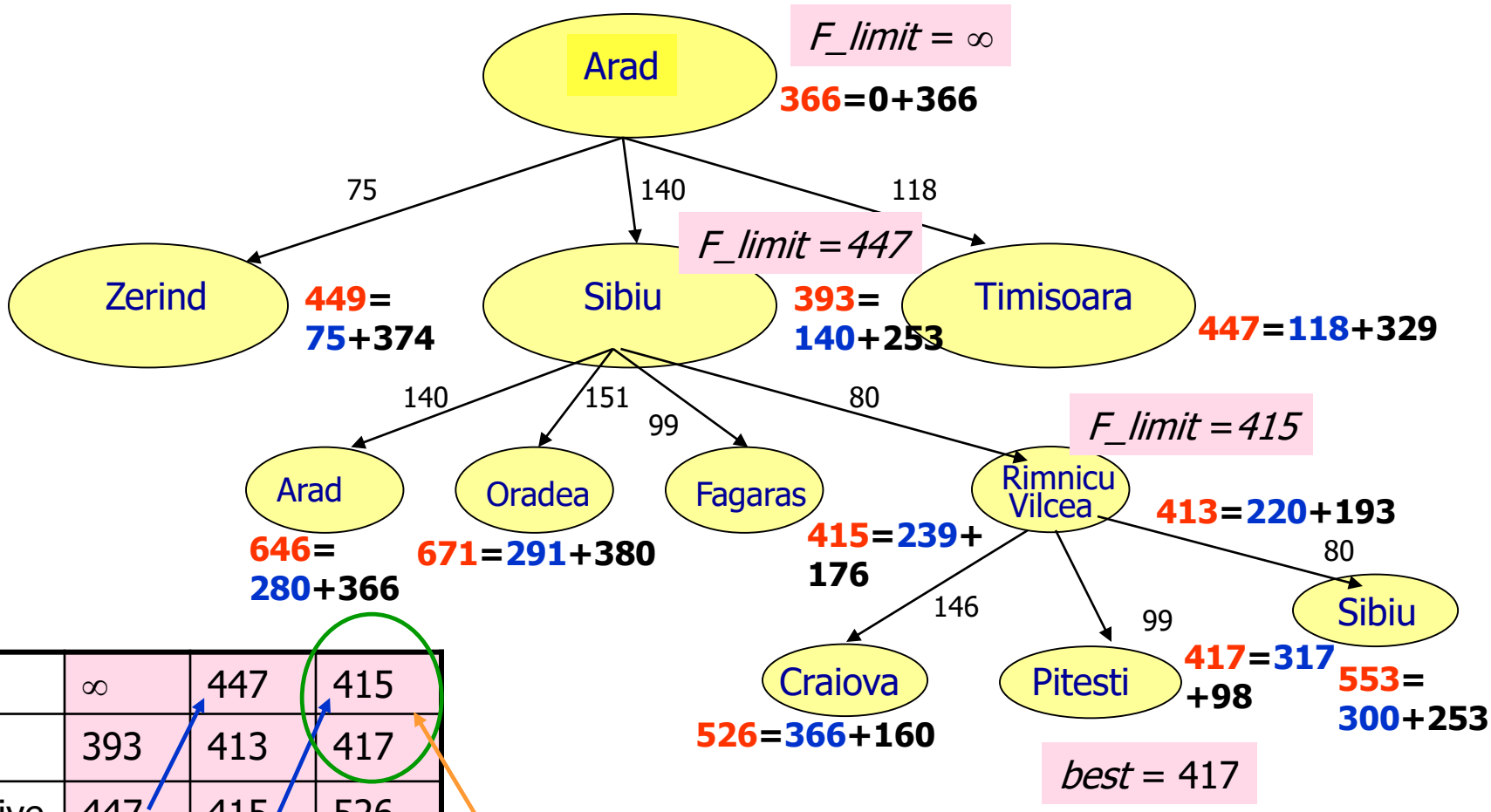
f_limit	∞
best	393
alternative	447
Current city	Arad
Next city	Sibiu

RBFS example -2



f_limit	∞	447
best	393	413
alternative	447	415
Current city	Arad	Sibiu
Next city	Sibiu	RV

RBFS example -3

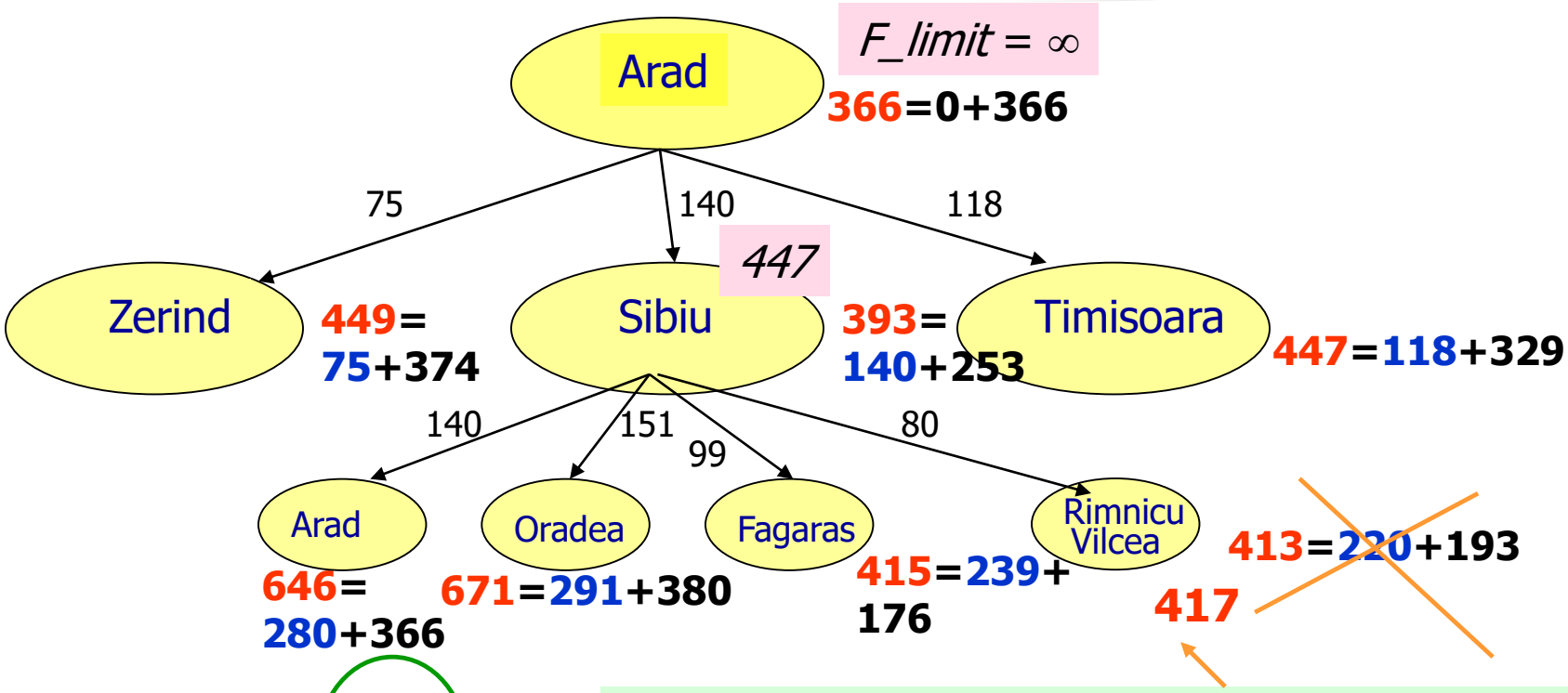


f_limit	∞	447	415
best	393	413	417
alternative	447	415	526
Current city	Arad	Sibiu	RV
Next city	Sibiu	RV	Fail

