Lecture notes for *Principles of Artificial Intelligence* (COMS 4720/5720)
Yan-Bin Jia, Iowa State University

Generation of Heuristics & A* Variations

Outline

I. Generating heuristics

II. Variations of A* search

* Figures are from the textbook site (or drawn by the instructor) unless the source is specifically cited.

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

If h_2 dominates h_1, A* using h_2 will not expand more nodes than using h_1.

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

If h_2 dominates h_1, A* using h_2 will not expand more nodes than using h_1.

A* expands every node n with $f(n) = g(n) + h(n) < C^*$ (optimal cost).

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

If h_2 dominates h_1, A* using h_2 will not expand more nodes than using h_1.

A* expands every node n with $f(n) = g(n) + h(n) < C^*$ (optimal cost).

Suppose that a node n is expanded by A* with h_2, and $g(n) + h_2(n) < C^*$.

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

If h_2 dominates h_1, A* using h_2 will not expand more nodes than using h_1.

A* expands every node n with $f(n) = g(n) + h(n) < C^*$ (optimal cost).

Suppose that a node n is expanded by A* with h_2, and $g(n) + h_2(n) < C^*$.

$h_1(n) \leq h_2(n)$

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

If h_2 dominates h_1, A* using h_2 will not expand more nodes than using h_1.

A* expands every node n with $f(n) = g(n) + h(n) < C^*$ (optimal cost).

Suppose that a node n is expanded by A* with h_2, and $g(n) + h_2(n) < C^*$.

$h_1(n) \leq h_2(n) \quad \Rightarrow \quad g(n) + h_1(n) \leq g(n) + h_2(n)$

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

If h_2 dominates h_1, A* using h_2 will not expand more nodes than using h_1.

A* expands every node n with $f(n) = g(n) + h(n) < C^*$ (optimal cost).

Suppose that a node n is expanded by A* with h_2, and $g(n) + h_2(n) < C^*$.

$$h_1(n) \leq h_2(n) \Rightarrow g(n) + h_1(n) \leq g(n) + h_2(n) \Rightarrow g(n) + h_1(n) < C^*$$

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

If h_2 dominates h_1, A* using h_2 will not expand more nodes than using h_1.

A* expands every node n with $f(n) = g(n) + h(n) < C^*$ (optimal cost).

Suppose that a node n is expanded by A* with h_2, and $g(n) + h_2(n) < C^*$.

$$h_1(n) \leq h_2(n) \Rightarrow g(n) + h_1(n) \leq g(n) + h_2(n) \Rightarrow g(n) + h_1(n) < C^*$$

The node n will be expanded by A* with h_1.

I. Better Heuristics

Given two *admissible* heuristic functions h_1 and h_2, we say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ at every node n.

If h_2 dominates h_1, A* using h_2 will not expand more nodes than using h_1.

A* expands every node n with $f(n) = g(n) + h(n) < C^*$ (optimal cost).

Suppose that a node n is expanded by A* with h_2, and $g(n) + h_2(n) < C^*$.

$$h_1(n) \leq h_2(n) \Rightarrow g(n) + h_1(n) \leq g(n) + h_2(n) \Rightarrow g(n) + h_1(n) < C^*$$

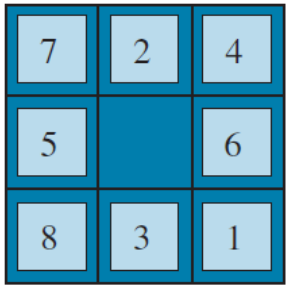
The node n will be expanded by A* with h_1.

♣ h_1 might cause other nodes to be expanded as well.

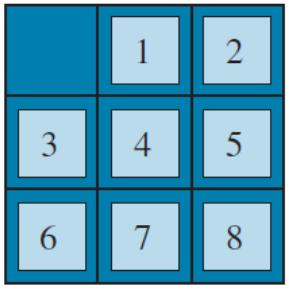
Generating Heuristics by Relaxation

- An admissible heuristic can be derived from exact solution cost of a *relaxed problem*.

with fewer restrictions on the actions



Start State



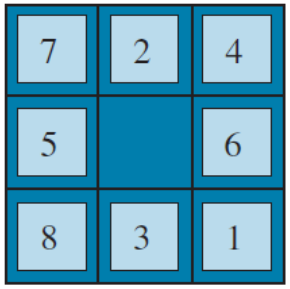
Goal State

Generating Heuristics by Relaxation

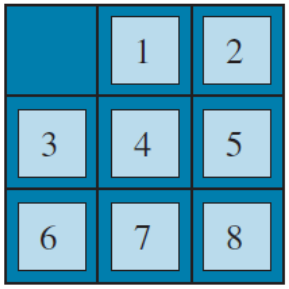
- An admissible heuristic can be derived from exact solution cost of a *relaxed problem*.

with fewer restrictions on the actions

Relaxation 1: A tile can move anywhere.



Start State



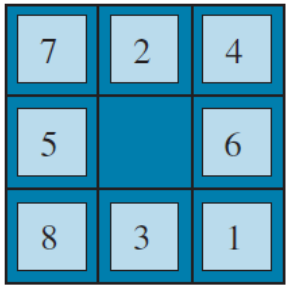
Goal State

Generating Heuristics by Relaxation

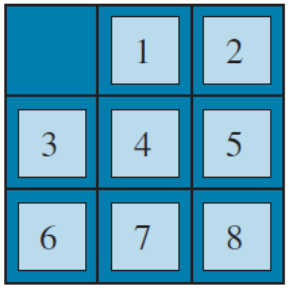
- An admissible heuristic can be derived from exact solution cost of a *relaxed problem*.

with fewer restrictions on the actions

Relaxation 1: A tile can move anywhere.



Start State



Goal State

h_1 = # tiles misplaced

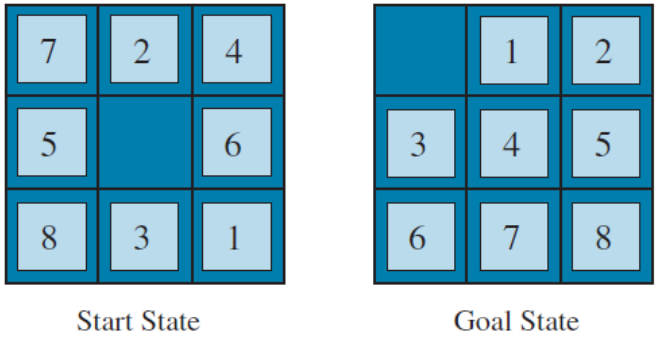
Generating Heuristics by Relaxation

- An admissible heuristic can be derived from exact solution cost of a *relaxed problem*.

↑ with fewer restrictions on the actions

Relaxation 1: A tile can move anywhere.

⇩ h_1 = # tiles misplaced

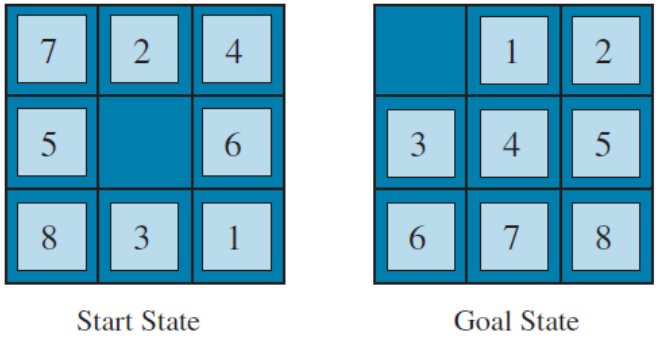


h_1 would give the length of the shortest solution.

Generating Heuristics by Relaxation

- An admissible heuristic can be derived from exact solution cost of a *relaxed problem*.

↑ with fewer restrictions on the actions



Relaxation 1: A tile can move anywhere.

⇩ h_1 = # tiles misplaced

h_1 would give the length of the shortest solution.

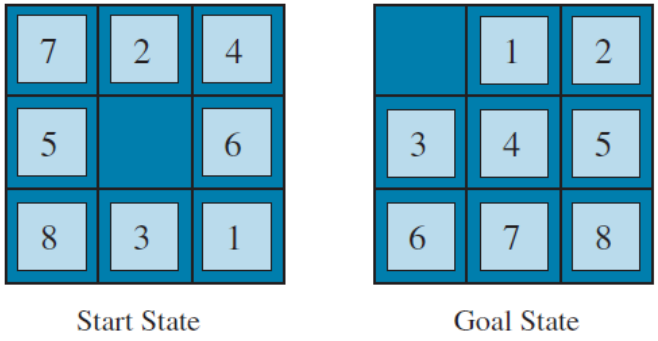
Relaxation 2: A tile can move one square in any direction, even onto an occupied square.

Generating Heuristics by Relaxation

- An admissible heuristic can be derived from exact solution cost of a *relaxed problem*.

↑

with fewer restrictions on the actions



Start State Goal State

Relaxation 1: A tile can move anywhere.

⇩ h_1 = # tiles misplaced

h_1 would give the length of the shortest solution.

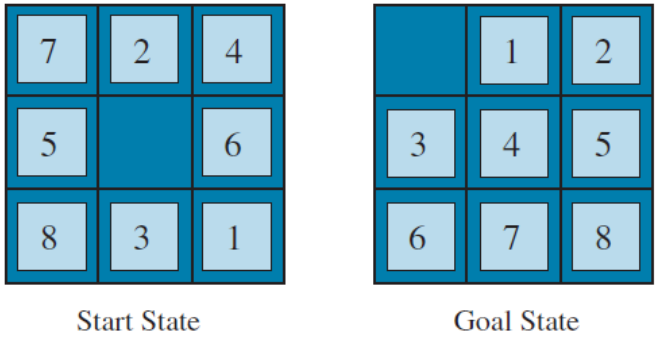
Relaxation 2: A tile can move one square in any direction, even onto an occupied square.

h_2 = sum of Manhattan distances of the tiles from their goal positions

Generating Heuristics by Relaxation

- An admissible heuristic can be derived from exact solution cost of a *relaxed problem*.

↑ with fewer restrictions on the actions



Relaxation 1: A tile can move anywhere.

⇩ h_1 = # tiles misplaced

h_1 would give the length of the shortest solution.

Relaxation 2: A tile can move one square in any direction, even onto an occupied square.

⇩ h_2 = sum of Manhattan distances of the tiles from their goal positions

h_2 would give the length of the shortest solution.

Admissibility & Consistency

- A solution in the original problem is also a solution in the relaxed problem.

Admissibility & Consistency

- A solution in the original problem is also a solution in the relaxed problem.
- But an optimal solution to the relaxed problem may be shorter than an optimal solution to the original problem.

Admissibility & Consistency

- A solution in the original problem is also a solution in the relaxed problem.
- But an optimal solution to the relaxed problem may be shorter than an optimal solution to the original problem.

The cost of an optimal solution to the relaxed problem is an admissible heuristic for the original problem.

Admissibility & Consistency

- A solution in the original problem is also a solution in the relaxed problem.
- But an optimal solution to the relaxed problem may be shorter than an optimal solution to the original problem.

⇩

The cost of an optimal solution to the relaxed problem is an admissible heuristic for the original problem.

- Such heuristic is an exact cost for the relaxed problem.

Admissibility & Consistency

- A solution in the original problem is also a solution in the relaxed problem.
- But an optimal solution to the relaxed problem may be shorter than an optimal solution to the original problem.

⇩

The cost of an optimal solution to the relaxed problem is an admissible heuristic for the original problem.

- Such heuristic is an exact cost for the relaxed problem.

⇩

It must satisfy the triangle inequality.

Admissibility & Consistency

- A solution in the original problem is also a solution in the relaxed problem.
- But an optimal solution to the relaxed problem may be shorter than an optimal solution to the original problem.

⇩

The cost of an optimal solution to the relaxed problem is an admissible heuristic for the original problem.

- Such heuristic is an exact cost for the relaxed problem.

⇩

It must satisfy the triangle inequality.

⇩

The heuristic is consistent.

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

> A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

(a) A tile can move from square X to square Y if X is adjacent to Y.

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

(a) A tile can move from square X to square Y if X is adjacent to Y. ⇨ h_2 (Manhattan distance)

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

(a) A tile can move from square X to square Y if X is adjacent to Y. ⇨ h_2 (Manhattan distance)

(b) A tile can move from square X to square Y if Y is blank.

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

(a) A tile can move from square X to square Y if X is adjacent to Y. ⇨ h_2 (Manhattan distance)

(b) A tile can move from square X to square Y if Y is blank.

(c) A tile can move from square X to square Y.

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

(a) A tile can move from square X to square Y if X is adjacent to Y. ⇨ h_2 (Manhattan distance)

(b) A tile can move from square X to square Y if Y is blank.

(c) A tile can move from square X to square Y. ⇨ h_1 (misplaced tiles)

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

(a) A tile can move from square X to square Y if X is adjacent to Y. ⇨ h_2 (Manhattan distance)

(b) A tile can move from square X to square Y if Y is blank.

(c) A tile can move from square X to square Y. ⇨ h_1 (misplaced tiles) → Allows decomposition of the problem into 8 independent subproblems, one for moving each tile.

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

(a) A tile can move from square X to square Y if X is adjacent to Y. ⇨ h_2 (Manhattan distance)

(b) A tile can move from square X to square Y if Y is blank.

(c) A tile can move from square X to square Y. ⇨ h_1 (misplaced tiles)

↓

Allows decomposition of the problem into 8 independent subproblems, one for moving each tile.

- The relaxed problems should be solved without search.

  Otherwise, evaluation of the corresponding heuristic will be expensive.

Heuristics from Formal Specification

Formal specification of a problem (8-puzzle):

A tile can move from square X to square Y if X is adjacent to Y and Y is blank.

⇩ Relaxation by removing one or two conditions

(a) A tile can move from square X to square Y if X is adjacent to Y. ⇨ h_2 (Manhattan distance)

(b) A tile can move from square X to square Y if Y is blank.

(c) A tile can move from square X to square Y. ⇨ h_1 (misplaced tiles) → Allows decomposition of the problem into 8 independent subproblems, one for moving each tile.

- The relaxed problems should be solved without search.
  Otherwise, evaluation of the corresponding heuristic will be expensive.
- Program ABSOLVER generates heuristics automatically from problem definitions, including the best one for the 8-puzzle and the first one for the Rubik's Cube puzzle.

Multiple Heuristics Available

Admissible heuristics $h_1, h_2, \cdots, h_k$ are available but none is clearly better than the others.

