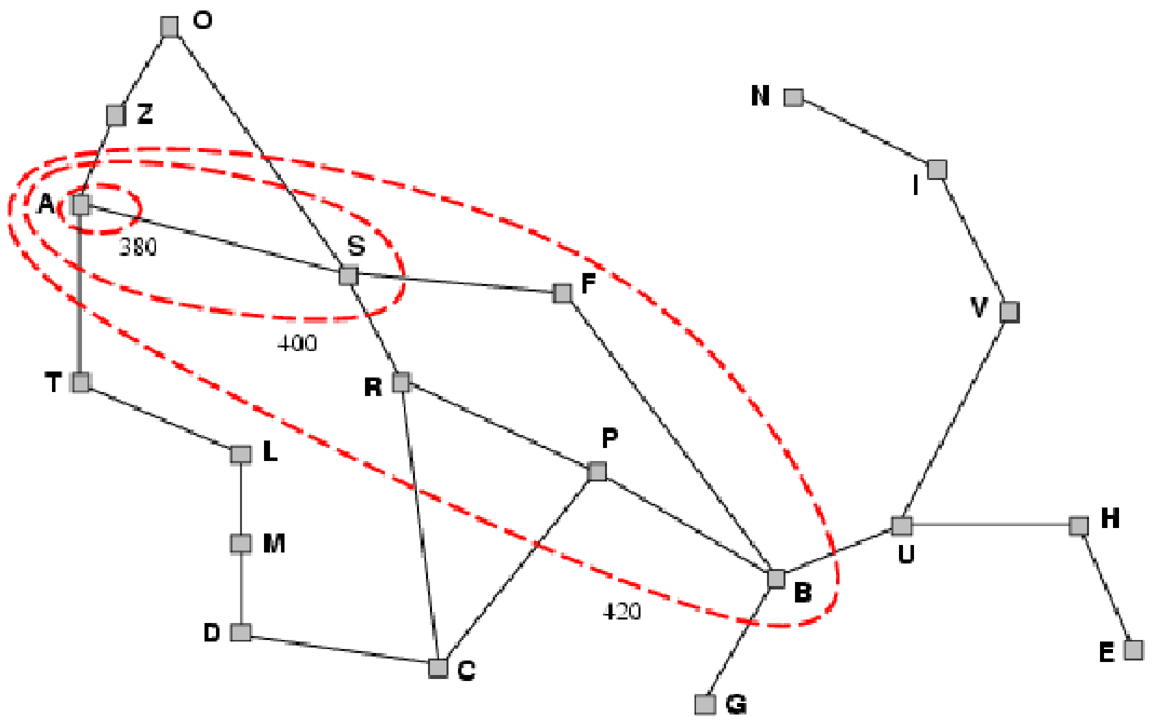
Contours of A* Search

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$

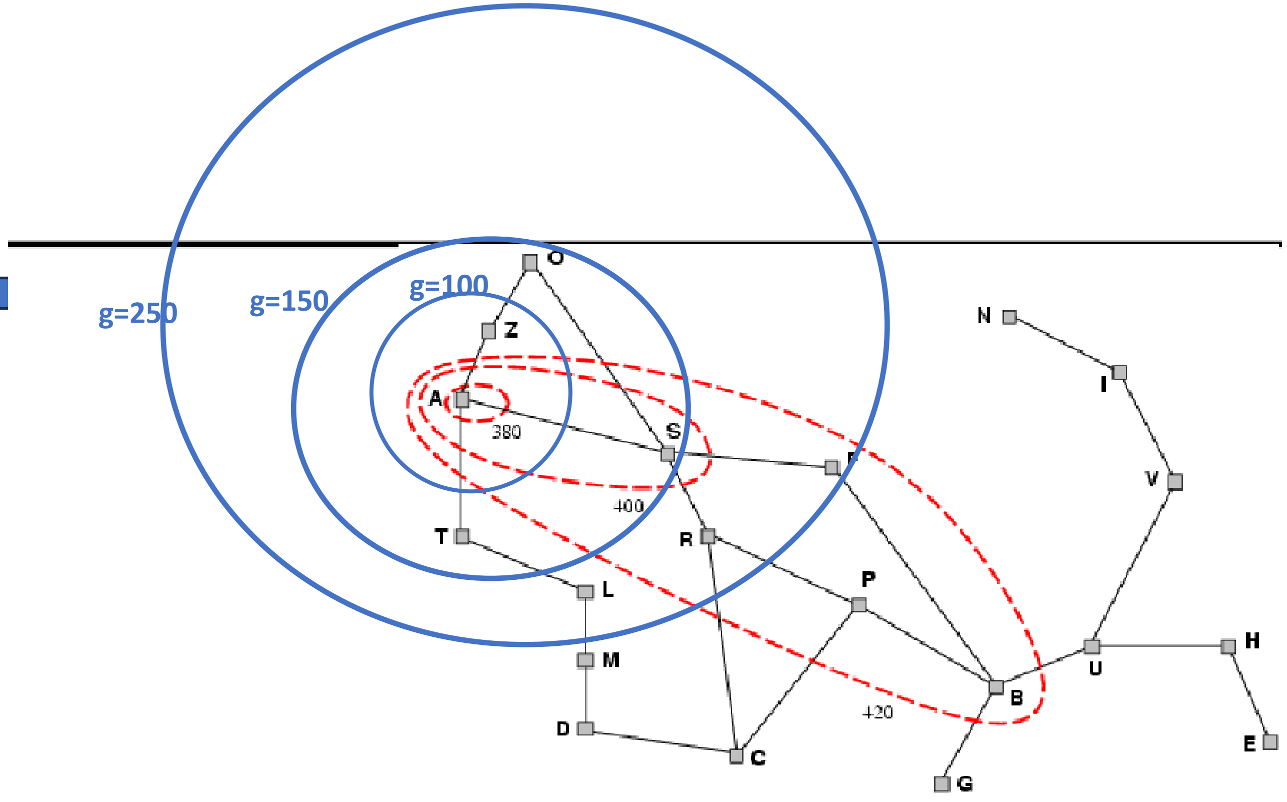


For g=100
Contour={A,Z(75)}

For g = 150
Contour += {O(146), S(140), T(118)}

For g=250
Contour += {F(249), R(220), L(229)}

UCS contours cover larger parts of space,
hence UCS expands more nodes, even those farther from the goal



- With uniform-cost (h(n) = 0, contours will be circular
- With good heuristics, contours will be focused around optimal path
- A* will expand all nodes with cost f(n) < C*