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

f(RV) =max(f[RV], f(best))=max(413,417)=417

RBFS example -4

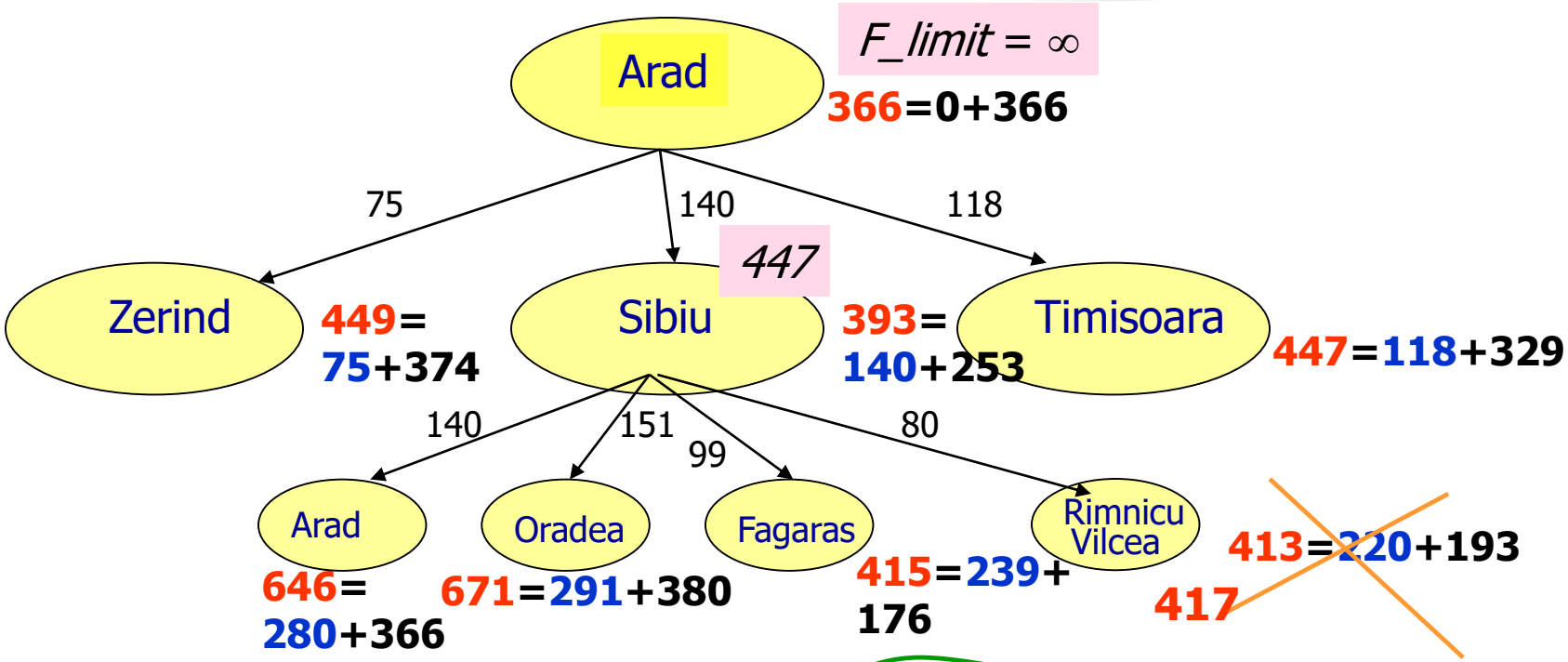


f_limit	∞	447	415
best	393	413	417
alternative	447	415	525
Current city	Arad	Sibiu	RV
Next city	Sibiu	RV	Fail

f(RV) =max(f[RV], f(best))=max(413, 417)=417

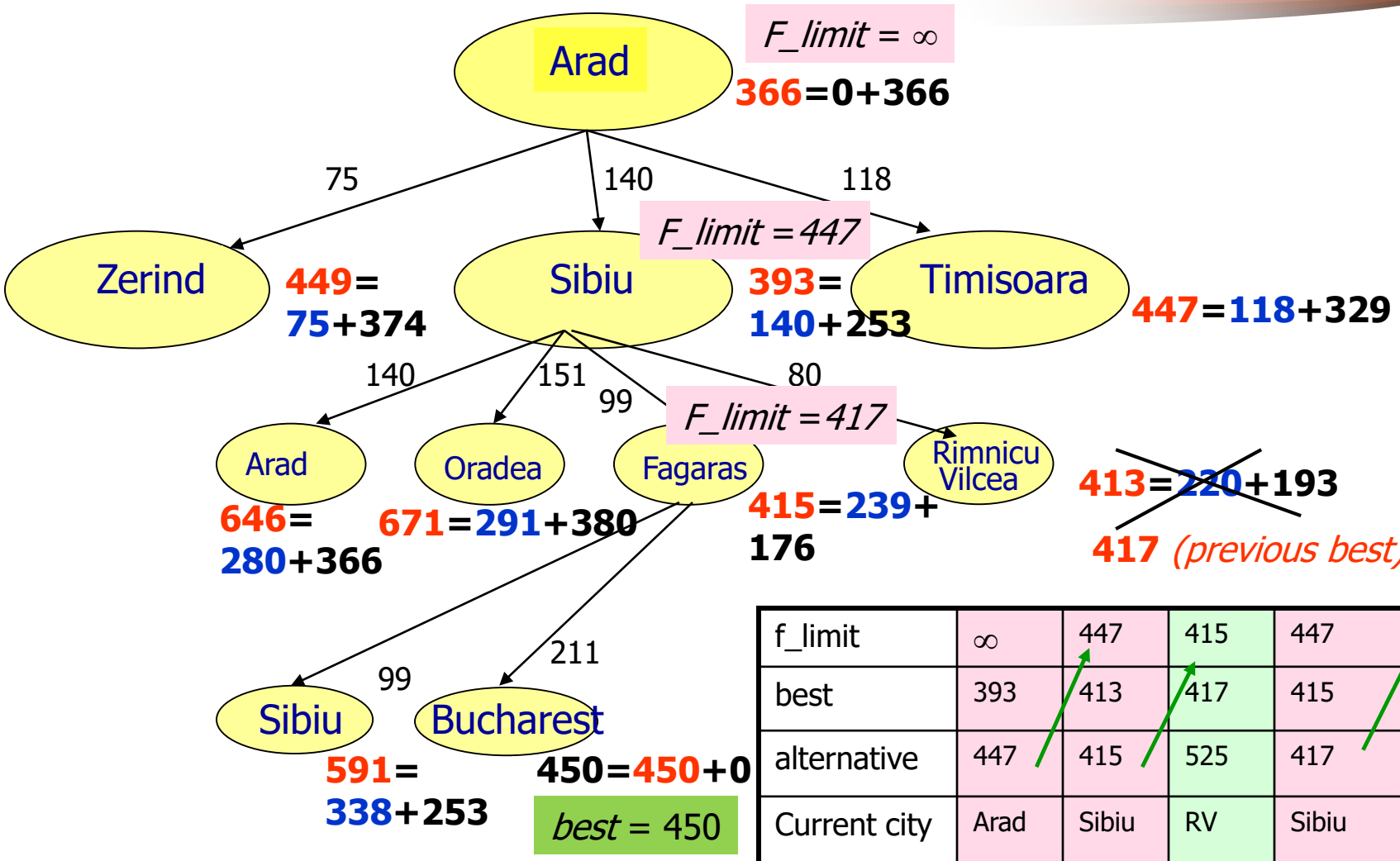
Backtrack to Sibiu!!! Current_city=Sibiu
F_limit=447; best =415; alternative=417

RBFS example -5



f_limit	∞	447	415	447
best	393	413	417	415
alternative	447	415	525	417
Current city	Arad	Sibiu	RV	Sibiu
Next city	Sibiu	RV	Fail	Fagaras

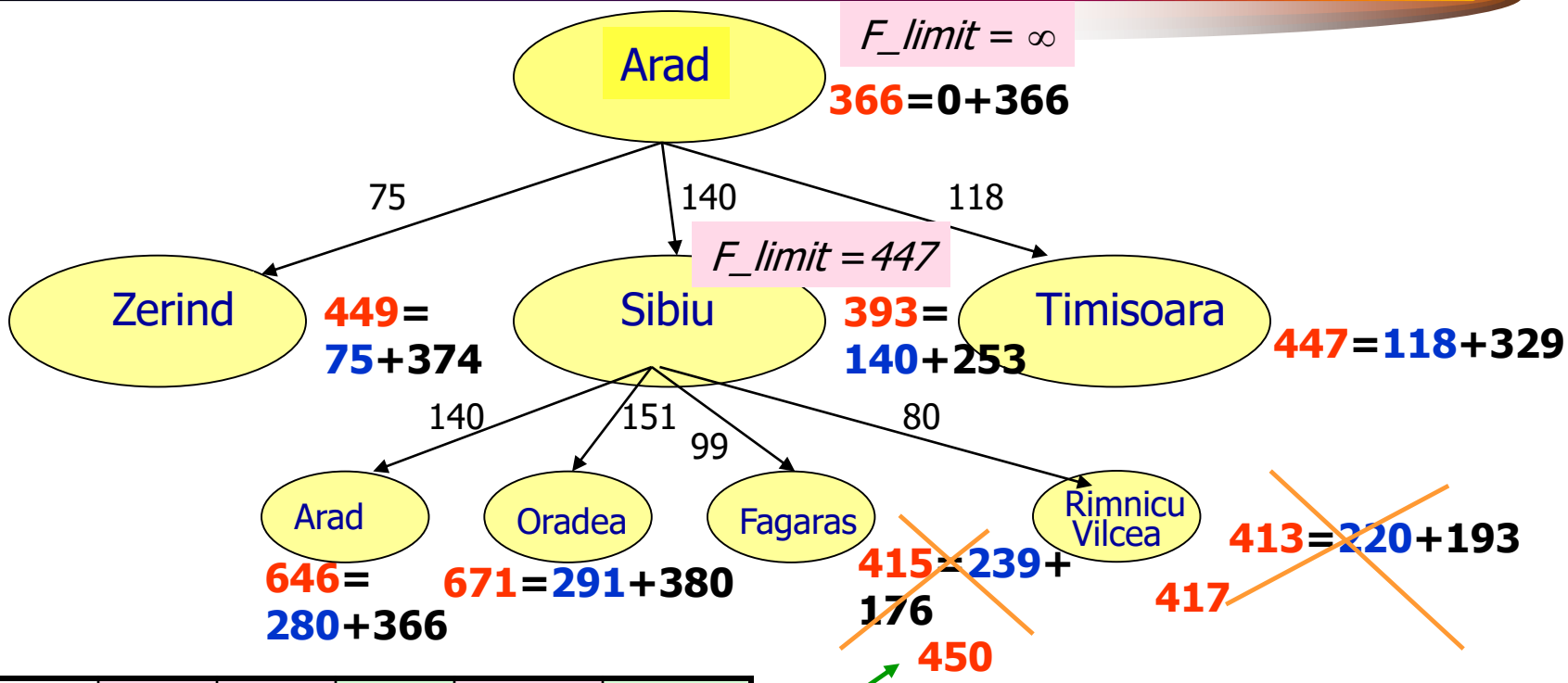
RBFS example -6



f_limit	∞	447	415	447	417
best	393	413	417	415	450
alternative	447	415	525	417	591
Current city	Arad	Sibiu	RV	Sibiu	Fagaras
Next city	Sibiu	RV	FAIL	Fagaras	FAIL