Use a composite heuristic:

$$h(n) = \max\{h_1(n), h_2(n), \cdots, h_k(n)\}$$

Multiple Heuristics Available

Admissible heuristics $h_1, h_2, \cdots, h_k$ are available but none is clearly better than the others.

Use a composite heuristic:

$$h(n) = \max\{h_1(n), h_2(n), \cdots, h_k(n)\}$$

- h is *admissible* because $h_1, h_2, \cdots, h_k$ are.

Multiple Heuristics Available

Admissible heuristics $h_1, h_2, \cdots, h_k$ are available but none is clearly better than the others.

Use a composite heuristic:

$$h(n) = \max\{h_1(n), h_2(n), \cdots, h_k(n)\}$$

- h is *admissible* because $h_1, h_2, \cdots, h_k$ are.
- h *dominates* $h_1, h_2, \cdots, h_k$.

Multiple Heuristics Available

Admissible heuristics $h_1, h_2, \cdots, h_k$ are available but none is clearly better than the others.

Use a composite heuristic:

$$h(n) = \max\{h_1(n), h_2(n), \cdots, h_k(n)\}$$

- ♦ h is *admissible* because $h_1, h_2, \cdots, h_k$ are.
- ♦ h *dominates* $h_1, h_2, \cdots, h_k$.
- ♣ h takes longer to compute – at least $O(k)$.
  May randomly select one heuristic at each evaluation.

II. Sacrificing Search

- A* explores a lot of nodes due to equal weighting of g and h in $f = g + h$ which often distracts it from the optimal path.
- Can explore fewer nodes if we are okay with satisficing (suboptimal but “good enough”) solutions .

II. Sacrificing Search

- A* explores a lot of nodes due to equal weighting of g and h in $f = g + h$ which often distracts it from the optimal path.
- Can explore fewer nodes if we are okay with satisficing (suboptimal but “good enough”) solutions .

Use an inadmissible heuristic.

II. Sacrificing Search

- A* explores a lot of nodes due to equal weighting of g and h in $f = g + h$ which often distracts it from the optimal path.
- Can explore fewer nodes if we are okay with satisficing (suboptimal but "good enough") solutions .

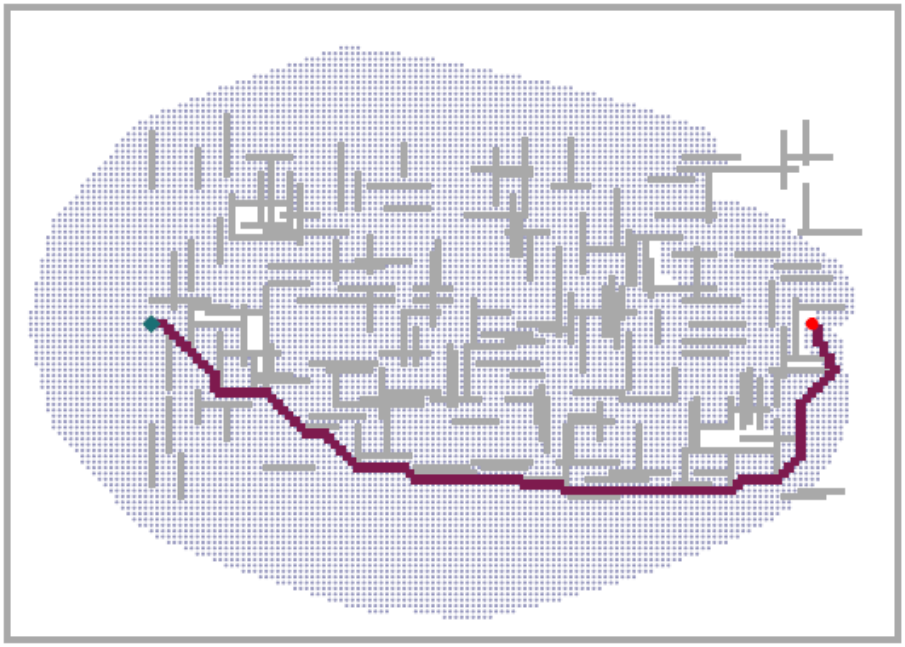
Use an inadmissible heuristic.

Detour index: multiplier applied to the straight-line distance.

e.g., a detour index of 1.3 implies a good estimate of 13 miles between two cities that are 10 miles apart.

Weighted A*

Evaluation function: $f(n) = g(n) + W \times h(n)$ for some $W > 1$.

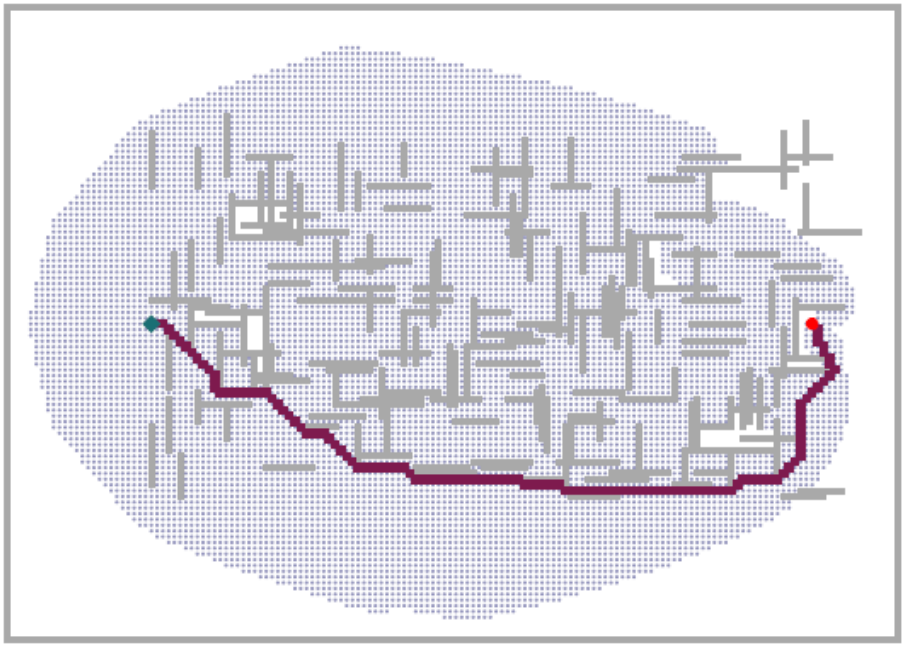


A* search

Gray bars: obstacles
Dots: reached states

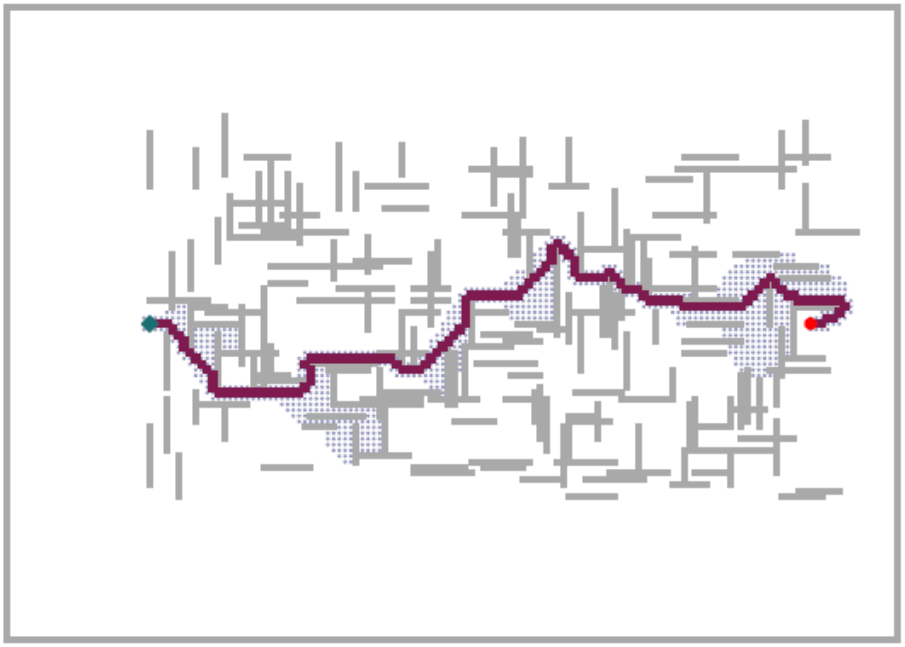
Weighted A*

Evaluation function: $f(n) = g(n) + W \times h(n)$ for some $W > 1$.



A* search

Gray bars: obstacles
Dots: reached states



Weighted A* search
($W = 2$ on the same grid)

Weighted A* As a Generalization

- Weighted A* finds a solution with cost between C^* and $W \times C^*$.
- Cost is usually much closer to C^* in practice.

Weighted A* search $g(n) + W \times h(n)$ $(1 \leq W < \infty)$

Weighted A* As a Generalization

- Weighted A* finds a solution with cost between C^* and $W \times C^*$.
- Cost is usually much closer to C^* in practice.

A* search	$g(n) + h(n)$	$(W = 1)$
Uniform-cost search	$g(n)$	$(W = 0)$
Best-cost search	$h(n)$	$(W = \infty)$
Weighted A* search	$g(n) + W \times h(n)$	$(1 \leq W < \infty)$

Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

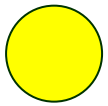
Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