Properties of A*

- Complete?
  - Yes (unless there are infinitely many nodes with $f \leq f(G)$)
- Optimal?
  - Yes
  - Also optimally efficient:
    - No other optimal algorithm will expand fewer nodes, for a given heuristic
- Time?
  - Exponential in worst case
- Space?
  - Exponential in worst case

It is efficient because it prunes unnecessary nodes (i.e., discards certain possibilities without even examining them).

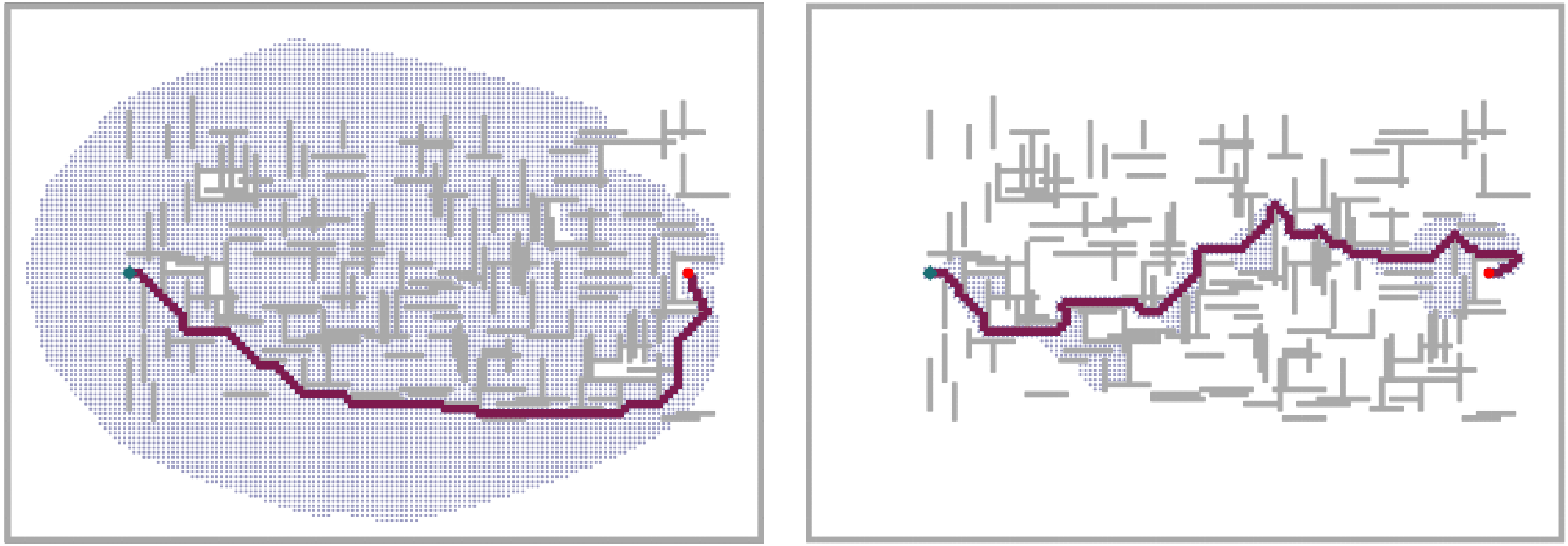
However, it expands too many nodes. Relax constraints. **Seek satisficing (suboptimal, but "good enough") solutions**.