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

RBFS example -7

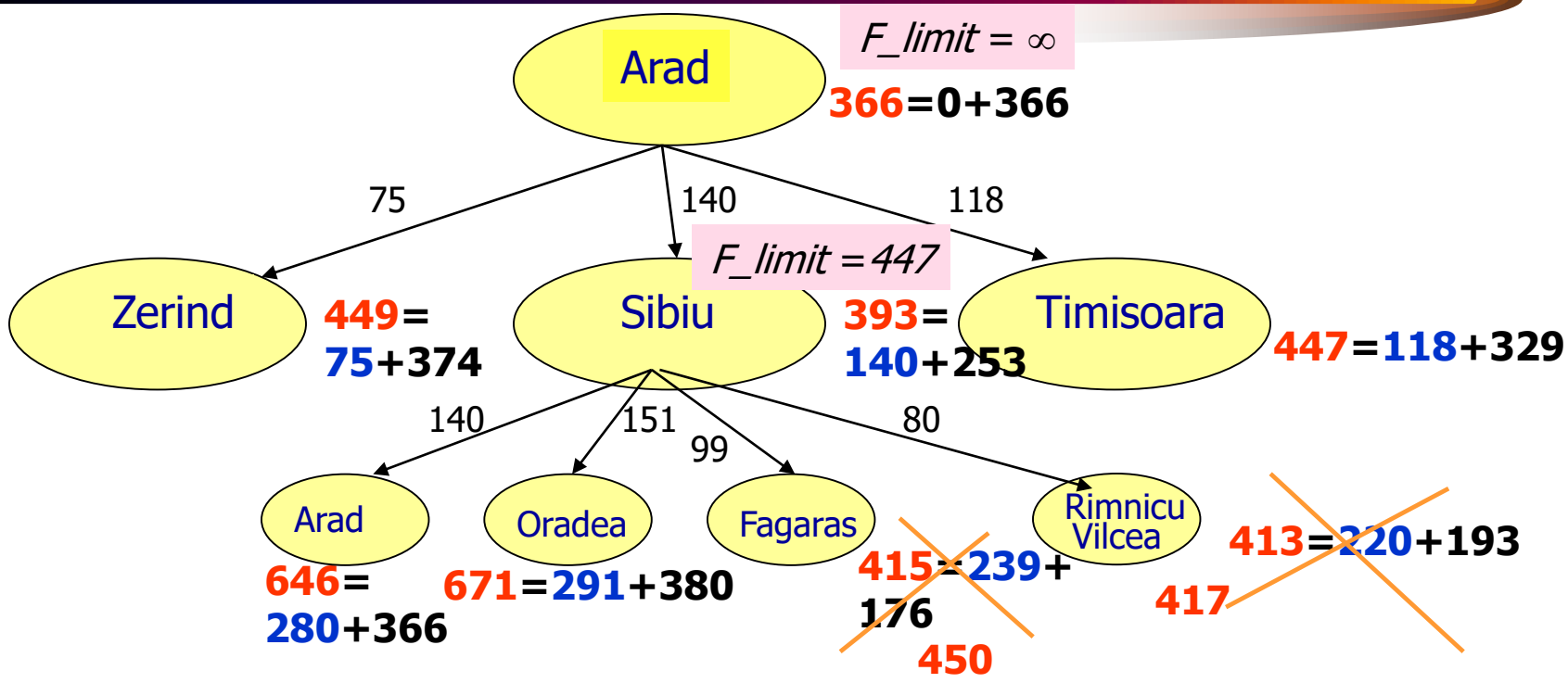


f_limit	∞	447	415	447	417
best	393	413	417	415	450
alternative	447	415	525	417	591
Current city	Arad	Sibiu	RV	Sibiu	Fagaras
Next city	Sibiu	RV	FAIL	Fagaras	FAIL

f(Fagaras) =max(f[Fagaras], f(best))=
=max(450,415)=450

Backtrack to Sibiu!!! Current_city=Sibiu
F_limit=447; best =417; alternative=450

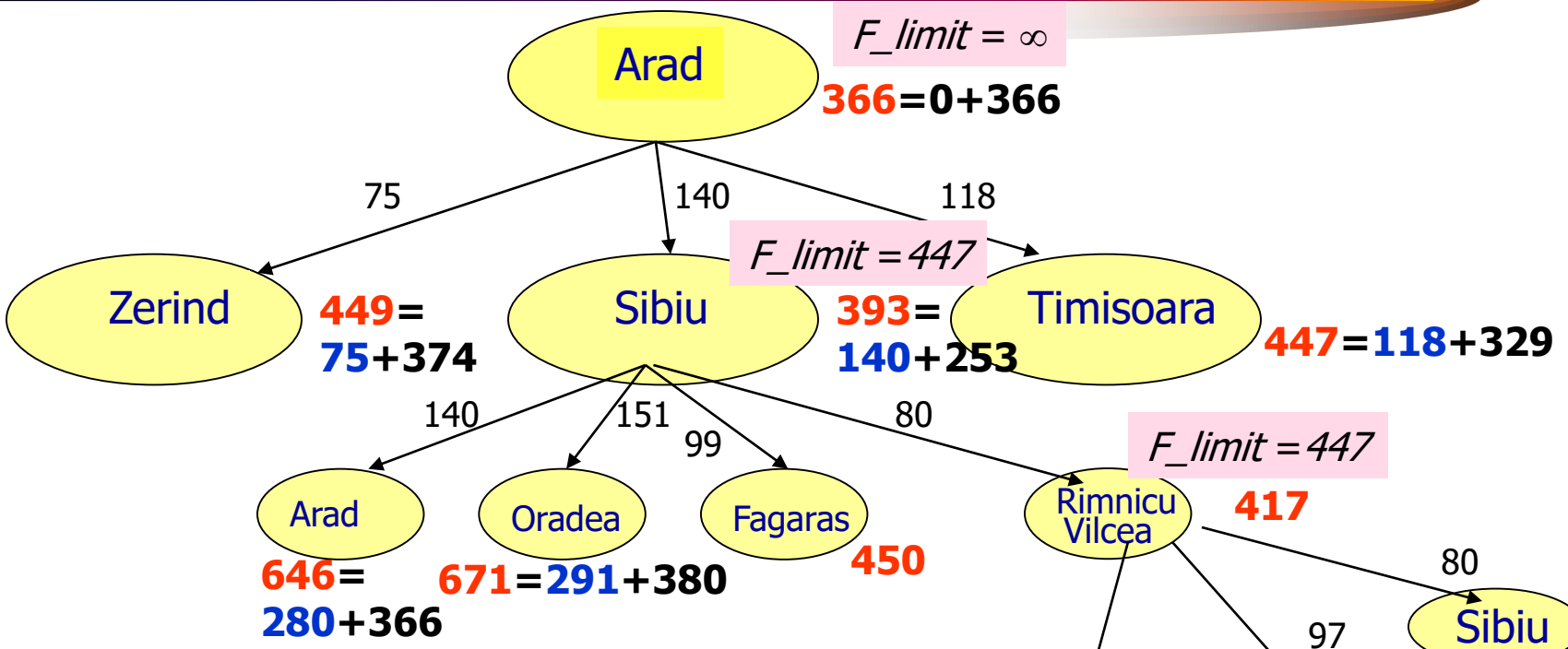
RBFS example -8



f_limit	∞	447	415	447	417	447
best	393	413	417	415	450	417
alternative	447	415	525	417	591	450
Current city	Arad	Sibiu	RV	Sibiu	Fagaras	Sibiu
Next city	Sibiu	RV	FAIL	Fagaras	FAIL	RV