$f = 5$

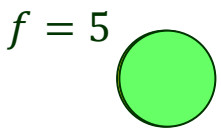


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

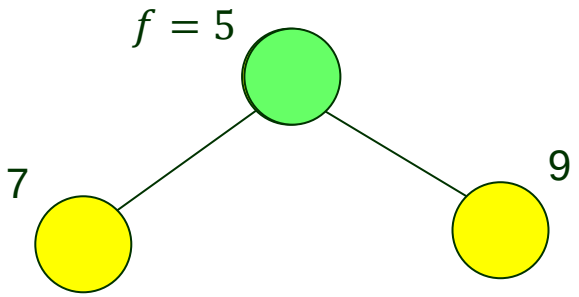


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

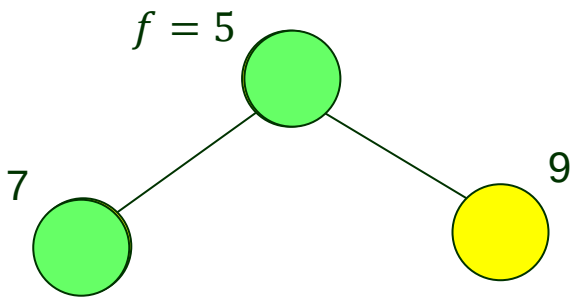


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

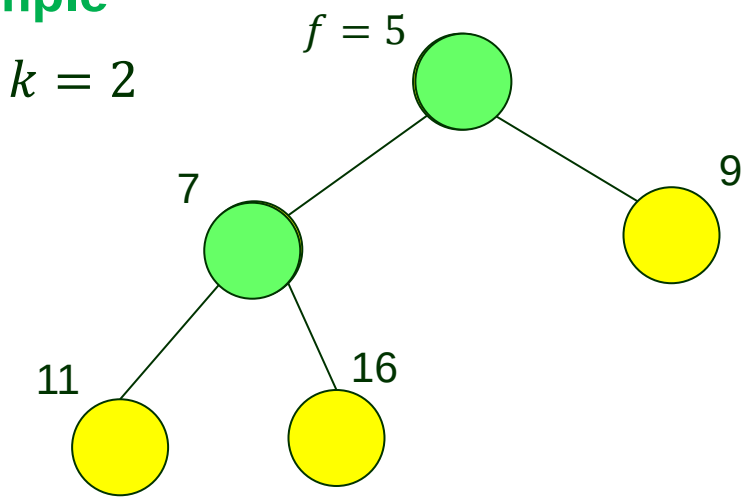


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

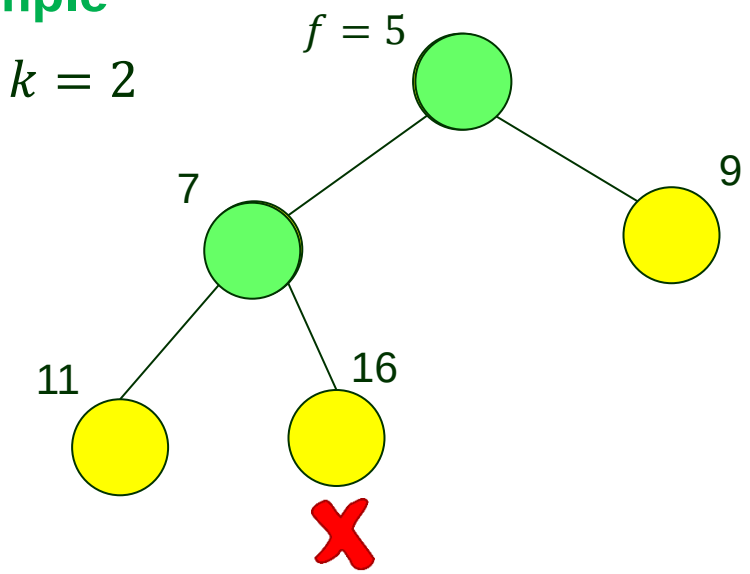


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

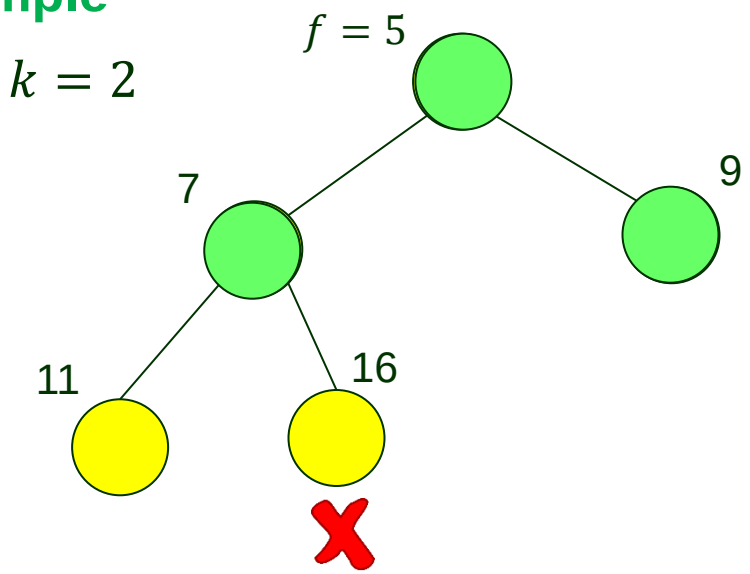


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

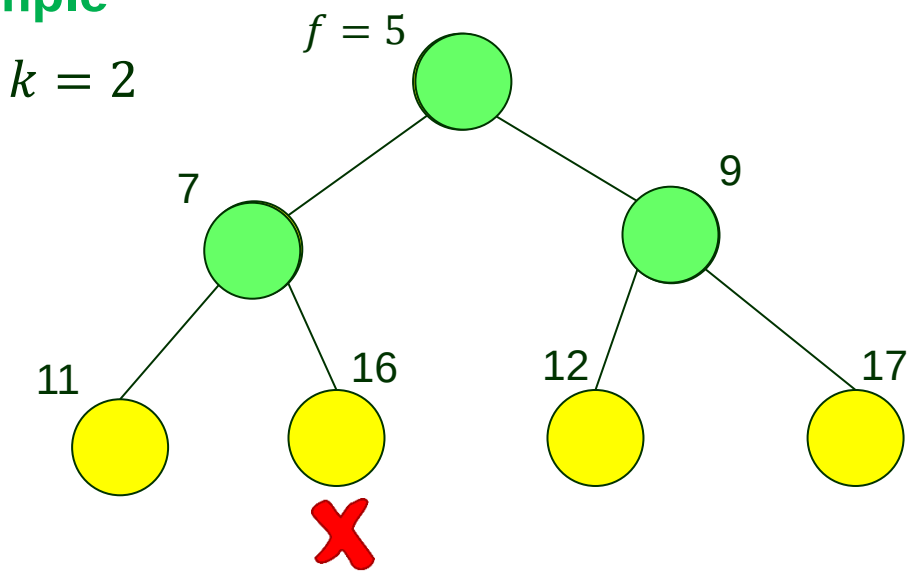


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

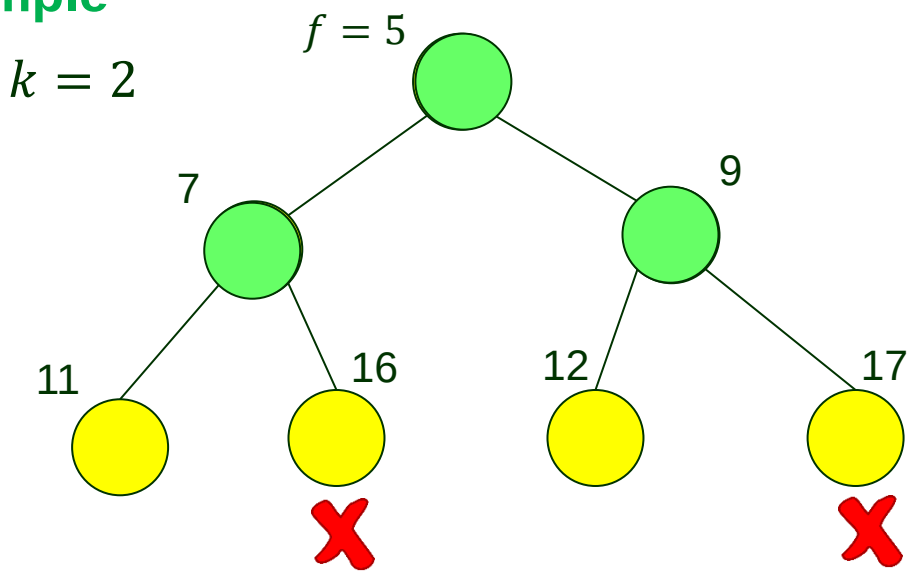


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

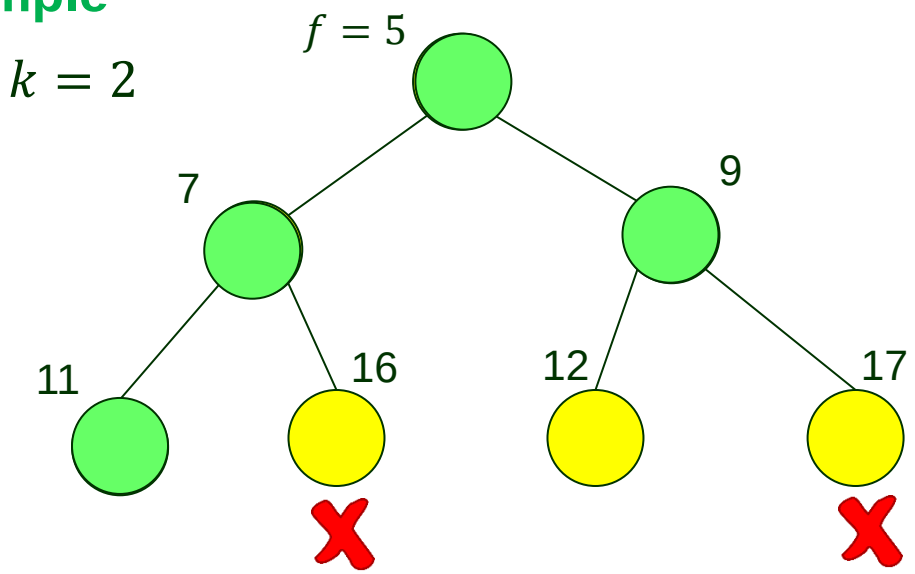


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

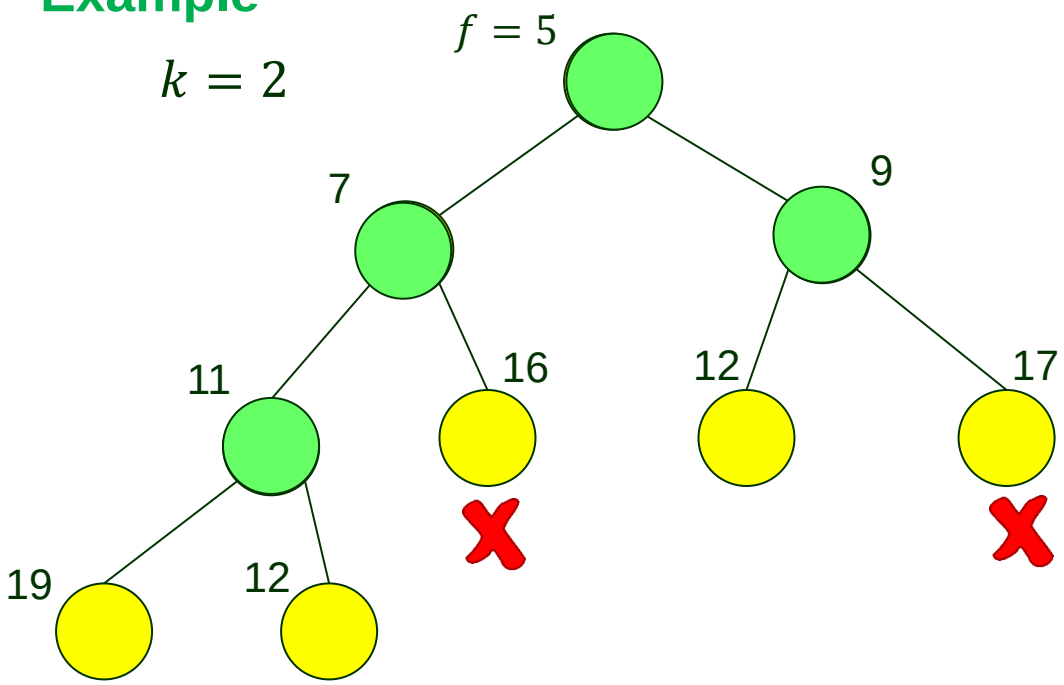


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

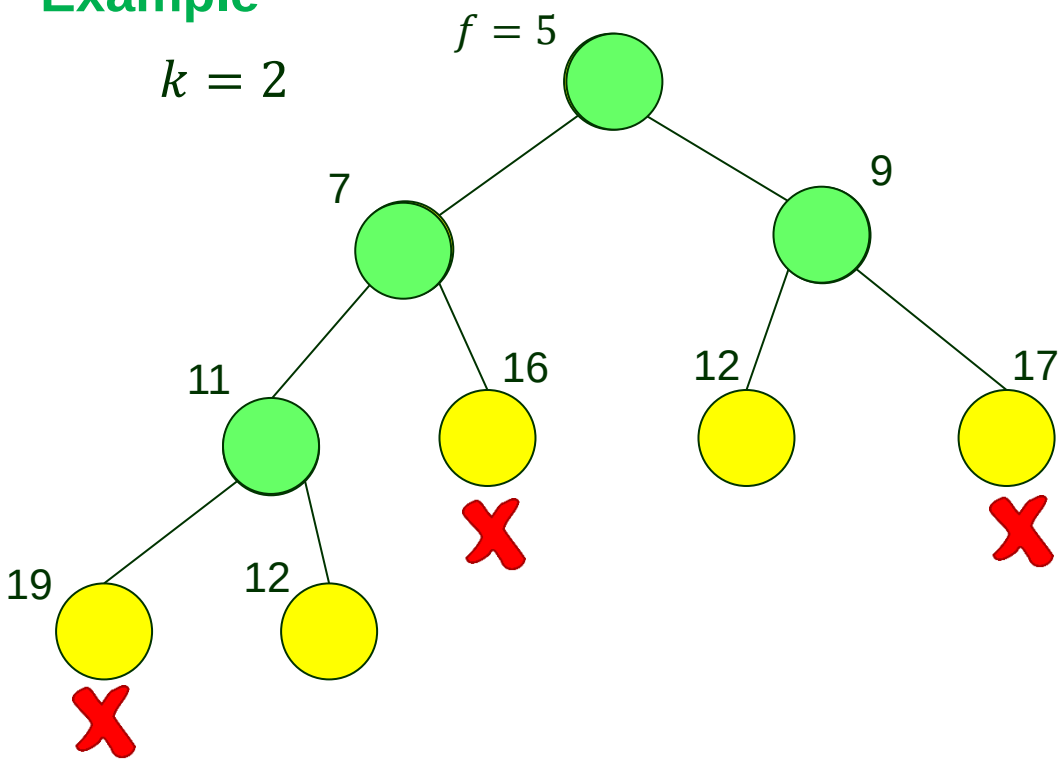


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

Example

$k = 2$

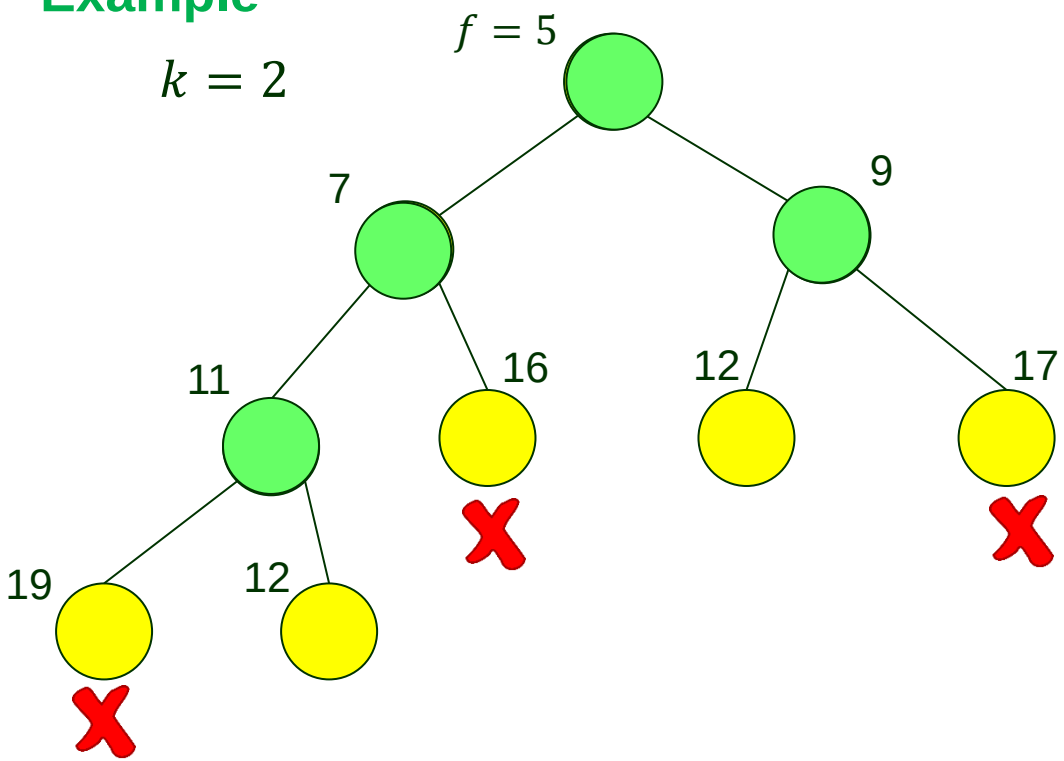


Memory-Bounded Search

Beam search keeps the k nodes with the best f scores.

- ♦ Less memory and faster execution.
- ♣ Incomplete and suboptimal.