Inadmissible heuristics can be more accurate, thereby reducing the number of nodes expanded, even if they can return non-optimal solutions.

Example: detour index, a multiplier applied to the straight-line distance to account for the typical curvature of roads

Weighted A* : it uses $f(n) = g(n) + W \times h(n)$, $W>1$

It finds solutions with cost c, $C^* <= c <= W \times C^*$
but in practice, we usually get results much closer to C^* than $W \times C^*$.



(a) (b)

Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

IDA* Iterative Deepening A*

- (IDA*) eliminates the memory constraints of A* search algorithm without sacrificing solution optimality.
- Each iteration of the algorithm is a depth-first search that keeps track of the cost, $f(n) = g(n) + h(n)$, of each node generated.
- when a node is generated whose cost exceeds a threshold for that iteration, its path is cut off, and the search backtracks.
- The cost threshold is initialized to the heuristic estimate of the initial state
- In each iteration cost threshold is increased to the total cost of the lowest-cost node that was pruned during the previous iteration.
- The algorithm terminates when a goal state is reached whose total cost does not exceed the current threshold.

Each iteration performs an exhaustive search within limit f, finds a node just beyond the boundary, and uses its f as the new limit.

IDA* Iterative Deepening A*

Advantages:

- IDA*: if the heuristic function is admissible, IDA* finds an optimal solution
- IDA* memory requirement is linear with respect to the maximum search depth

DISADVANTAGES

3. IDA* can visit the same state multiple times during the same iteration
4. When IDA* restarts, it discards all information except the next threshold

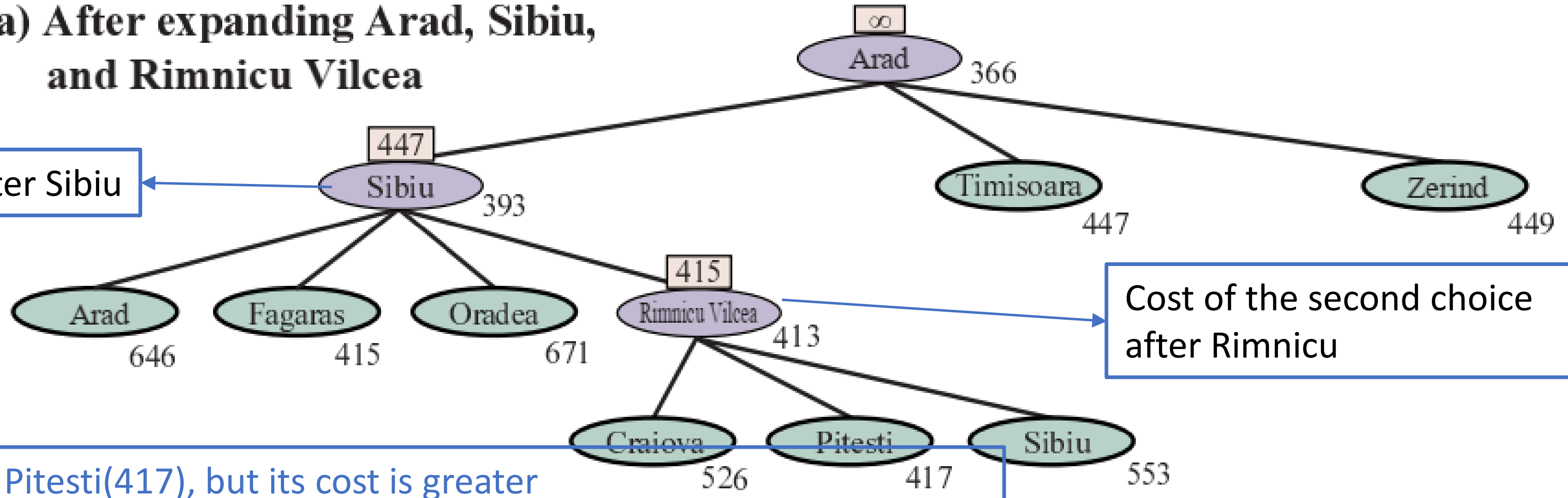
RBFS: best-first search, but using only linear space.