F_limit for RV is:
F_limit=min(f_limit,alternative)=
=min(447,450) = 447

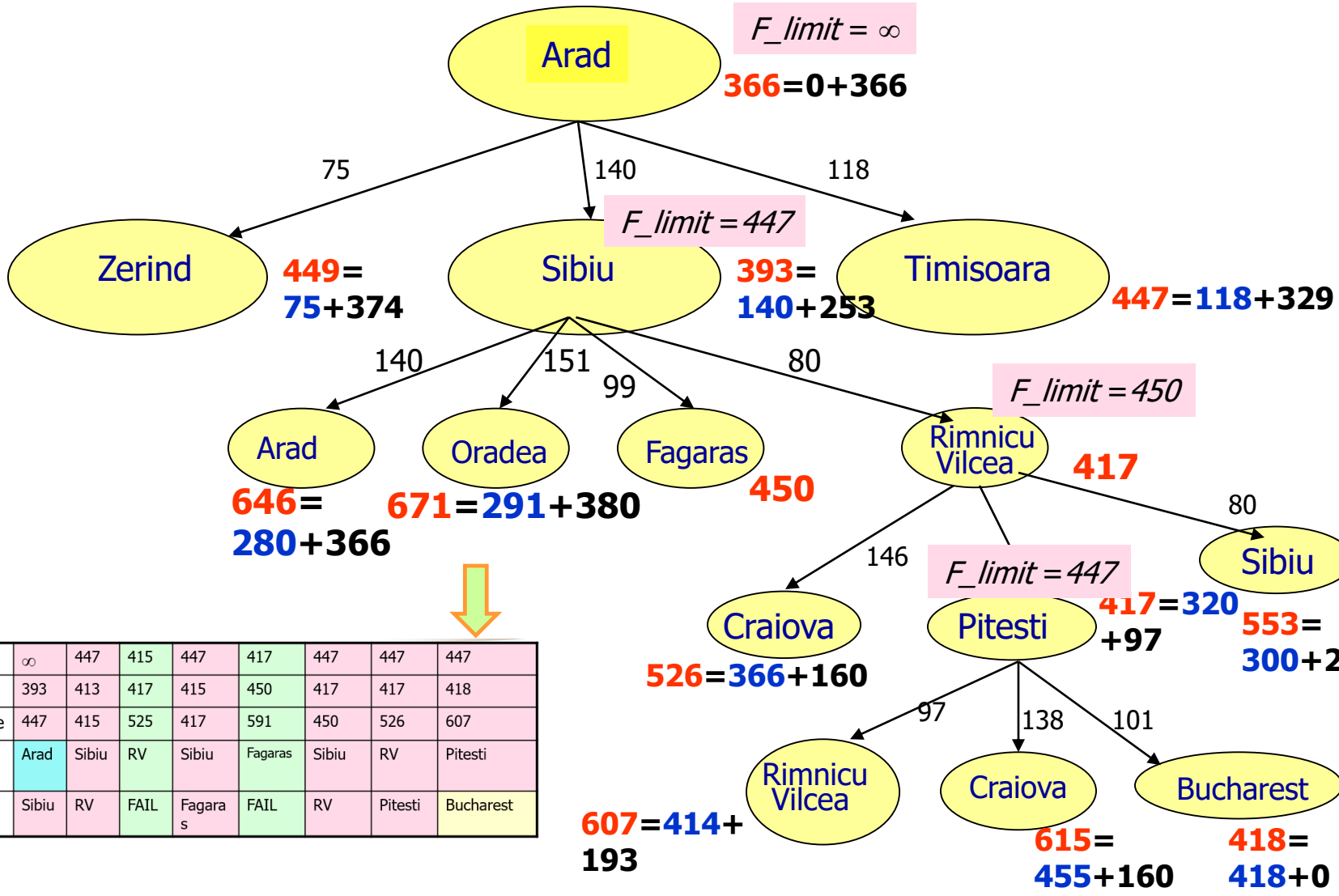
RBFS example -9



f_limit	∞	447	415	447	417	447	447
best	393	413	417	415	450	417	417
alternative	447	415	525	417	591	450	526
Current city	Arad	Sibiu	RV	Sibiu	Fagaras	Sibiu	RV
Next city	Sibiu	RV	FAIL	Fagaras	FAIL	RV	Pitesti

F_limit for Pitesti is:
F_limit=min(f_limit,alternative)=
=min(447,526) = 447

RBFS example -10



f_limit	∞	447	415	447	417	447	447	447
best	393	413	417	415	450	417	417	418
alternative	447	415	525	417	591	450	526	607
Current city	Arad	Sibiu	RV	Sibiu	Fagaras	Sibiu	RV	Pitesti
Next city	Sibiu	RV	FAIL	Fagaras	FAIL	RV	Pitesti	Bucharest

Final trace

f_limit	∞	447	415	447	417	447	447	447
best	393	413	417	415	450	417	417	418
alternative	447	415	525	417	591	450	526	607
Current city	Arad	Sibiu	RV	Sibiu	Fagaras	Sibiu	RV	Pitesti
Next city	Sibiu	RV	FAIL	Fagaras	FAIL	RV	Pitesti	Bucharest

f(RV) changed From 413 to 417

f(Fagaras) changed From 415 to 450

What did we use?

1/ best ← the lowest f-value node in *successors*

2/ alternative ← the second-lowest f-value among *successors*

3/ **if** *f[best] > f_limit* **then return** failure, *f[previous-best] changed!*

4/ f_limit ← *min(f_limit, alternative)*

5/ *f[node]* ← *max(f[node],f[best-successor])*

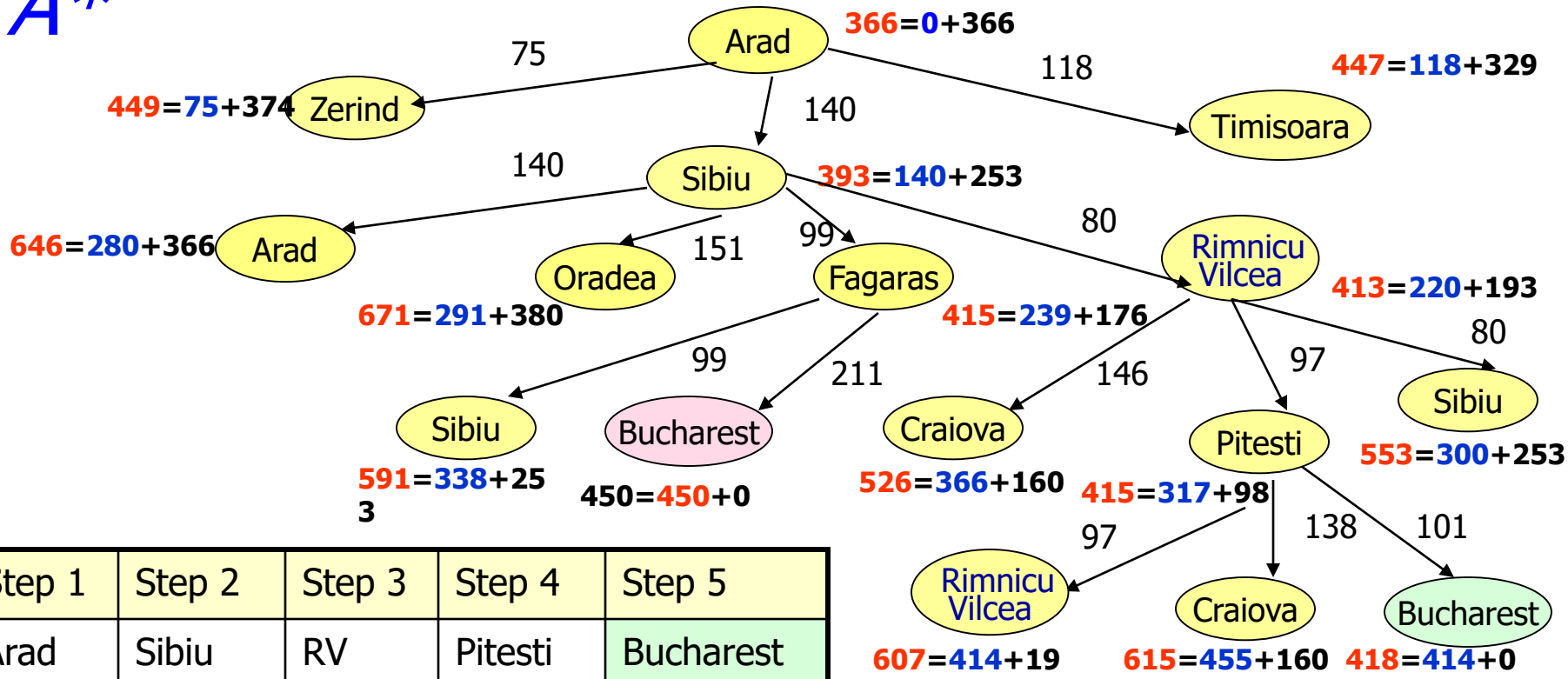
How much memory we use???

f_limit	∞	447	415	447	417	447	450	526
best	393	413	417	415	450	417	417	418
alternative	447	415	525	417	591	450	526	607
Current city	Arad	Sibiu	RV	Sibiu	Fagaras	Sibiu	RV	Pitesti
Next city	Sibiu	RV	FAIL	Fagaras	FAIL	RV	Pitesti	Bucharest

1/ best – and the best node/ next city
2/ f_limit – and the previous alternate node
3/ previous-best node
4/ previous current city
+ all the current successors = branching factor

Compare traces!!!