Example



Iterative-Deepening A* Search (IDA*)

- The cutoff at each iteration is the f-cost rather than depth.
- Determine the (new) cutoff value for the current iteration as

$S = \{n \mid$ node n generated in the previous iteration and $f(n) >$ old $f_{\text{cutoff}}\}$

new $f_{cutoff} \leftarrow \min_{n \in S} f(n)$

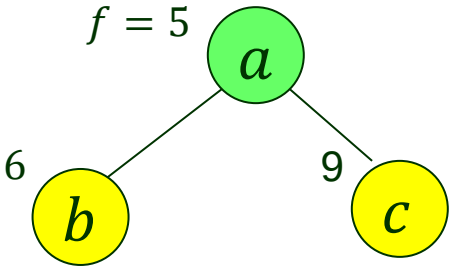
Iterative-Deepening A* Search (IDA*)

- The cutoff at each iteration is the f-cost rather than depth.
- Determine the (new) cutoff value for the current iteration as

$S = \{n \mid$ node n generated in the previous iteration and $f(n) >$ old $f_{\text{cutoff}}\}$

new $f_{cutoff} \leftarrow \min_{n \in S} f(n)$

Example



Iteration 1 ($f_{cutoff} = 5$)

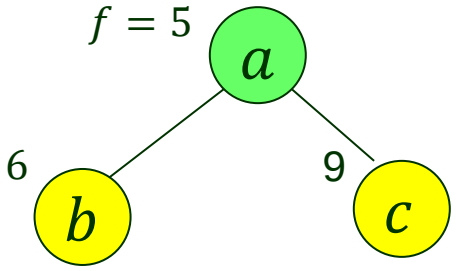
Iterative-Deepening A* Search (IDA*)

- The cutoff at each iteration is the f-cost rather than depth.
- Determine the (new) cutoff value for the current iteration as

$S = \{n \mid$ node n generated in the previous iteration and $f(n) >$ old $f_{\text{cutoff}}\}$

new $f_{cutoff} \leftarrow \min_{n \in S} f(n)$

Example



Iteration 1 ($f_{cutoff} = 5$) Iter. 2 ($f_{cutoff} = 6$)

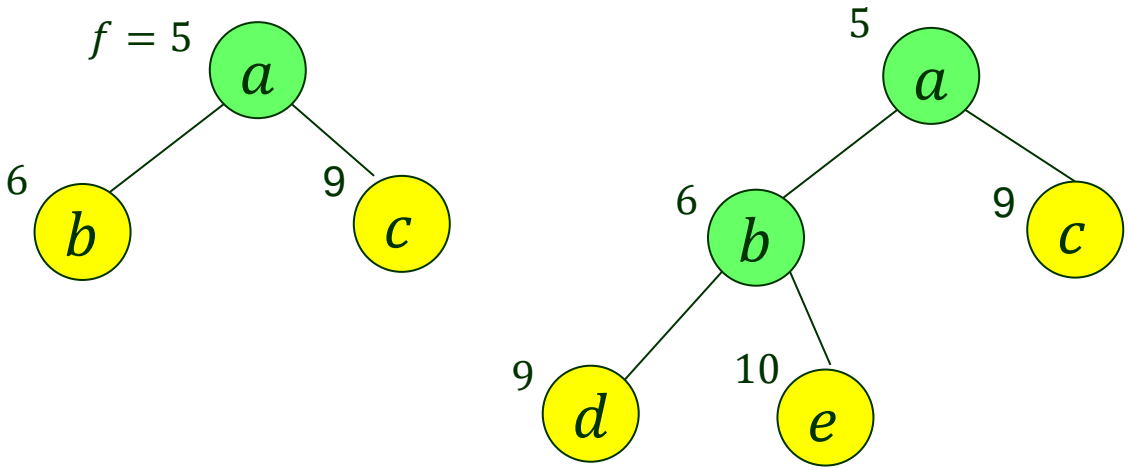
Iterative-Deepening A* Search (IDA*)

- The cutoff at each iteration is the f-cost rather than depth.
- Determine the (new) cutoff value for the current iteration as

$S = \{n \mid$ node n generated in the previous iteration and $f(n) >$ old $f_{\text{cutoff}}\}$

new $f_{cutoff} \leftarrow \min_{n \in S} f(n)$

Example



Iteration 1 ($f_{cutoff} = 5$) Iter. 2 ($f_{cutoff} = 6$)

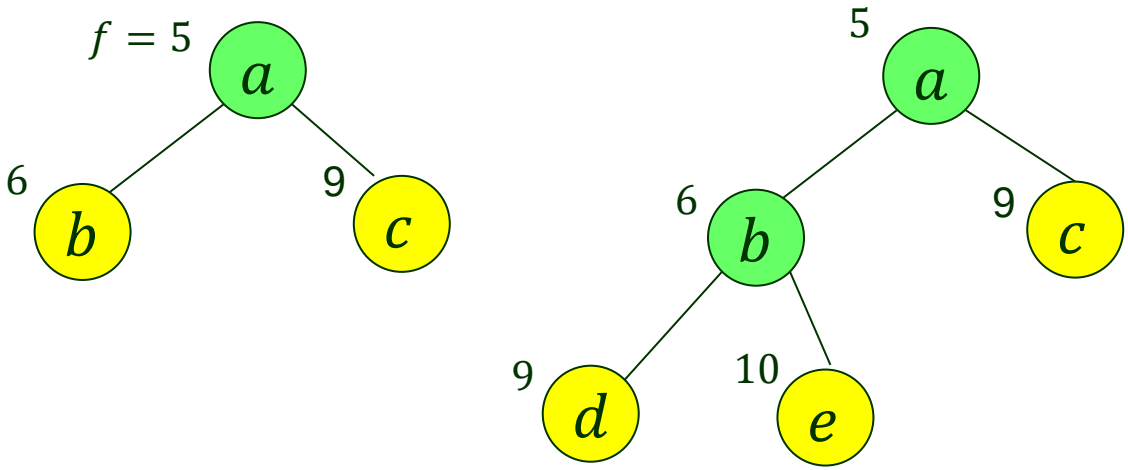
Iterative-Deepening A* Search (IDA*)

- The cutoff at each iteration is the f-cost rather than depth.
- Determine the (new) cutoff value for the current iteration as

$S = \{n \mid$ node n generated in the previous iteration and $f(n) >$ old $f_{\text{cutoff}}\}$

new $f_{cutoff} \leftarrow \min_{n \in S} f(n)$

Example



Iteration 1 ($f_{cutoff} = 5$) Iter. 2 ($f_{cutoff} = 6$) Iter. 3 ($f_{cutoff} = 9$)

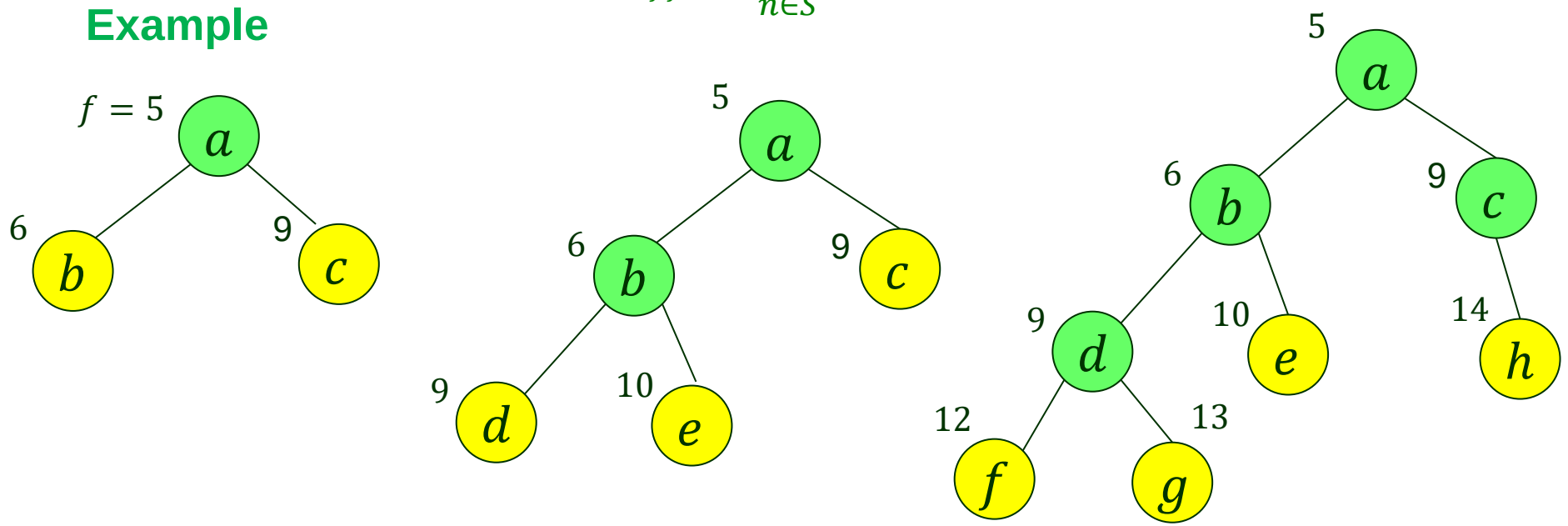
Iterative-Deepening A* Search (IDA*)

- The cutoff at each iteration is the f-cost rather than depth.
- Determine the (new) cutoff value for the current iteration as

$S = \{n \mid \text{node } n \text{ generated in the previous iteration and } f(n) > \text{old } f_{\text{cutoff}}\}$

new $f_{cutoff} \leftarrow \min_{n \in S} f(n)$

Example



Iteration 1 ($f_{cutoff} = 5$) Iter. 2 ($f_{cutoff} = 6$) Iter. 3 ($f_{cutoff} = 9$)

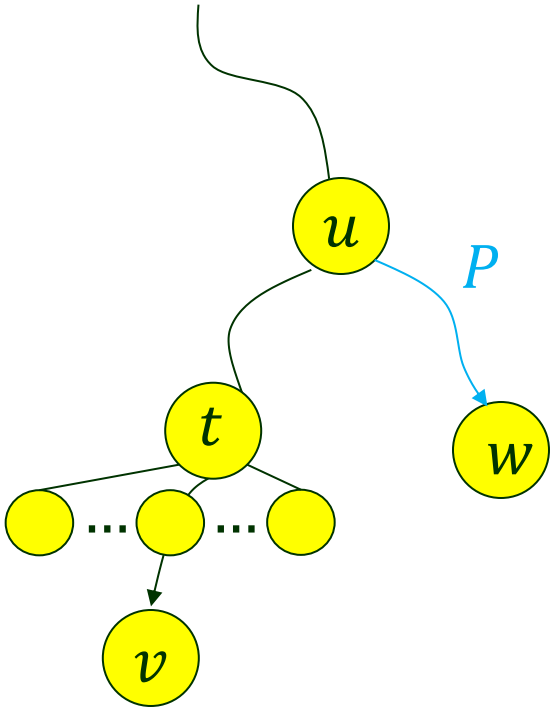
IDA* (cont'd)

- Steady progress towards the goal if f-cost of every path is an integer.

  $\leq C^*$ iterations

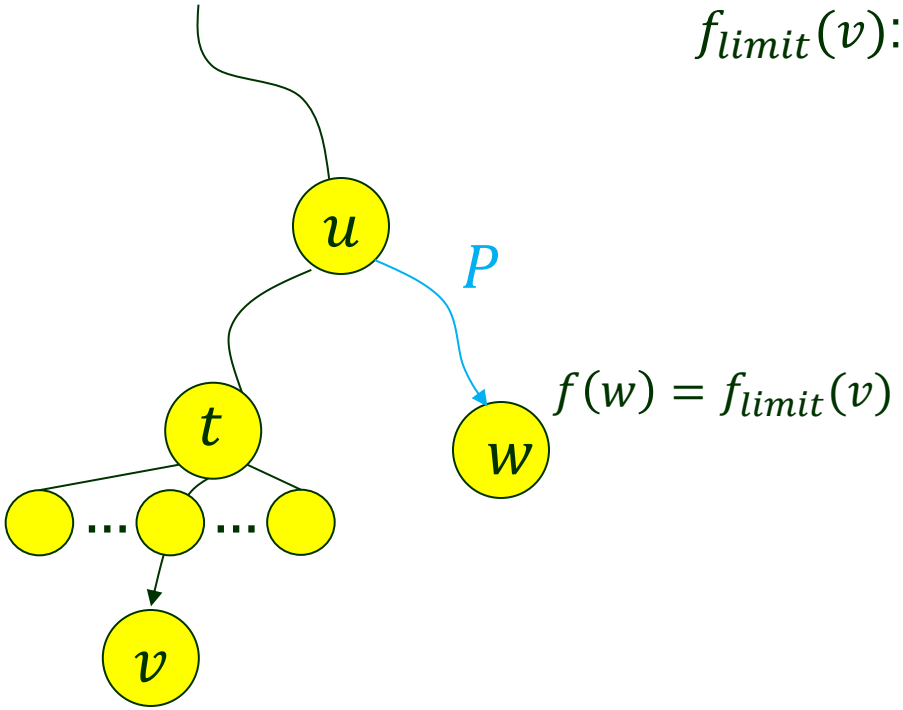
- In the worst case, #iterations = #states

Recursive Best-First Search (RBFS)



Idea: Remembers the f-value of the best leaf in the forgotten subtree in case it's worth a re-expansion at some later time.

Recursive Best-First Search (RBFS)

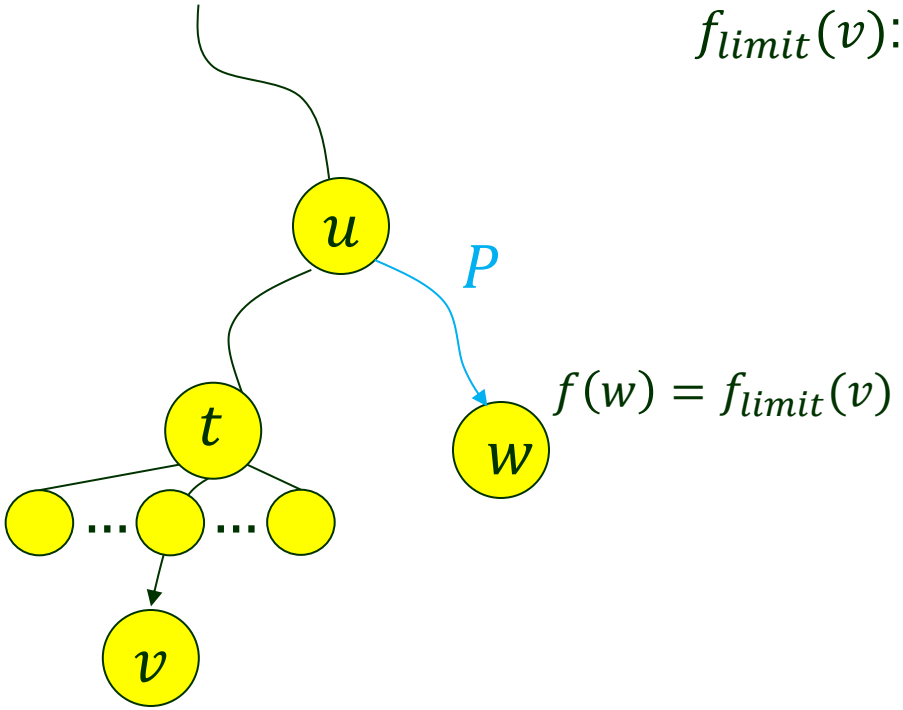


$f_{limit}(v)$: f-value of the *best alternative path* from any *ancestor* of the node v.