→ It doesn't maintain open all the branches (like A* does). It used a variable to remember the best second choice.

→ **When the first choice fails (it becomes worse than the second), the algorithm knows where to go**

→ It deletes the failing branch, remembering the best value for f in that branch

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea

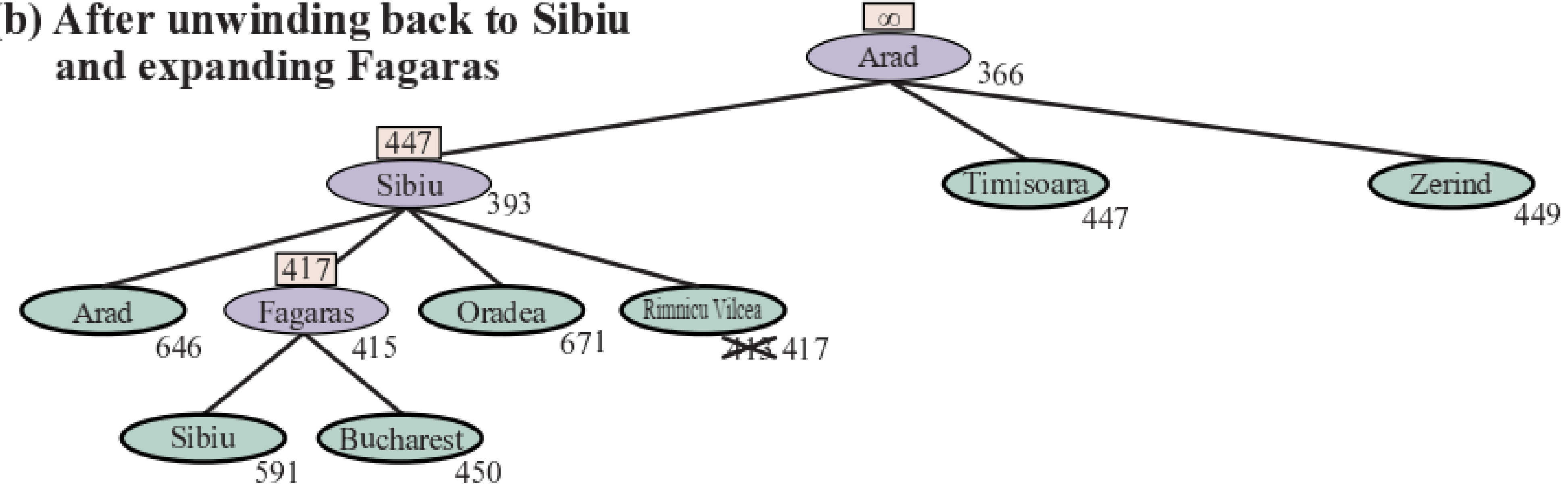


Cost of the second choice after Sibiu

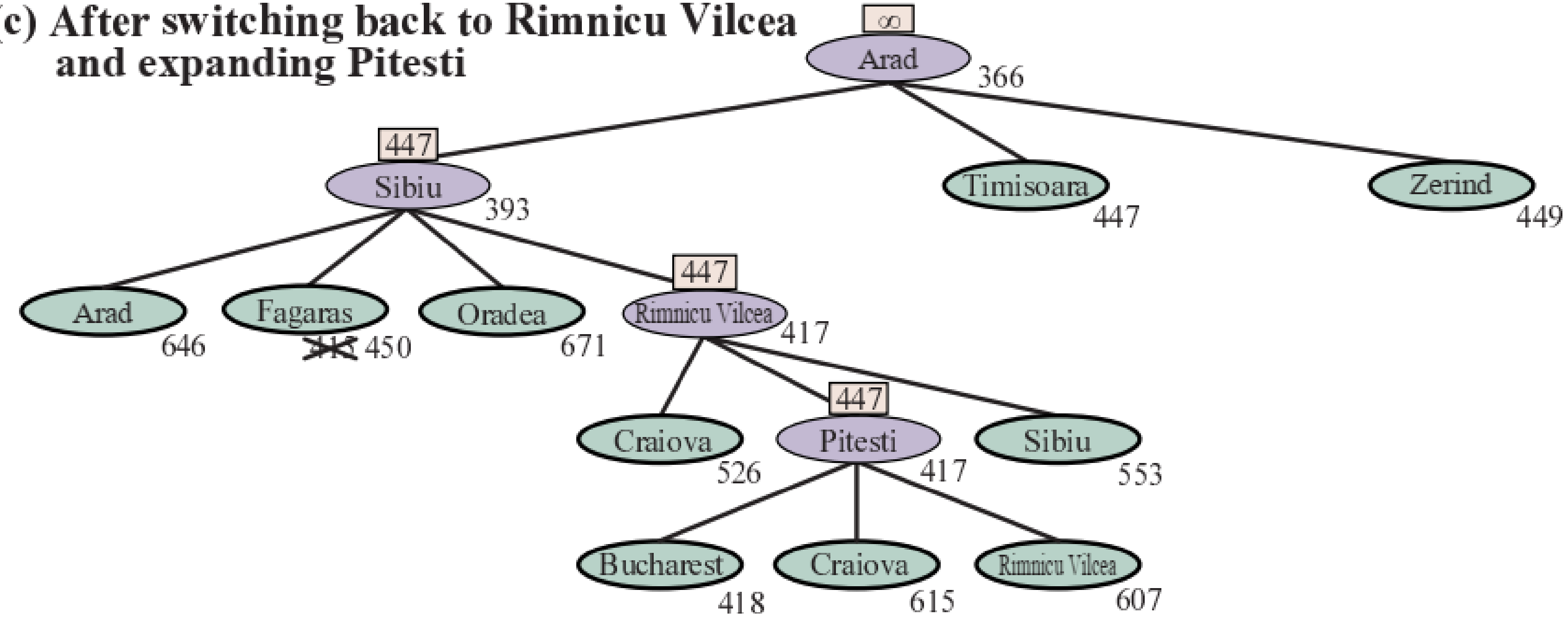
Cost of the second choice after Rimnicu

After Rimnicu we should expand Pitesti(417), but its cost is greater than the second best for Rimnicu (415). The branch from Rimnicu is deleted, 417 is recorded and the search continues from Fagaras (2nd choice)

(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti



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

function RBFS(problem, node, f_limit) returns a solution or failure, and a new f-cost limit
  if problem.IS-GOAL(node.STATE) then return node
  successors ← LIST(EXPAND(node))
  if successors is empty then return failure, ∞
  for each s in  successors do        // update f with value from previous search
      s.f ← max(s.PATH-COST + h(s), node.f))
  while true do
      best ← the node in successors with lowest f-value
      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, best.f
```

Figure 3.22 The algorithm for recursive best-first search.

RBFS properties

- Like A*, optimal if *h(n)* is admissible

- Time complexity difficult to characterize
  - Depends on accuracy if h(n) and how often best path changes.
  - Can end up "switching" back and fort

- Space complexity is *O(bd)*
  - Other extreme to A* - uses ***too little*** memory.

TOO LITTLE MEMORY → REGENERATION OF MANY NODES

Even if memory is available it cannot use it

⇒ MEMORY BOUNDED A* = MA*

SMA* = simplified MA*

SMA*

- like A* expands the best child
  BUT it has a memory limit
- when memory is full it has to drop a node
- it drops the worst node (highest f-value)
- like RBFS backs up f-value of the forgotten node to its parent

✓ complete if there is enough memory to contain the solution

✓ optimal if an optimal solution is reachable