A*

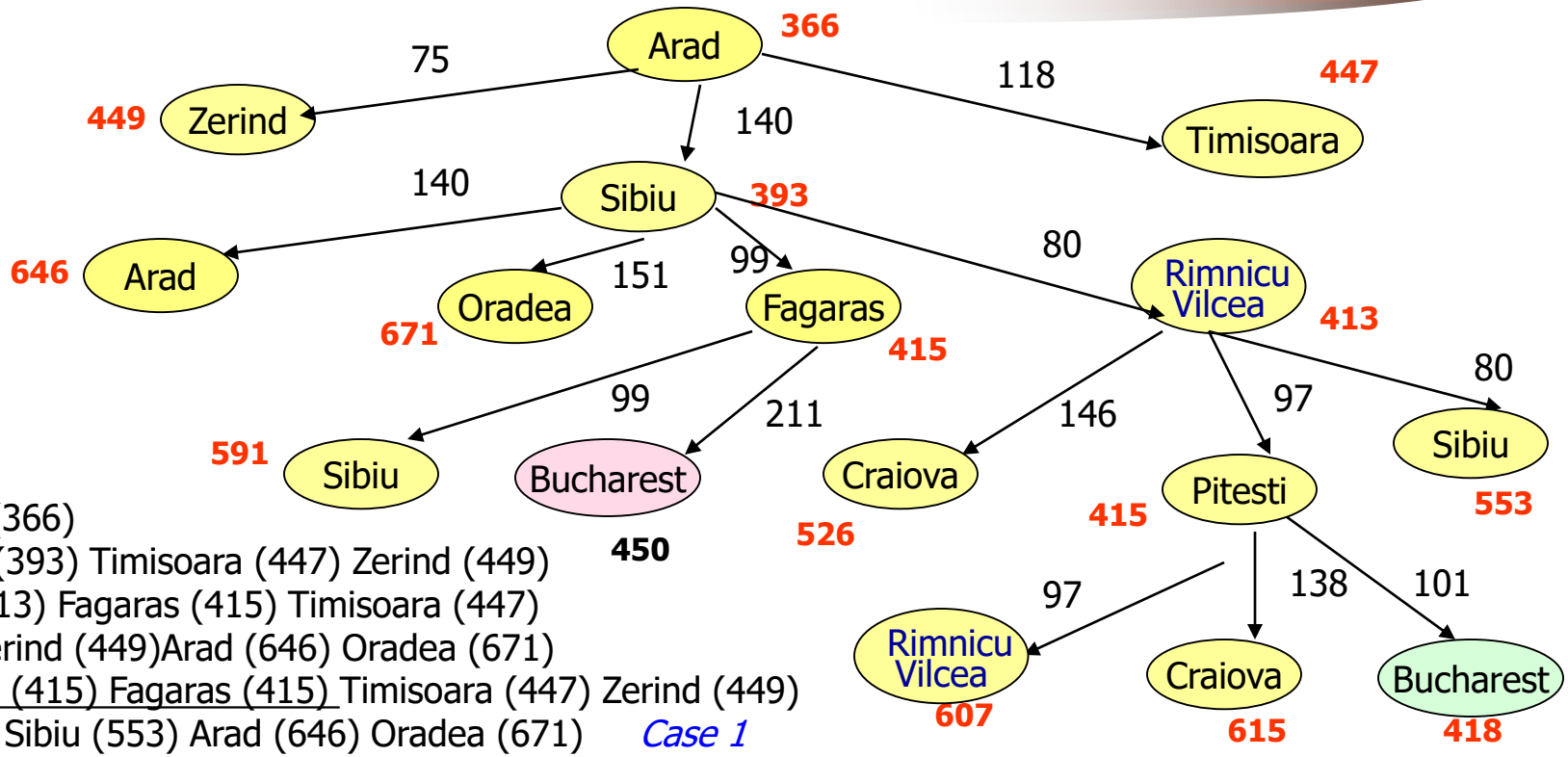


Step 1	Step 2	Step 3	Step 4	Step 5
Arad (366)	Sibiu (393)	RV (413)	Pitesti (415)	Bucharest (418)
			Fagaras (415)	Bucharest (450)

Arad	Sibiu	RV	Sibiu	Fagaras	Sibiu	RV	Pitesti	Bucharest

The frontier!!

A*



Step 1: Arad (366)
Step 2: Sibiu (393) Timisoara (447) Zerind (449)
Step 3: RV (413) Fagaras (415) Timisoara (447)
Zerind (449)Arad (646) Oradea (671)
Step 4: Pitesti (415) Fagaras (415) Timisoara (447) Zerind (449)
Craiova (526) Sibiu (553) Arad (646) Oradea (671) *Case 1*
Or : Fagaras (415) Pitesti (415) Timisoara (447) Zerind (449) Craiova (526)
Sibiu (553) Arad (646) Oradea (671) *Case 2*
Step 5: *Case 1:* Fagaras (415) Bucharest (418) Timisoara (447) Zerind (449) Craiova (526)
Sibiu (553) RV (607) Craiova (615) Arad (646) Oradea (671)
Case 2: Pitesti (415) Bucharest (450) Timisoara (447) Zerind (449) Craiova (526)
Sibiu (553) Sibiu (591) Arad (646) Oradea (671)
Step 6: **Bucharest (418)** Bucharest (450) Timisoara (447) Zerind (449) Craiova (526)
Sibiu (553) Sibiu (591) RV (607) Craiova (615) Arad (646) Oradea (671)

RBFS uses too little memory

- RBFS ends up expanding the same states many times, because it has no memory available to remember that!
- How about using all available memory??
  - Memory-Bounded A*
  - Simplified Memory-bounded A* : it cannot add a new node to the search tree without dropping one: it drops the worst leaf node (with highest f-value!).
    - Then it backs-up the value of the forgotten node to its parent (in this way the ancestor of the forgotten tree knows the quality of the best path in that subtree)

Heuristic Functions

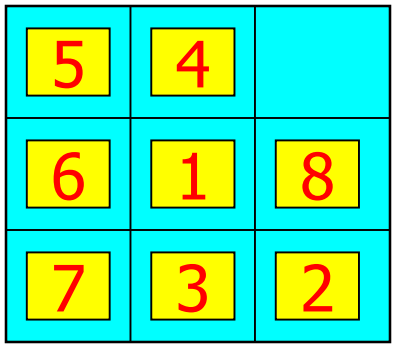
- The 8-puzzle problem generates very many states
- 362880 states is still a large number
- Solution: good heuristic that does not overestimate the number of steps to the goal

2 candidates:

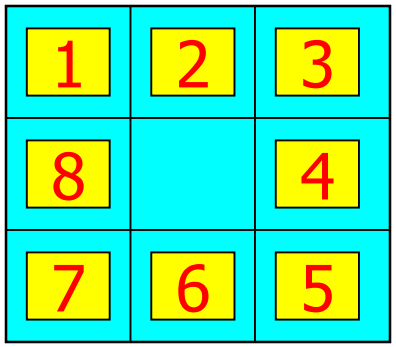
- Heuristic 1: h_1 = the number of tiles that are in the wrong position

Heuristics for the 8-Puzzle problem

For example, in the start state, $h_1=7$



Start state



Goal state

Note: Heuristic 1 is admissible because it is clear that any tile that is out of place must be moved at least once

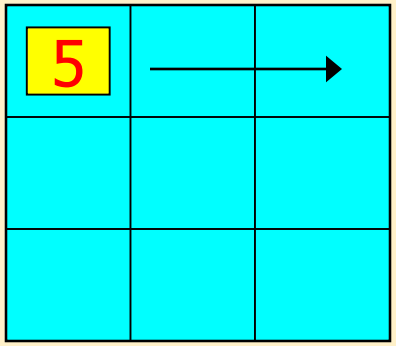
Manhattan distance

Heuristic 2: *h_2 = the sum of the distances of the tiles from their goal positions*

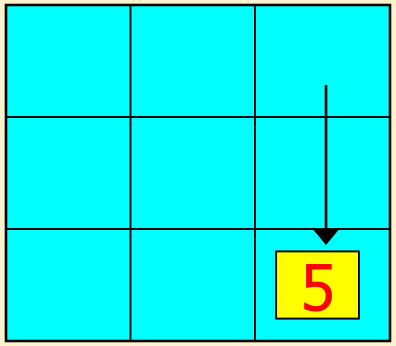
Manhattan distance

Distance = Σ (horizontal + vertical distances)

Example:



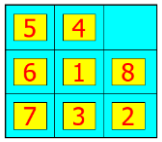
start state



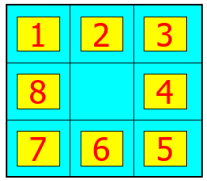
goal state

Distance = 4

For



Start state



Goal state

: $h_2 = 2+3+3+2+4+2+0+2 = 18$

The effect of heuristic accuracy on performance

- One way to characterize the quality of a heuristic is the effective branching factor

Definition: If N = the total number of nodes expanded by A* for a particular problem, and the solution depth is d, then b* is the branching factor of a uniform tree of depth d that has N nodes

Thus $N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$

Examples

A* finds a solution at depth 5 using 52 nodes b* = 1.92

Why? $N = ((b^*)^{d+1} - 1) / (b^* - 1)$

A well-designed heuristic would have b* close to 1, allowing fairly large problems to be solved

Comparison of search costs

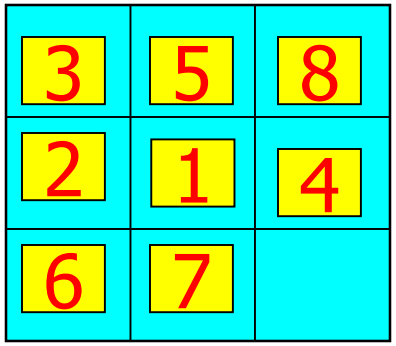
	Search cost			Effective branching factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	--	1301	211	--	1.45	1.25
18	--	3056	363	--	1.46	1.26
20	--	7276	676	--	1.47	1.27
22	--	18094	1219	--	1.48	1.28
24	--	39135	1641	--	1.48	1.26

h_2 is much better than h_1! IDS is much worse!