$f(w) = f_{limit}(v)$

Idea: Remembers the f-value of the best leaf in the forgotten subtree in case it's worth a re-expansion at some later time.

Recursive Best-First Search (RBFS)

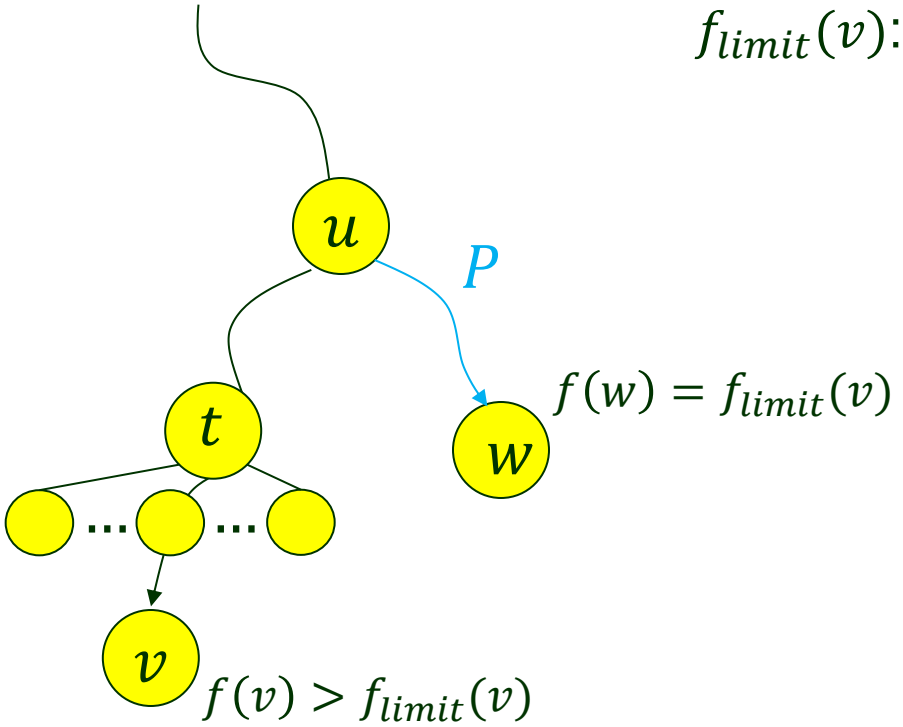


$f_{limit}(v)$: f-value of the *best alternative path* from any *ancestor* of the node v.

- Best-first search if, at the currently visited node v, it holds that $f(v) \leq f_{limit}(v)$.

Idea: Remembers the f-value of the best leaf in the forgotten subtree in case it's worth a re-expansion at some later time.

Recursive Best-First Search (RBFS)

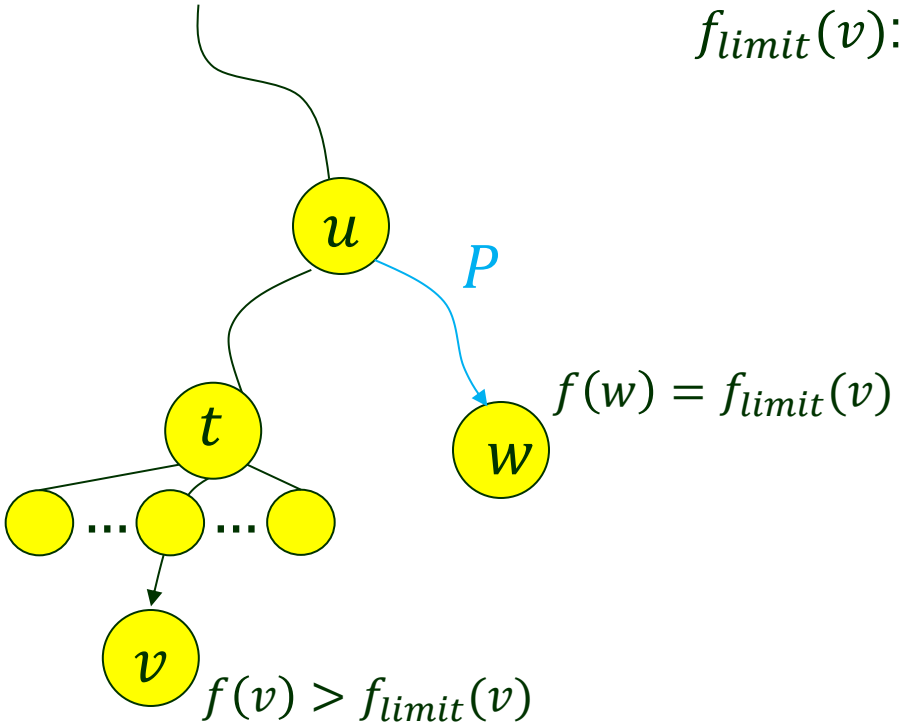


$f_{limit}(v)$: f-value of the *best alternative path* from any *ancestor* of the node v.

- Best-first search if, at the currently visited node v, it holds that $f(v) \leq f_{limit}(v)$.
- Otherwise (i.e., $f(v) > f_{limit}(v)$),

Idea: Remembers the f-value of the best leaf in the forgotten subtree in case it's worth a re-expansion at some later time.

Recursive Best-First Search (RBFS)

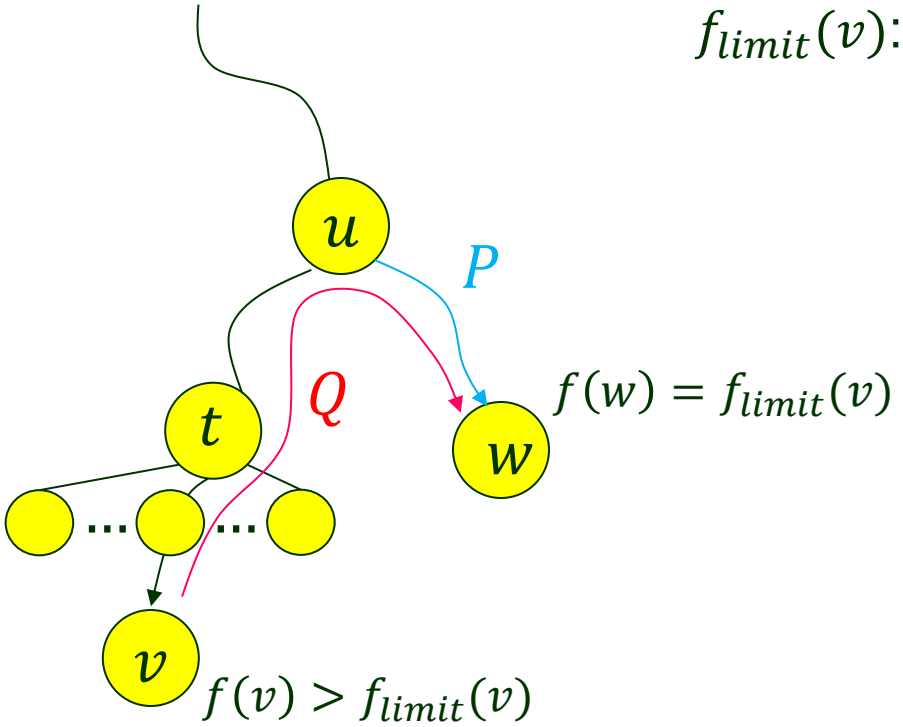


$f_{limit}(v)$: f-value of the *best alternative path* from any *ancestor* of the node v.

- Best-first search if, at the currently visited node v, it holds that $f(v) \leq f_{limit}(v)$.
- Otherwise (i.e., $f(v) > f_{limit}(v)$),
  - unwinds back to the alternative path P;

Idea: Remembers the f-value of the best leaf in the forgotten subtree in case it's worth a re-expansion at some later time.

Recursive Best-First Search (RBFS)

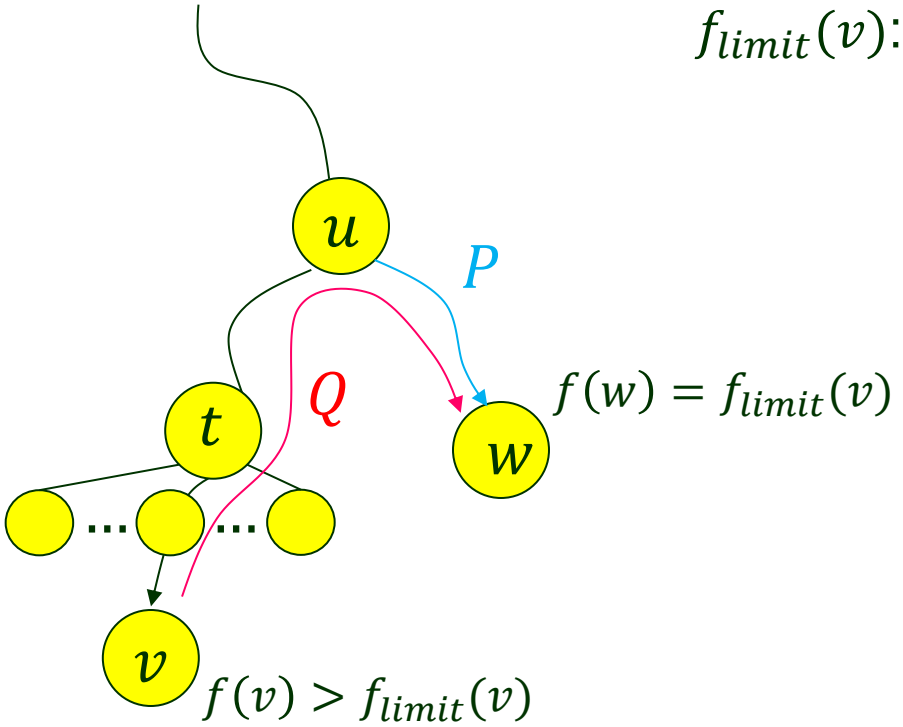


$f_{limit}(v)$: f-value of the *best alternative path* from any *ancestor* of the node v.

- Best-first search if, at the currently visited node v, it holds that $f(v) \leq f_{limit}(v)$.
- Otherwise (i.e., $f(v) > f_{limit}(v)$),
  - unwinds back to the alternative path P;
  - the visited nodes during the unwinding form a path Q.

Idea: Remembers the f-value of the best leaf in the forgotten subtree in case it's worth a re-expansion at some later time.

Recursive Best-First Search (RBFS)

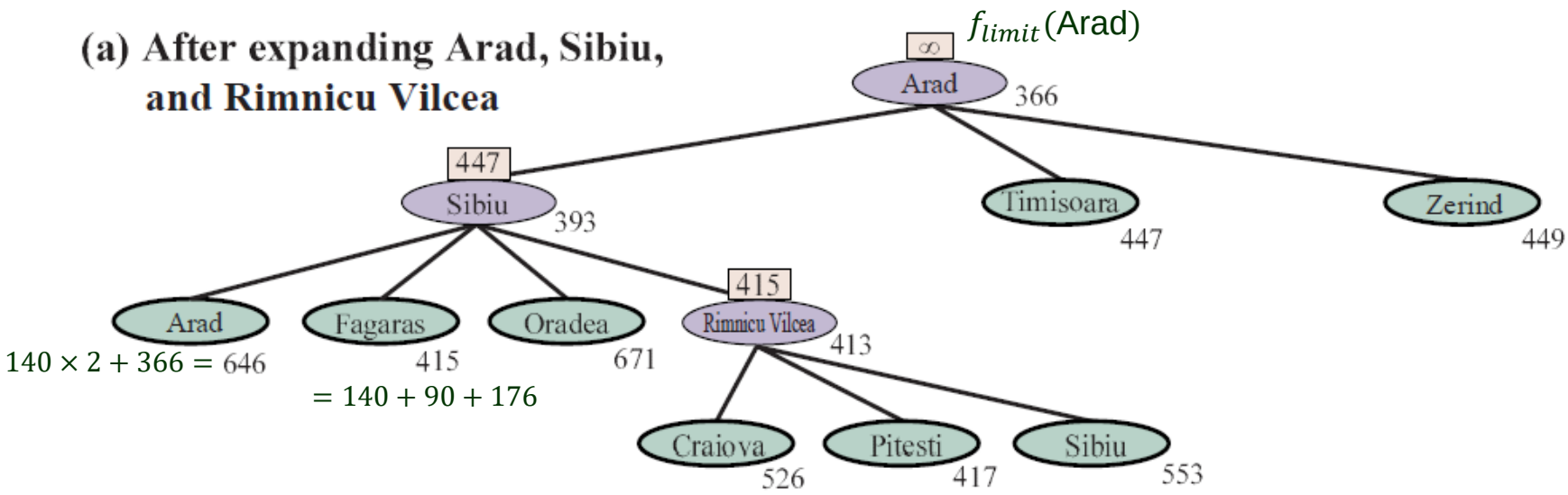


$f_{limit}(v)$: f-value of the *best alternative path* from any *ancestor* of the node v.

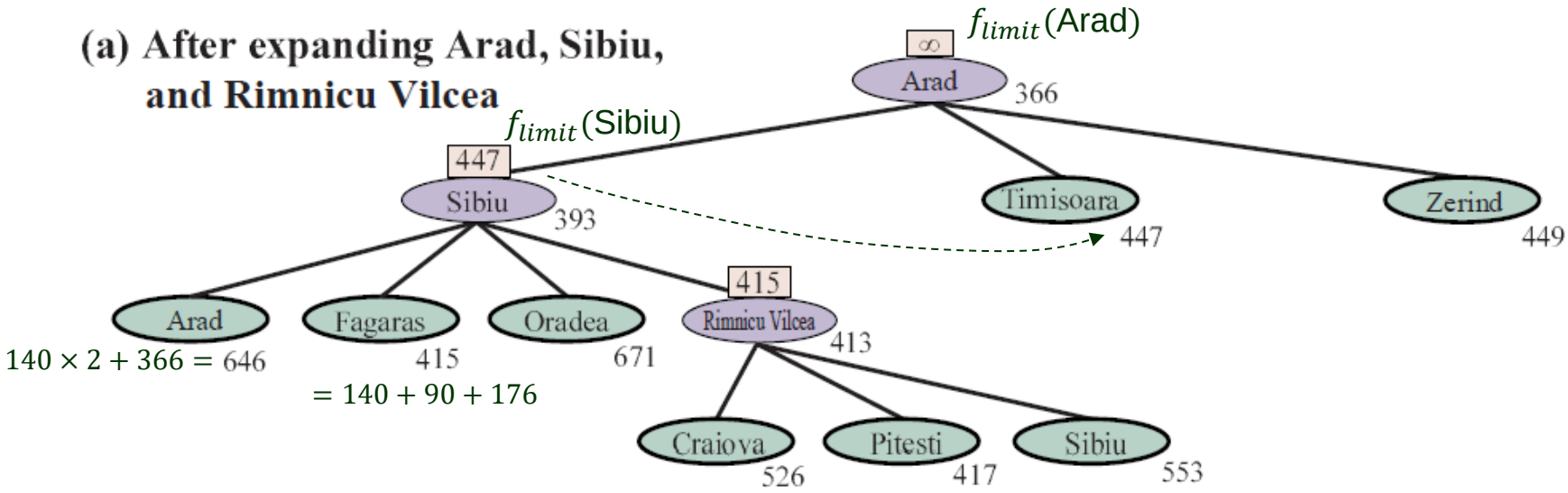
- Best-first search if, at the currently visited node v, it holds that $f(v) \leq f_{limit}(v)$.
- Otherwise (i.e., $f(v) > f_{limit}(v)$),
  - unwinds back to the alternative path P;
  - the visited nodes during the unwinding form a path Q.
  - set the f-value of every node t on Q to be the best f- value of its children.