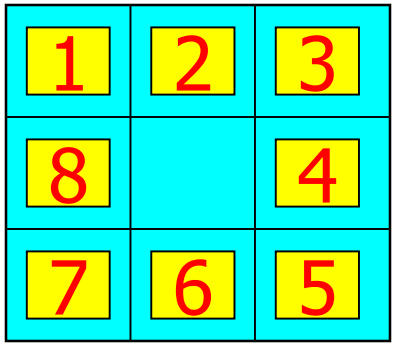
Testing heuristic functions

To test heuristics h_1 and h_2 we randomly generate problems each with solution path length 2,4,…,20 and solve them using A* search with h_1 and h_2

How do you solve the 8-puzzle??



Start state

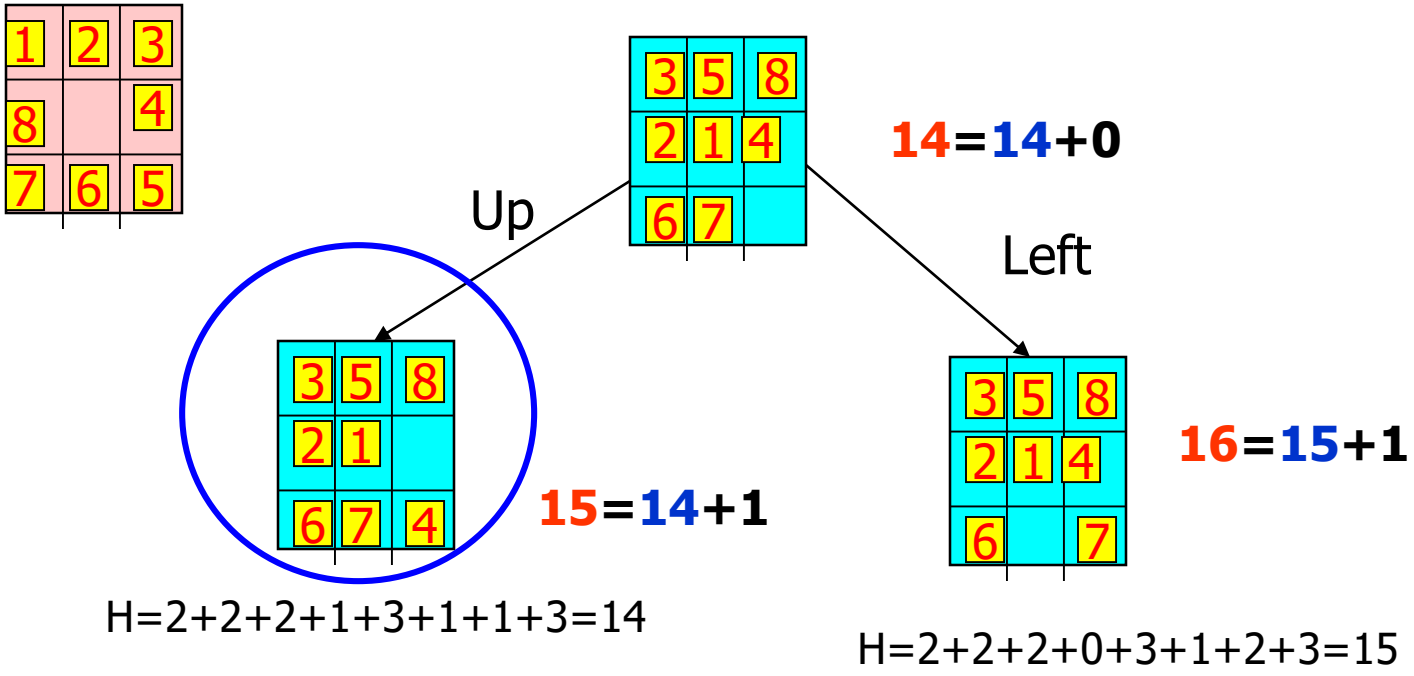


Goal state

A (with graph-search)*

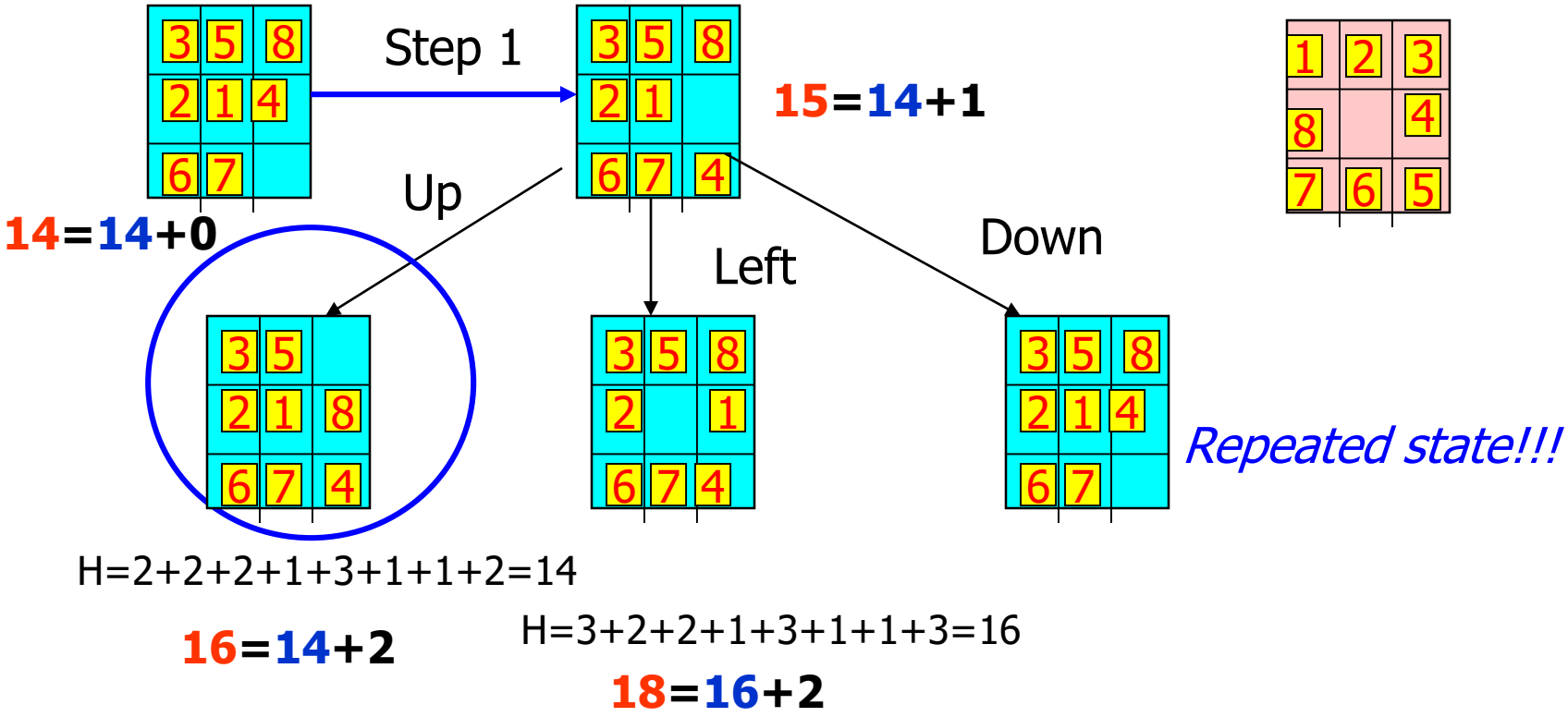
g(node) = g(father) + 1
h(node) =h2
h(init) = 2+2+2+0+3+1+1+3=14

Trace

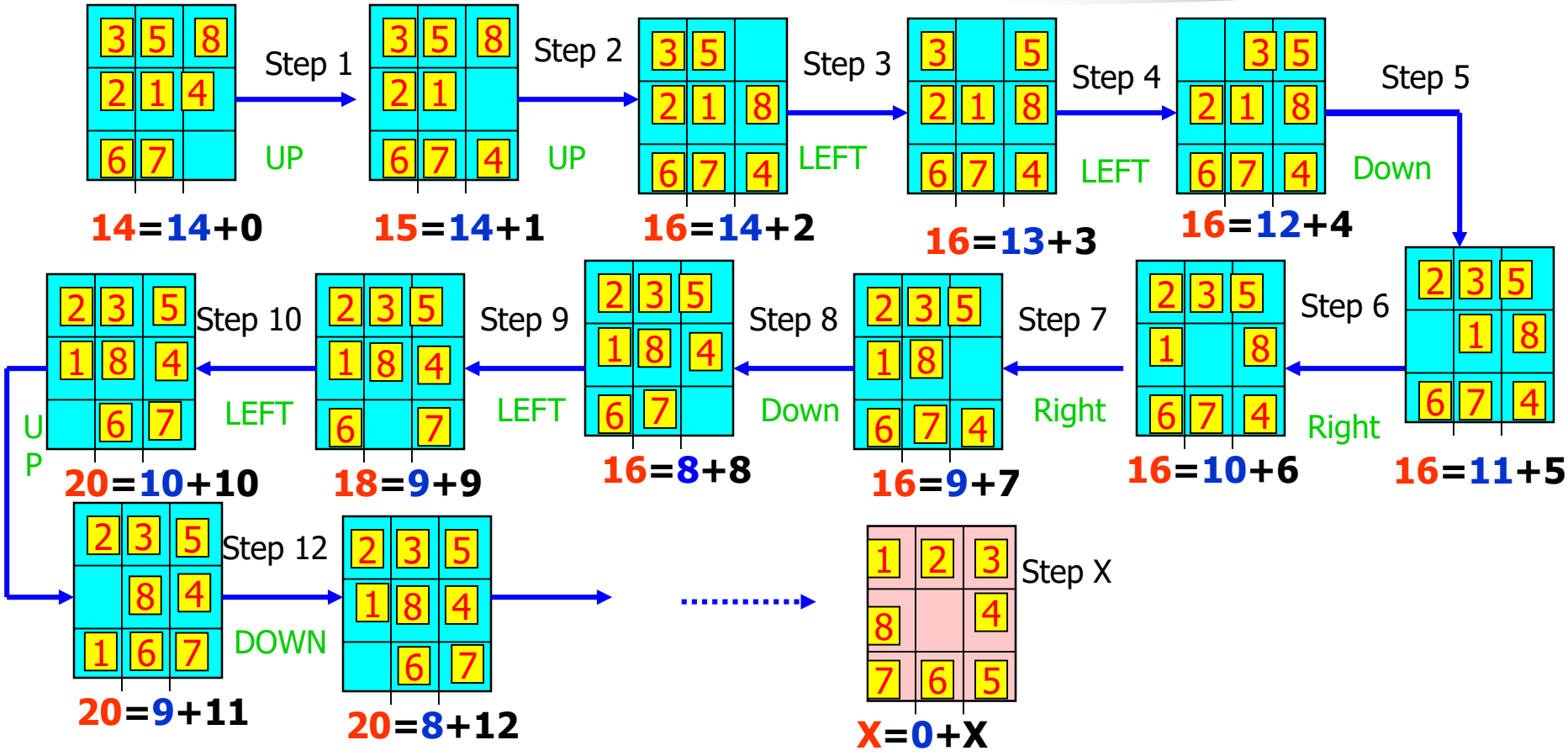


Which action to chose????

Solve: 1


Step 1
15=14+1
Up
Left
Down
14=14+0
Repeated state!!!
H=2+2+2+1+3+1+1+2=14
16=14+2
H=3+2+2+1+3+1+1+3=16
18=16+2

Solution: Starting



Inventing heuristic functions

- We have seen that both h_1 and h_2 are fairly good heuristics for the 8-puzzle, and h_2 is better than h_1
- How can we invent a heuristic function?

- h_1 and h_2 are estimates to the remaining path length for the 8-puzzle
  - they can be considered accurate path lengths for *simplified versions* of the puzzle. If a tile is moved anywhere $\rightarrow$ h_1 would give the number of steps to the shortest solution

- a problem with less restrictions is a relaxed problem

Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Admissible heuristics

- The cost of an optimal solution to a ***relaxed problem*** is an admissible heuristic for the original problem.
  - Why is it admissible?
    - The optimal solution in the original problem is also a solution to the relaxed problem!
    - An optimal solution in the original problem must be at least as expensive as the optimal solution of the relaxed problem
    - Admissible heuristics **never** overestimate the cost of reaching a goal
  - Is it also consistent?
    - *CONSISTENCY=MONOTONICITY*
    - A heuristic is consistent if for every node *n* and every successor *n'* of n generated by an action *a*, the estimated cost of reaching the goal from *n* is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from *n':*

      $h(n) \leq c(n,a,n') + h(n')$ ← The triangle inequality

Heuristic dominance

Is h_2 always better than h_1?

Yes! From the definition:

$(\forall)n \quad h_2(n) \geq h_1(n)$

h_2 dominates h_1

Dominance translates into efficiency!!

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

- Typical search costs (average number of nodes expanded):

- *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

IDS – iterative-deepening search

Lesson learned

- It is often the case that the cost of an exact solution to a relaxed problem is a good heuristic for the original problem
- One problem with generating new heuristic functions: one often fails to get the "best" heuristic

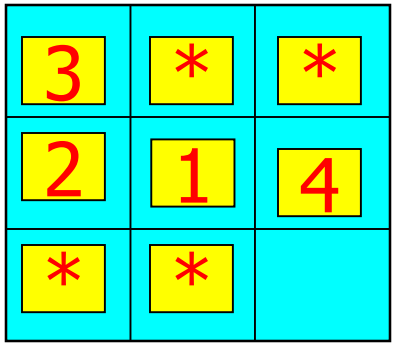
- If a collection of heuristics is available (a program called ABSOLVER can generate heuristics automatically)

  $h_1, h_2, \ldots, h_m$ *PROBLEM: none of them dominates the others, which to choose?*

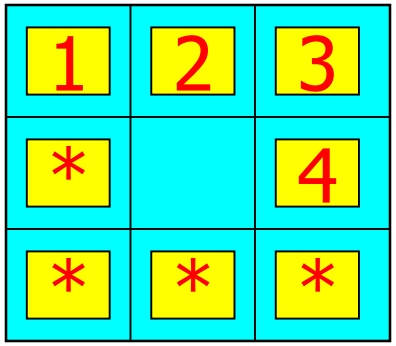
- $h(n) = \max (h_1(n), h_2(n), \ldots, h_m(n))$ *Composite heuristic*

Pattern databases

- Admissible heuristics can also be derived from a solution of a *sub-problem* of a given problem



Start state

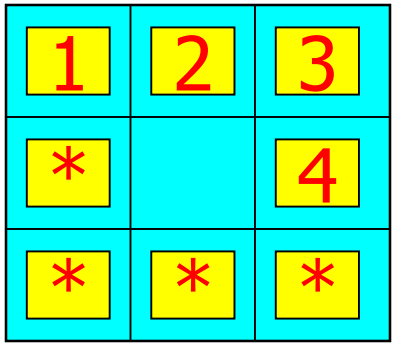


Goal state

-Find the exact solution cost to every possible sub-problem
-Store them in a **pattern database**
-Compute the admissible heuristic h_{DB} for each complete state encountered during search by looking up the corresponding sub-problem configuration in the database

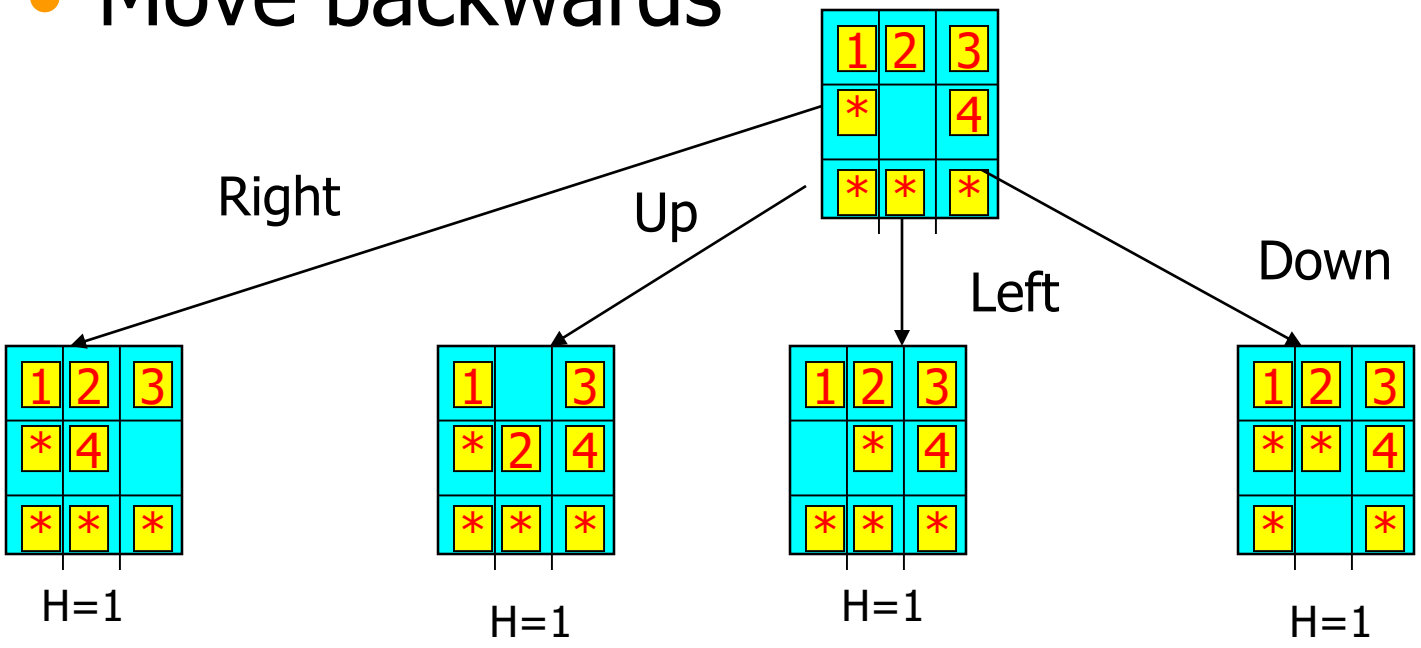
How to build the database?