Idea: Remembers the f-value of the best leaf in the forgotten subtree in case it's worth a re-expansion at some later time.

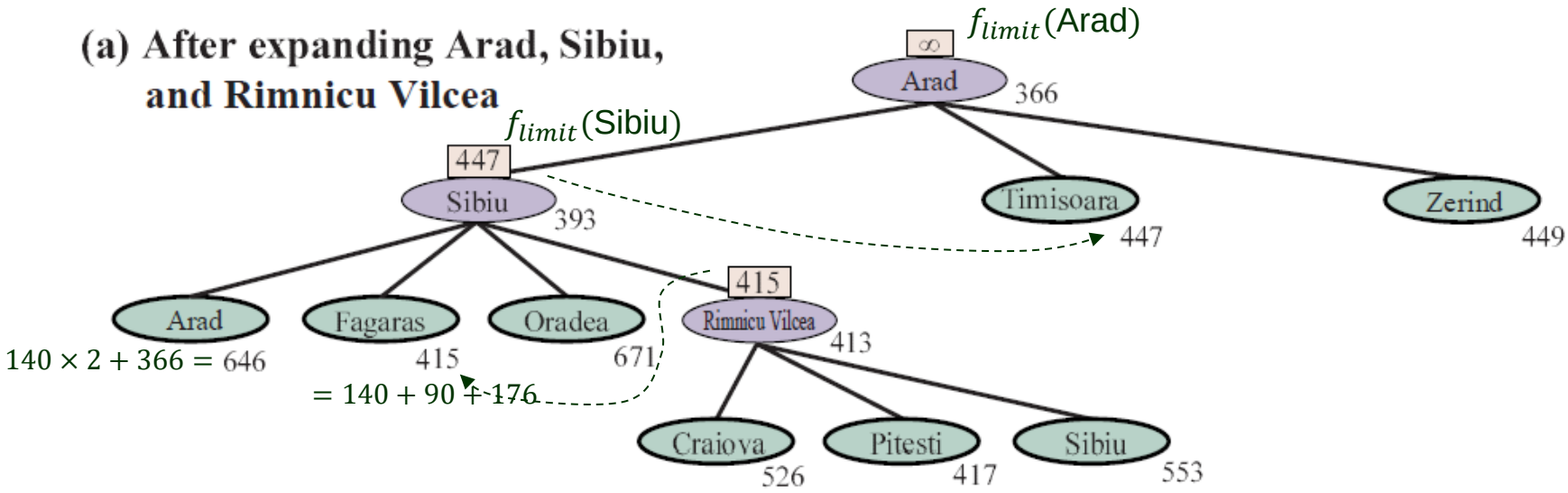
RBFS Example



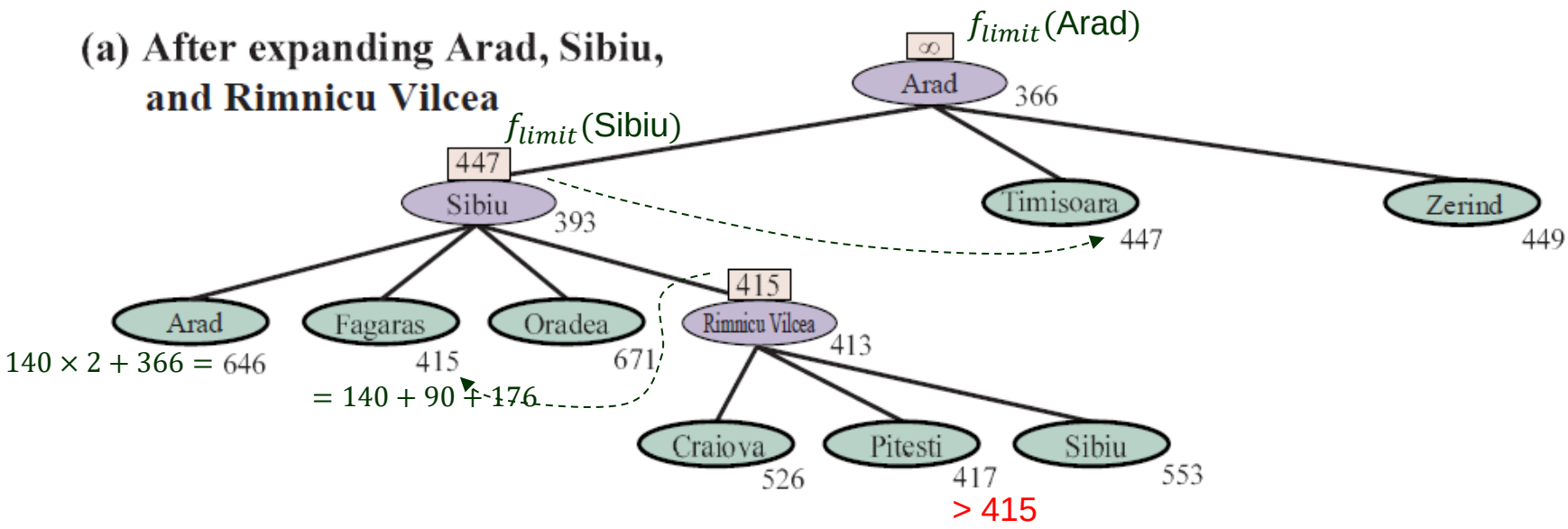
RBFS Example



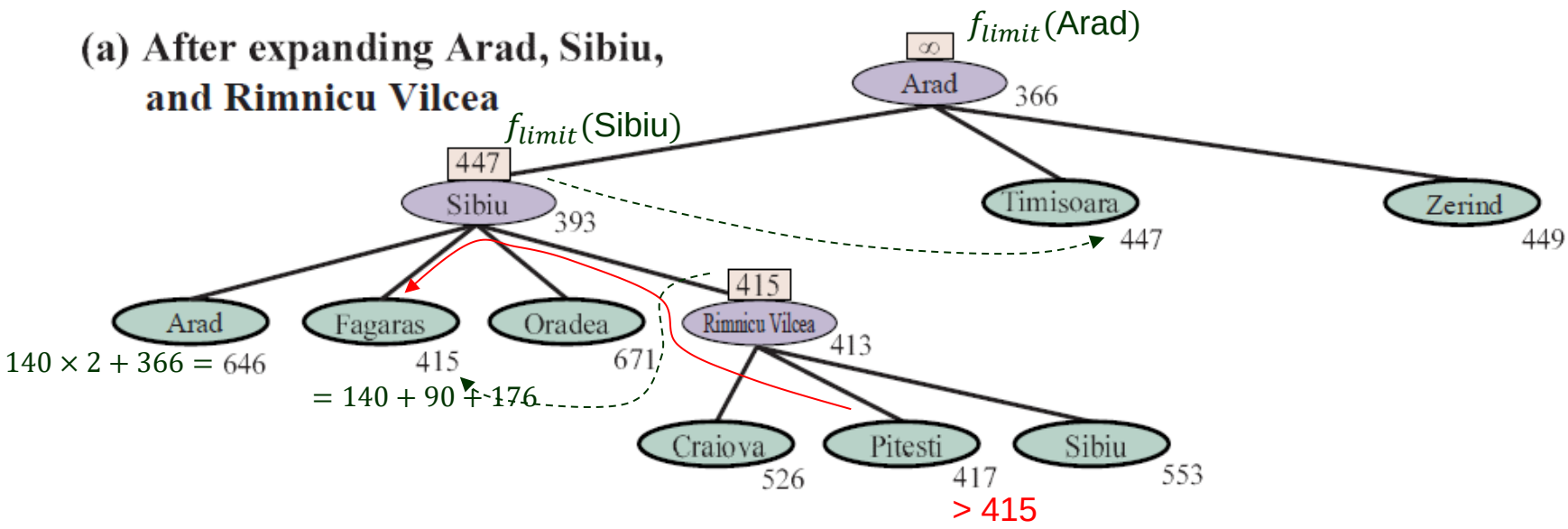
RBFS Example



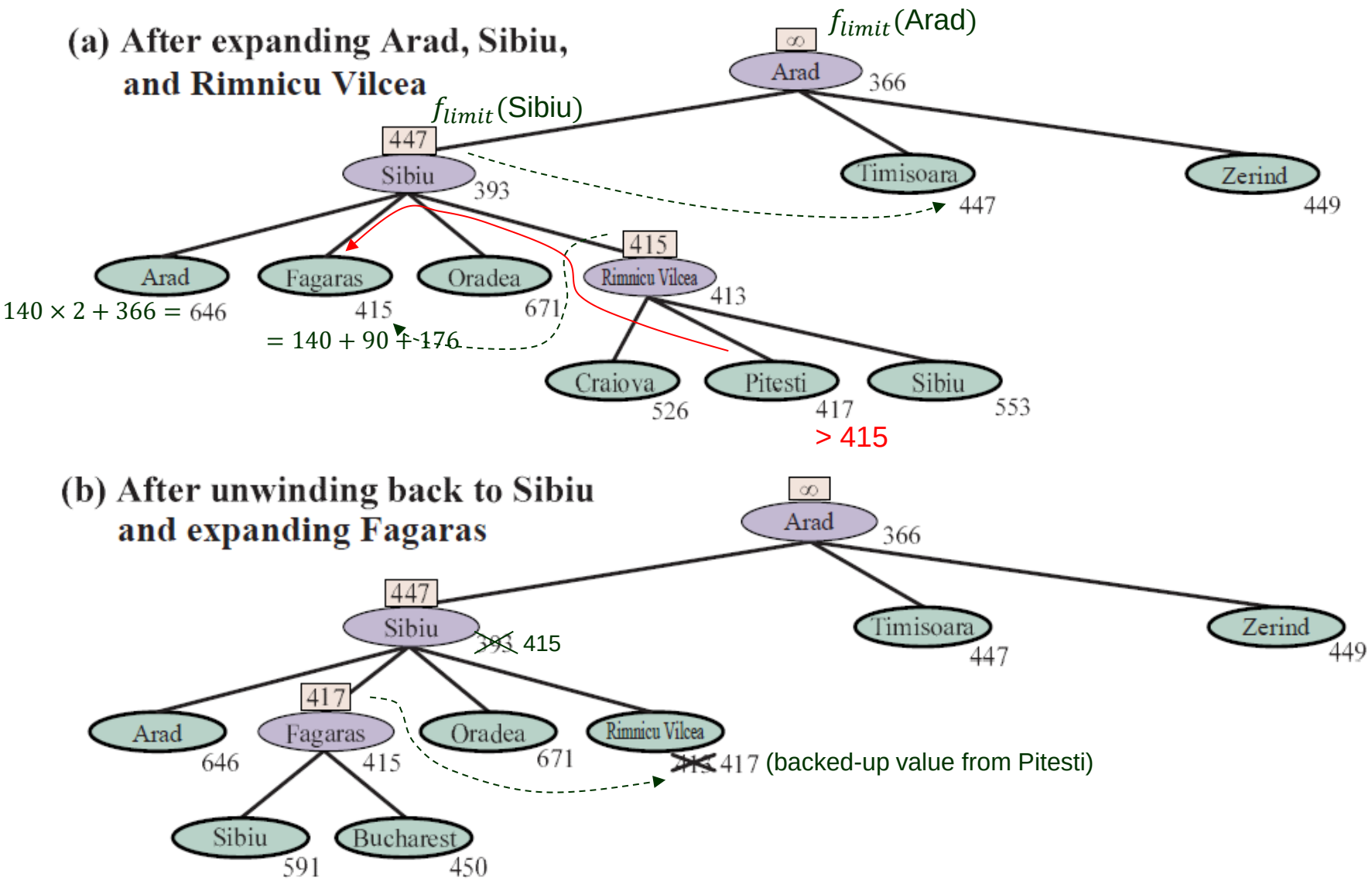
RBFS Example



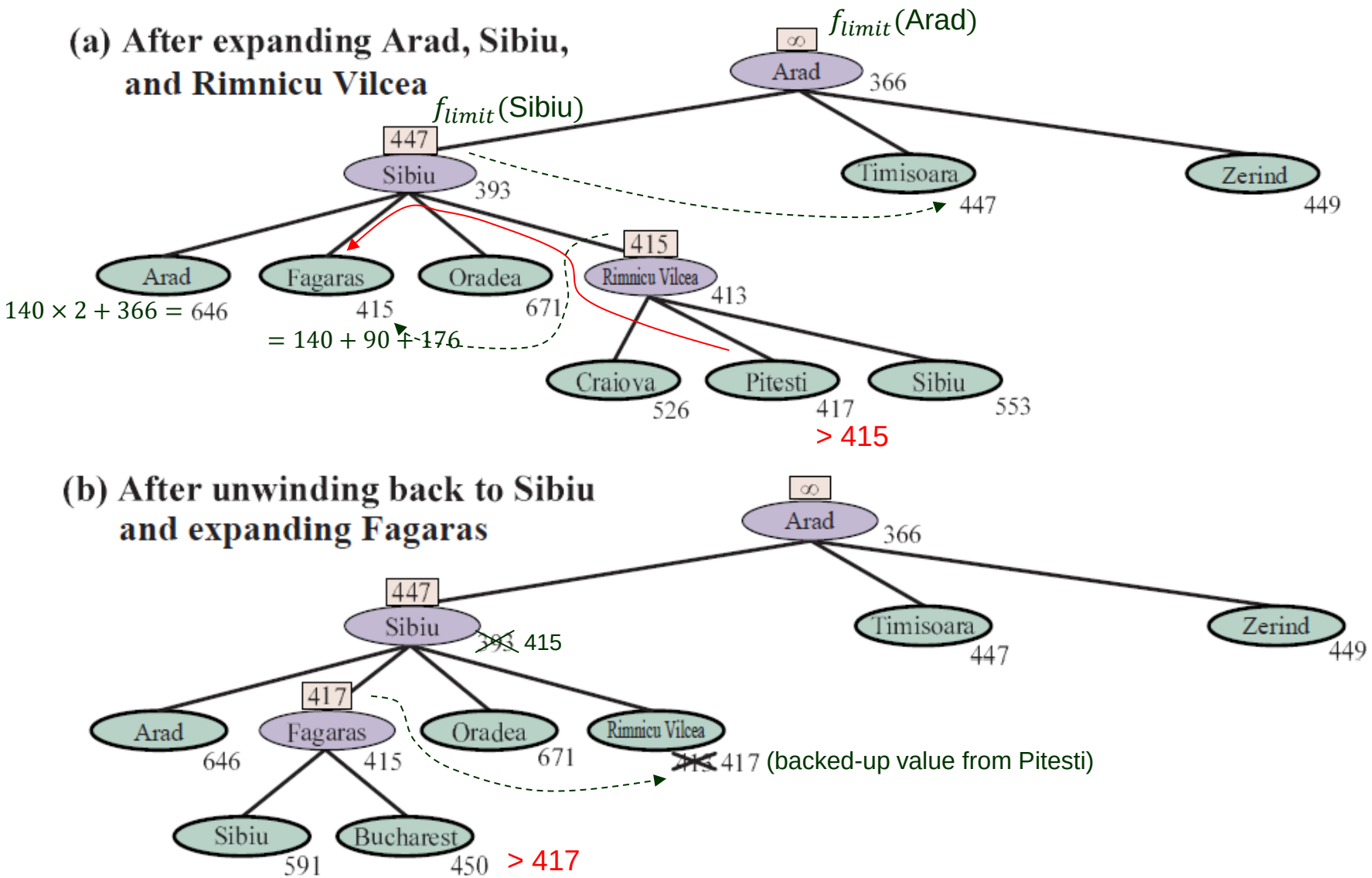
RBFS Example



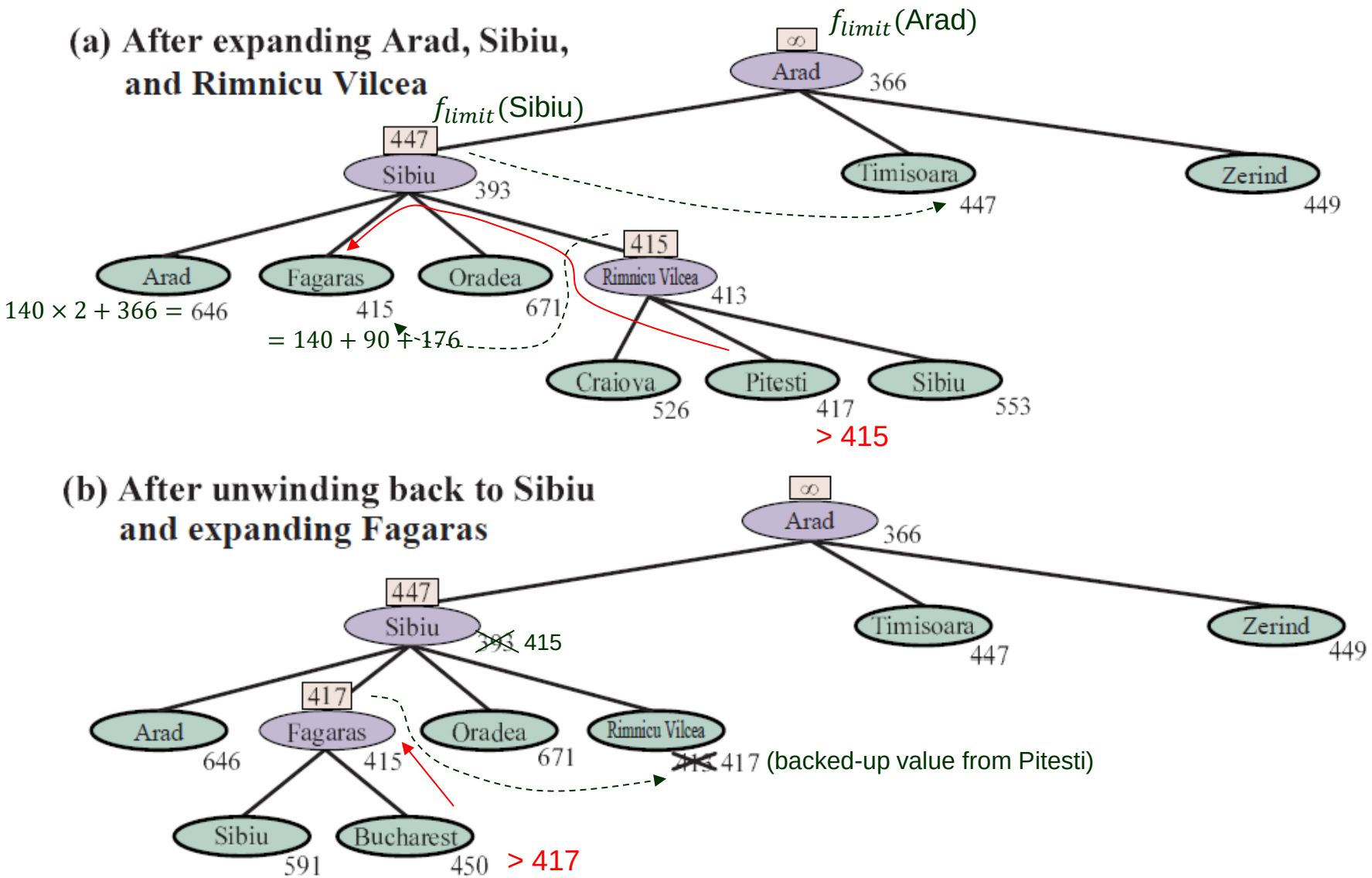
RBFS Example



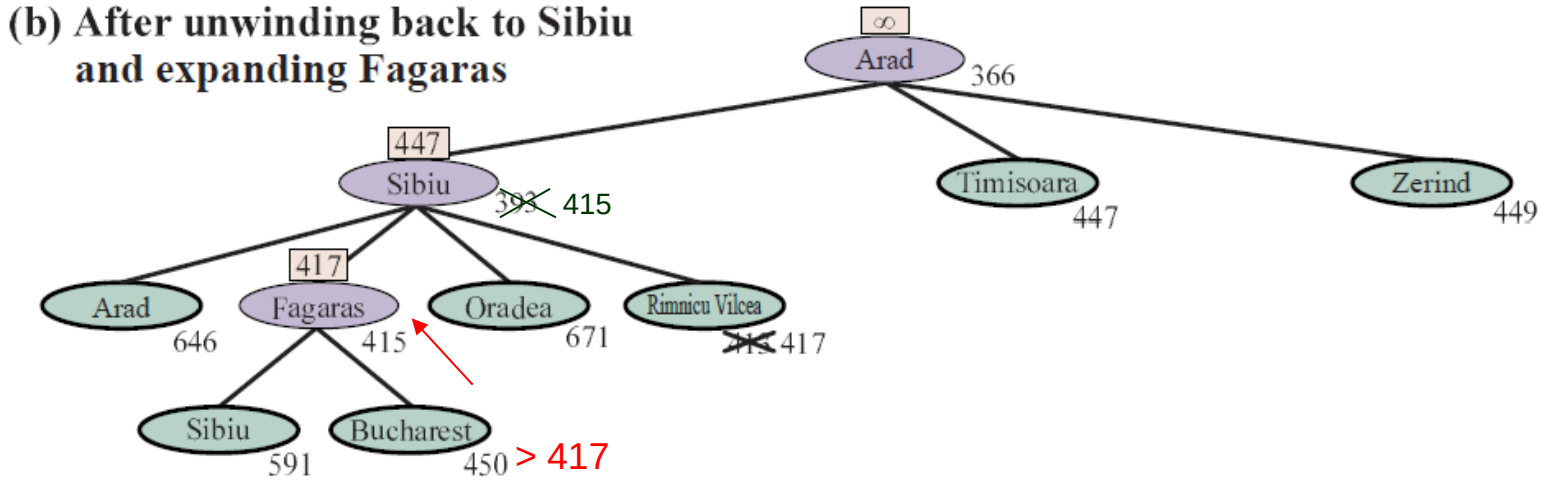
RBFS Example

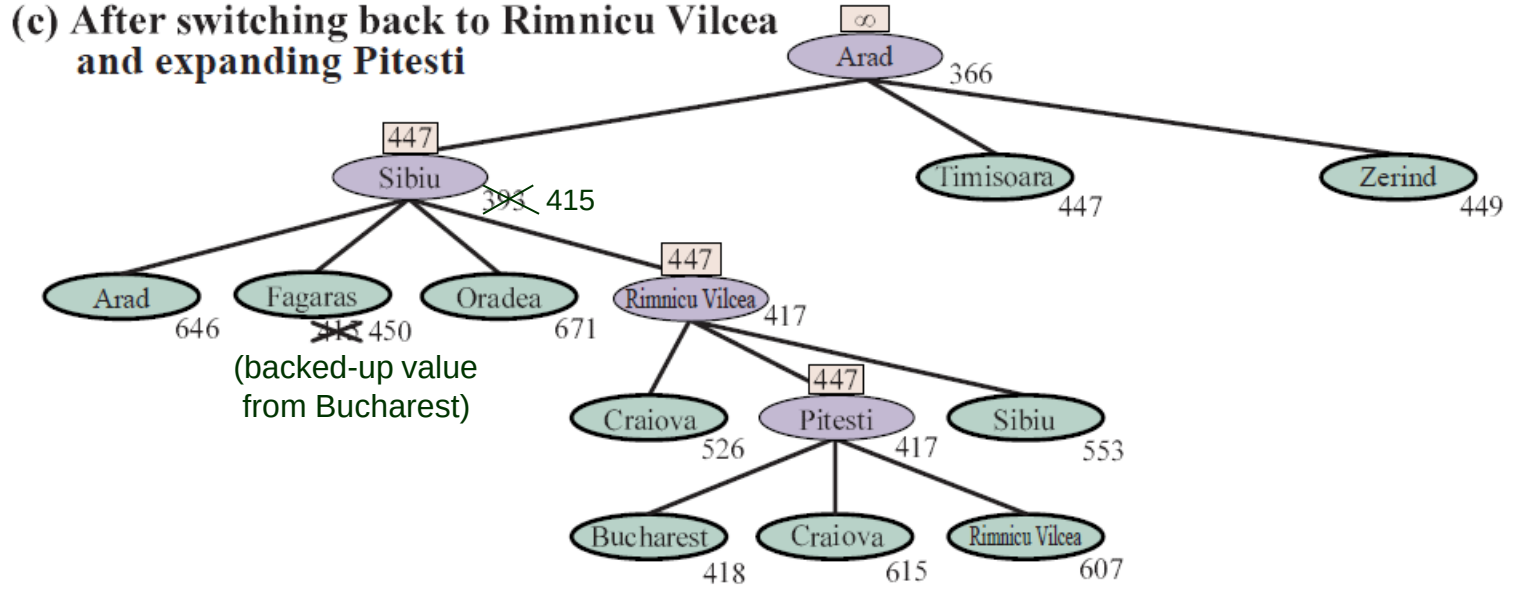


RBFS Example

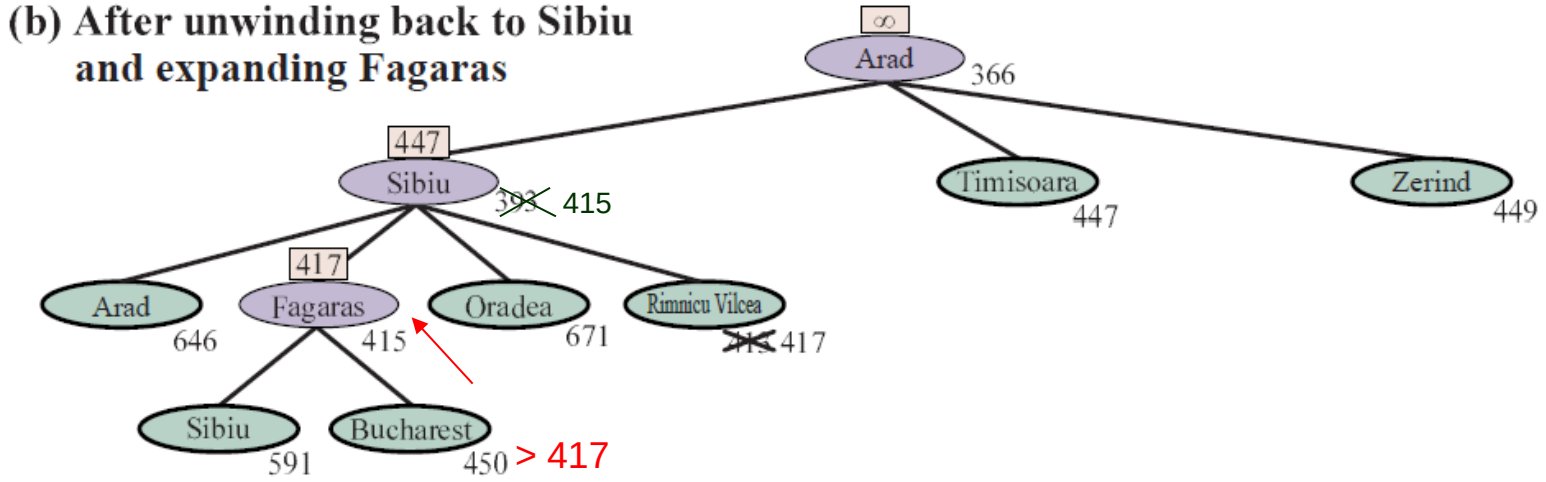


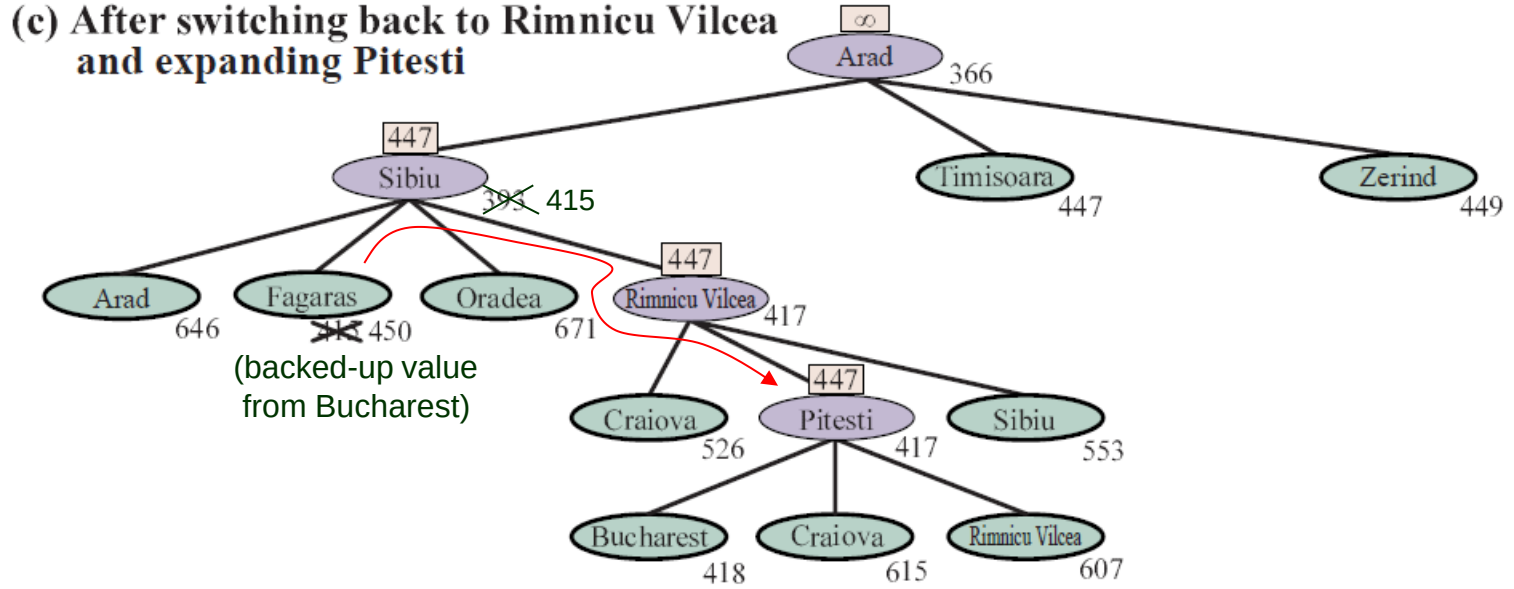
RBFS Example (cont'd)



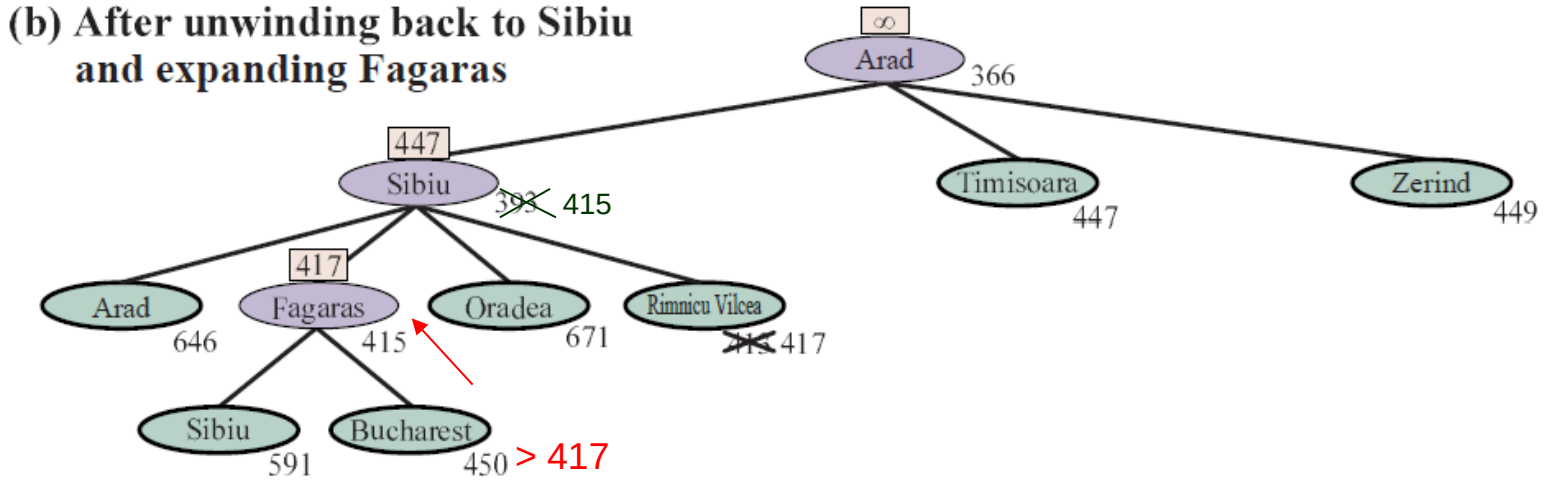


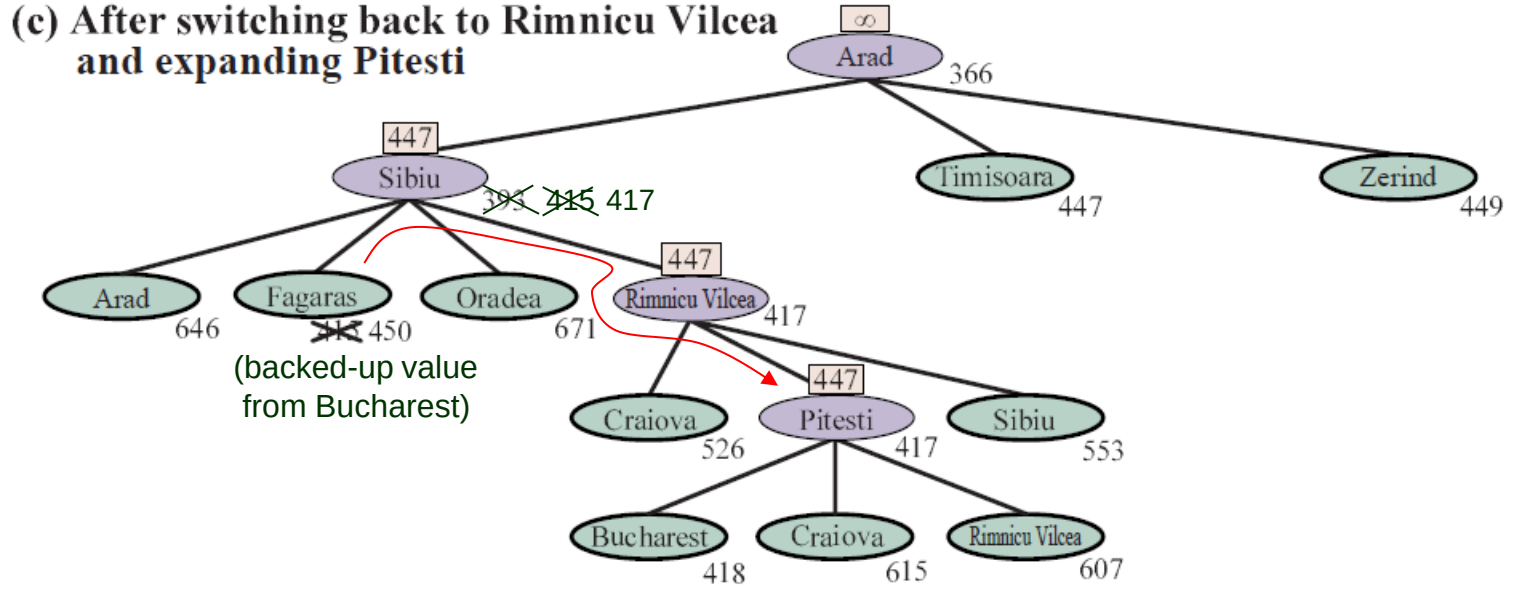
RBFS Example (cont'd)



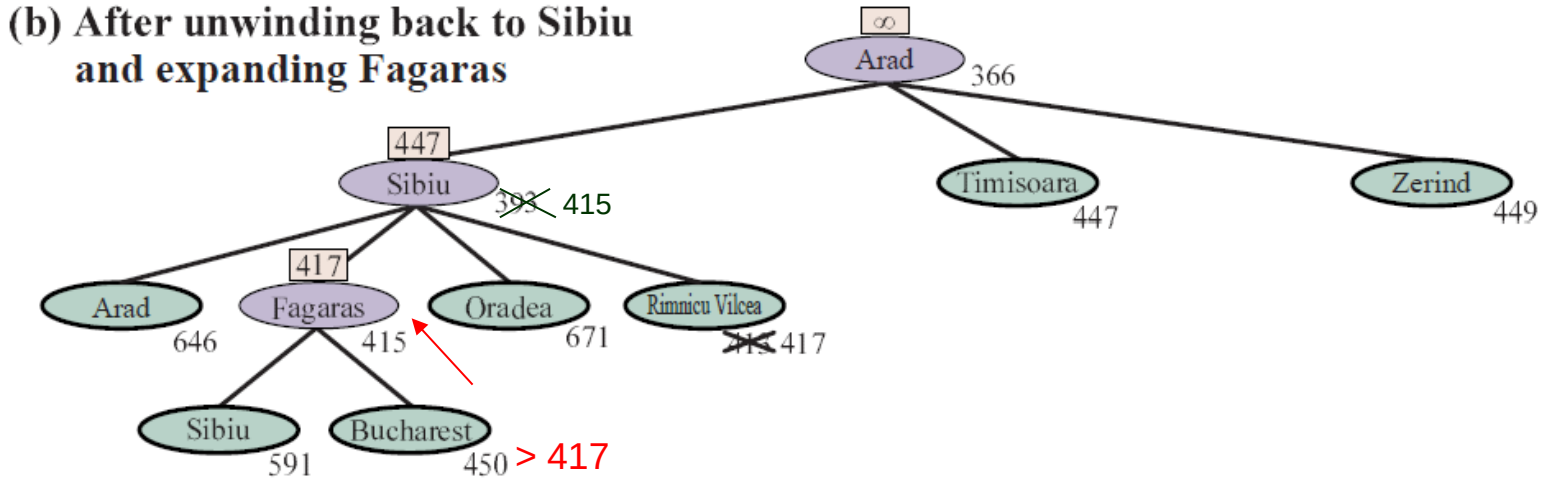


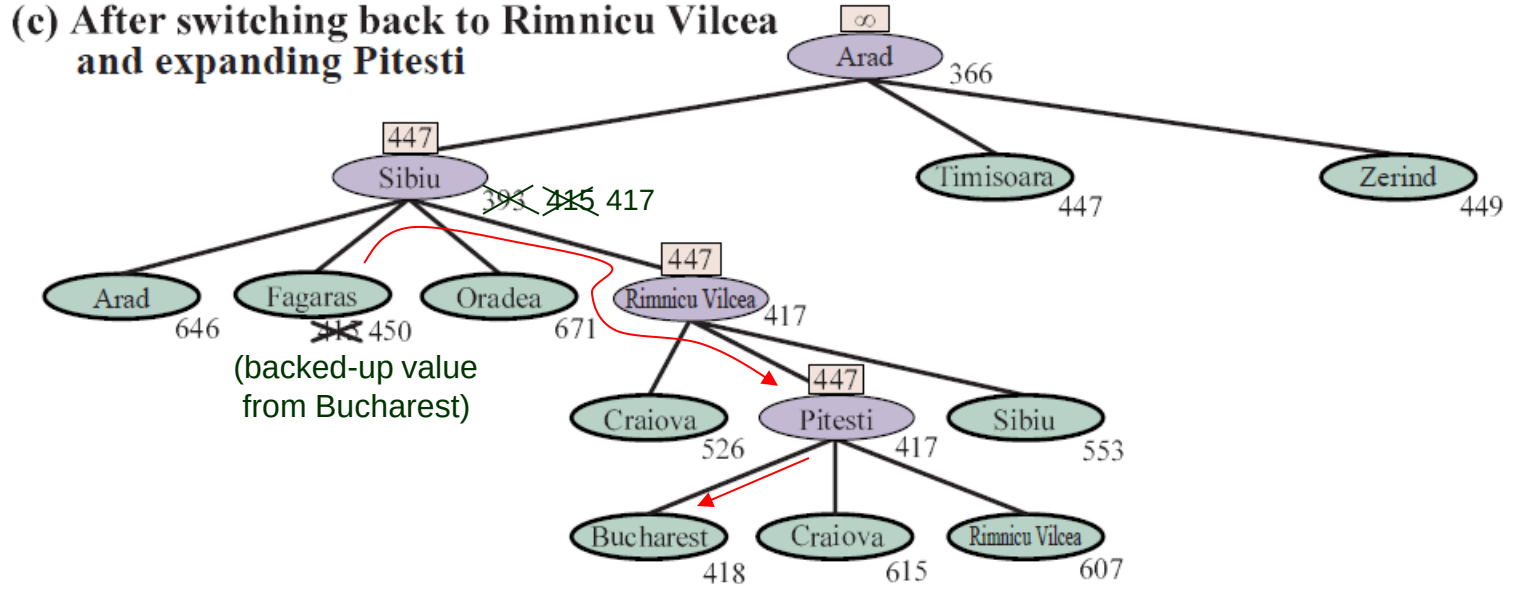
RBFS Example (cont'd)





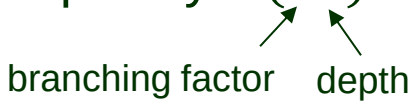
RBFS Example (cont'd)





RBFS Summary

- Optimal with admissible heuristic function $h(n)$.
- Space complexity $O(bd)$.



- Time complexity difficult to analyze, depending on
  - accuracy of $h(n)$
  - how often the best path changes
- Slightly more efficient than IDA*.
- Both IDA* and RBFS suffering from using too little memory and may explore the same state multiple times.