- Start from the goal



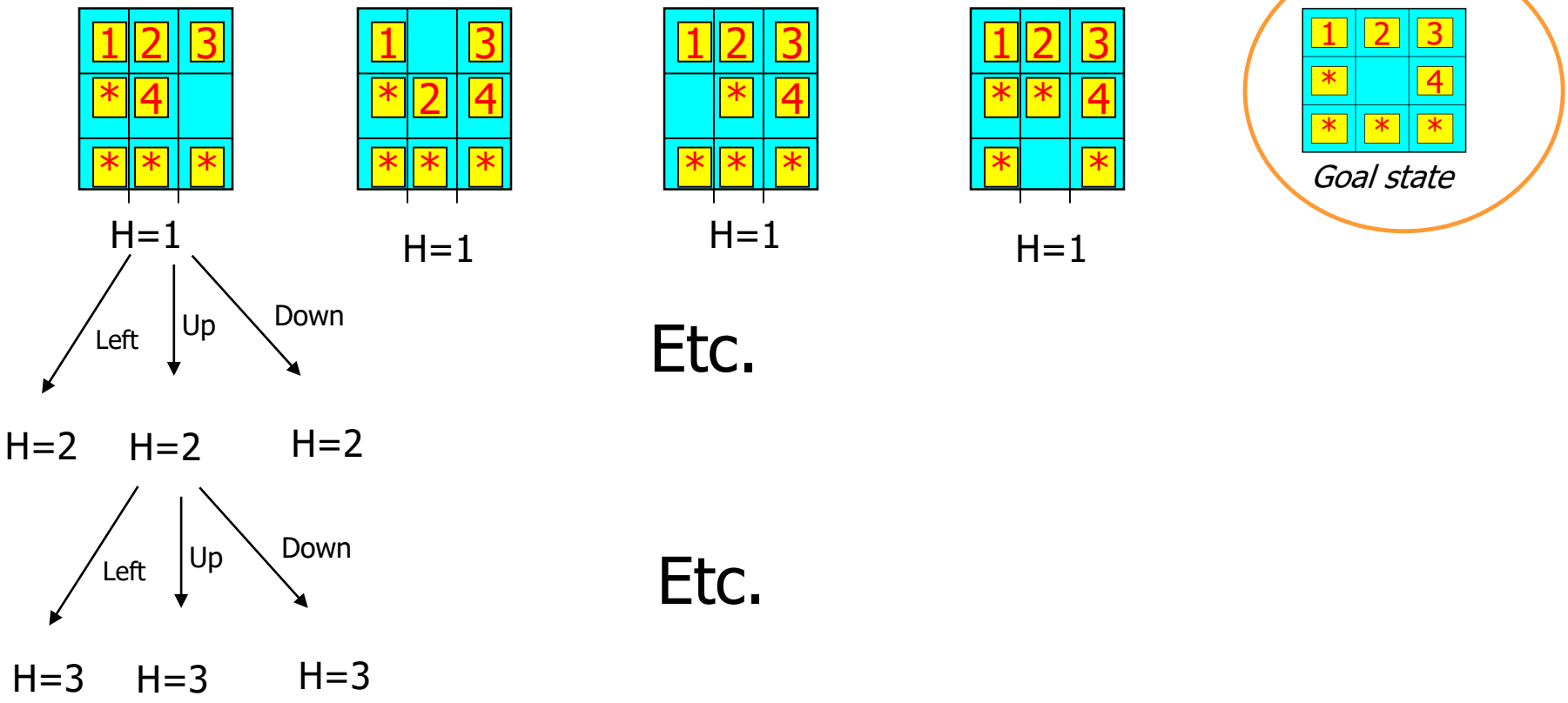
Goal state

- Move backwards



How to build the database? -2

- Continue backwards:



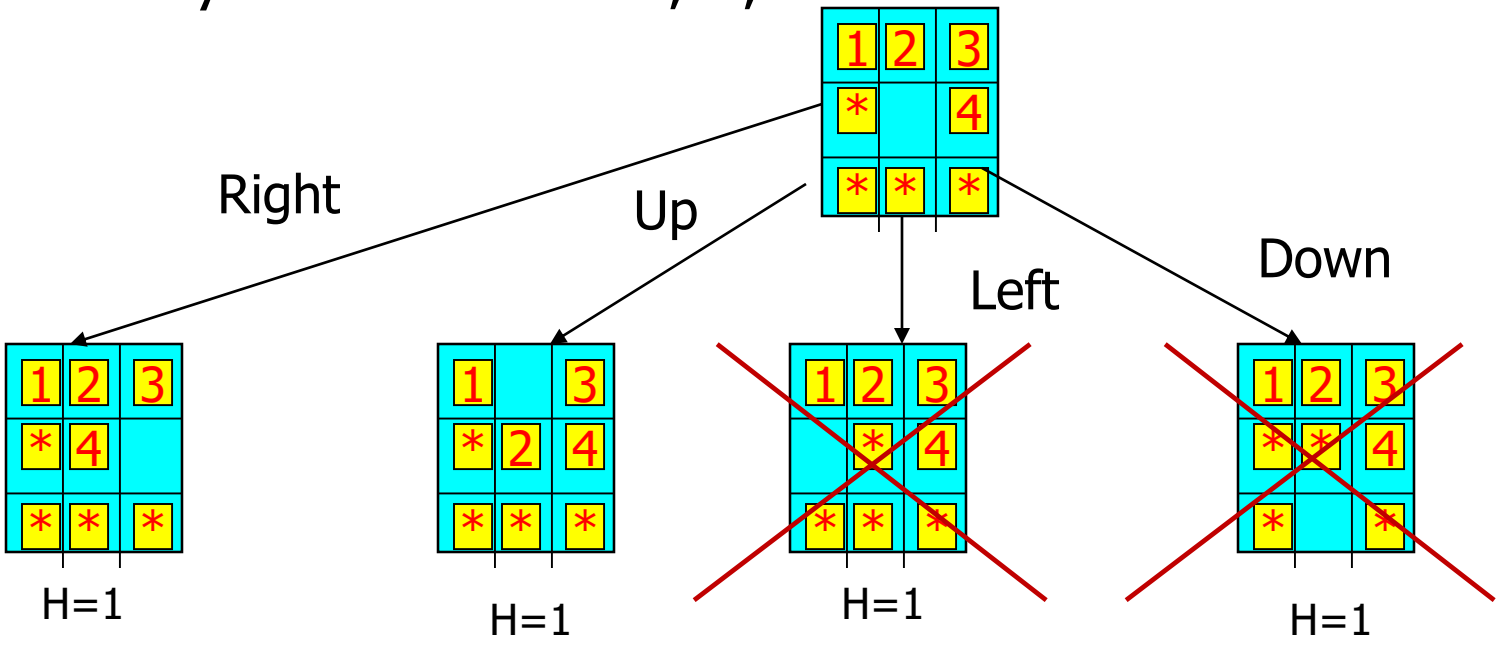
Could we combine heuristic databases???

- Yes and No!!!

- If we build database of 1-2-3-4 subproblem, DB_{1234}, and a database DB_{5678}, and one DB_{2468}, because each of them yield an admissible heuristic, we can combine them by taking the maximum value!!! ← *a composite heuristic*
- How about adding the heuristic from DB_{1234} with the heuristic from DB_{5678}??? Would we still have an admissible heuristic?
  - NO! Because it is likely that the solution to the 1-2-3-4 subproblem shares some moves (states) with the solution to the 5-6-7-8 problem, and hence it is overestimating the cost to the goal!
  - SOLUTION: Record the # moves that involve only 1-2-3-4, record the number of moves that involve only 5-6-7-8: then they do not share any common moves – the heuristic in admissible!!!

Disjoint pattern databses

- How do we build the disjoint DB_{1234}???
- Only the moves of 1,2 ,3 4!!!



- Great speedup is obtained!!! More in textbook.

Another way of inventing heuristics

- A learning algorithm can construct a function h(n) from many examples of solving a problem.
- For example: choose randomly 100 of the 8-puzzle configurations → gather statistics

Each example = a state from the solution path+ the cost of the solution from that point.

- Result: you find out that when $h_2(n)=14$ 90% of the time, the real distance to the goal is 18

Inventing heuristic functions

- You can also use statistics to come up with heuristic functions. Say you run your algorithm 100 times and find that 90% of the time when your heuristic gives a value of 14, the real value is 18. So you use 18, instead of 14.
  - The problem with this approach is that you lose optimality, because that 10% of the time, the real cost could be 15 and you have just over-estimated.

Inductive learning of the heuristics

- Use features of the states!

Example: # of misplaced tiles: feature $x_1(n)$

# of pairs of adjacent tiles that are not adjacent in the goal state: feature $x_2(n)$

Then : $h(n) = c_1x_1(n) + c_2x_2(n)$

What kind of heuristic is h(n)???

- *Admissible??? NO*
- *Consistent??? NO*