

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

Figure 3.22 Values of h_{SLD} —straight-line distances to Bucharest.

expanding a node that is not on the solution path; hence, its search cost is minimal. It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called “greedy”—at each step it tries to get as close to the goal as it can.

Greedy best-first tree search is also incomplete even in a finite state space, much like depth-first search. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first because it is closest to Fagaras, but it is a dead end. The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras. The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop. (The graph search version *is* complete in finite spaces, but not in infinite ones.) The worst-case time and space complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

3.5.2 A* search: Minimizing the total estimated solution cost

A* SEARCH

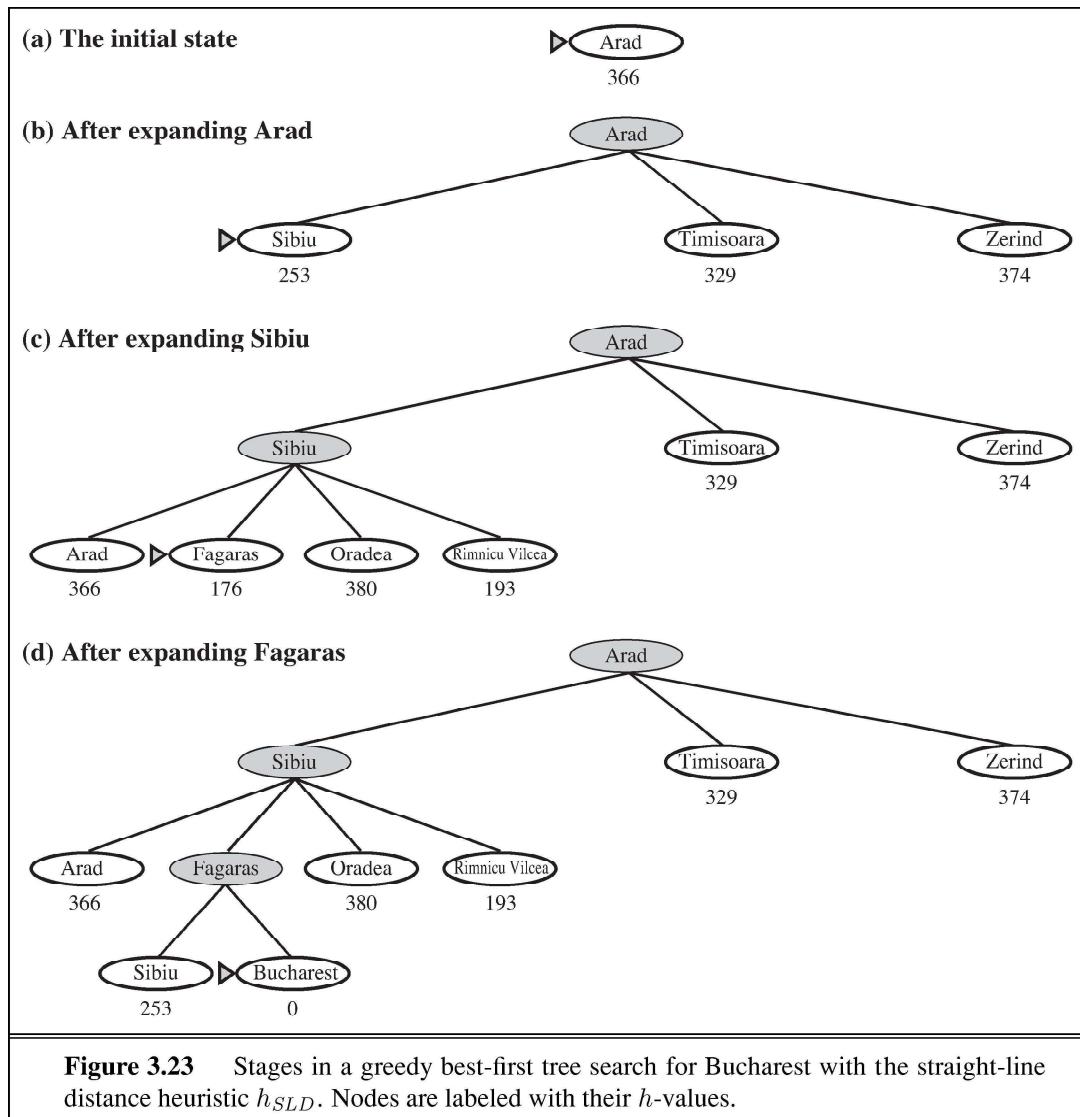
The most widely known form of best-first search is called **A* search** (pronounced “A-star search”). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

$$f(n) = g(n) + h(n).$$

Since $g(n)$ gives the path cost from the start node to node n , and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have

$$f(n) = \text{estimated cost of the cheapest solution through } n.$$

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses $g + h$ instead of g .



Conditions for optimality: Admissibility and consistency

ADMISSIBLE HEURISTIC

The first condition we require for optimality is that $h(n)$ be an **admissible heuristic**. An admissible heuristic is one that *never overestimates* the cost to reach the goal. Because $g(n)$ is the actual cost to reach n along the current path, and $f(n) = g(n) + h(n)$, we have as an immediate consequence that $f(n)$ never overestimates the true cost of a solution along the current path through n .

Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is. An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight

line cannot be an overestimate. In Figure 3.24, we show the progress of an A* tree search for Bucharest. The values of g are computed from the step costs in Figure 3.2, and the values of h_{SLD} are given in Figure 3.22. Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f -cost (450) is higher than that of Pitesti (417). Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.

CONSISTENCY
MONOTONICITY

A second, slightly stronger condition called **consistency** (or sometimes **monotonicity**) is required only for applications of A* to graph search.⁹ A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any 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' :

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

TRIANGLE
INEQUALITY

This is a form of the general **triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n , n' , and the goal G_n closest to n . For an admissible heuristic, the inequality makes perfect sense: if there were a route from n to G_n via n' that was cheaper than $h(n)$, that would violate the property that $h(n)$ is a lower bound on the cost to reach G_n .

It is fairly easy to show (Exercise 3.29) that every consistent heuristic is also admissible. Consistency is therefore a stricter requirement than admissibility, but one has to work quite hard to concoct heuristics that are admissible but not consistent. All the admissible heuristics we discuss in this chapter are also consistent. Consider, for example, h_{SLD} . We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance and that the straight-line distance between n and n' is no greater than $c(n, a, n')$. Hence, h_{SLD} is a consistent heuristic.

Optimality of A*



As we mentioned earlier, A* has the following properties: *the tree-search version of A* is optimal if $h(n)$ is admissible, while the graph-search version is optimal if $h(n)$ is consistent*.

We show the second of these two claims since it is more useful. The argument essentially mirrors the argument for the optimality of uniform-cost search, with g replaced by f —just as in the A* algorithm itself.



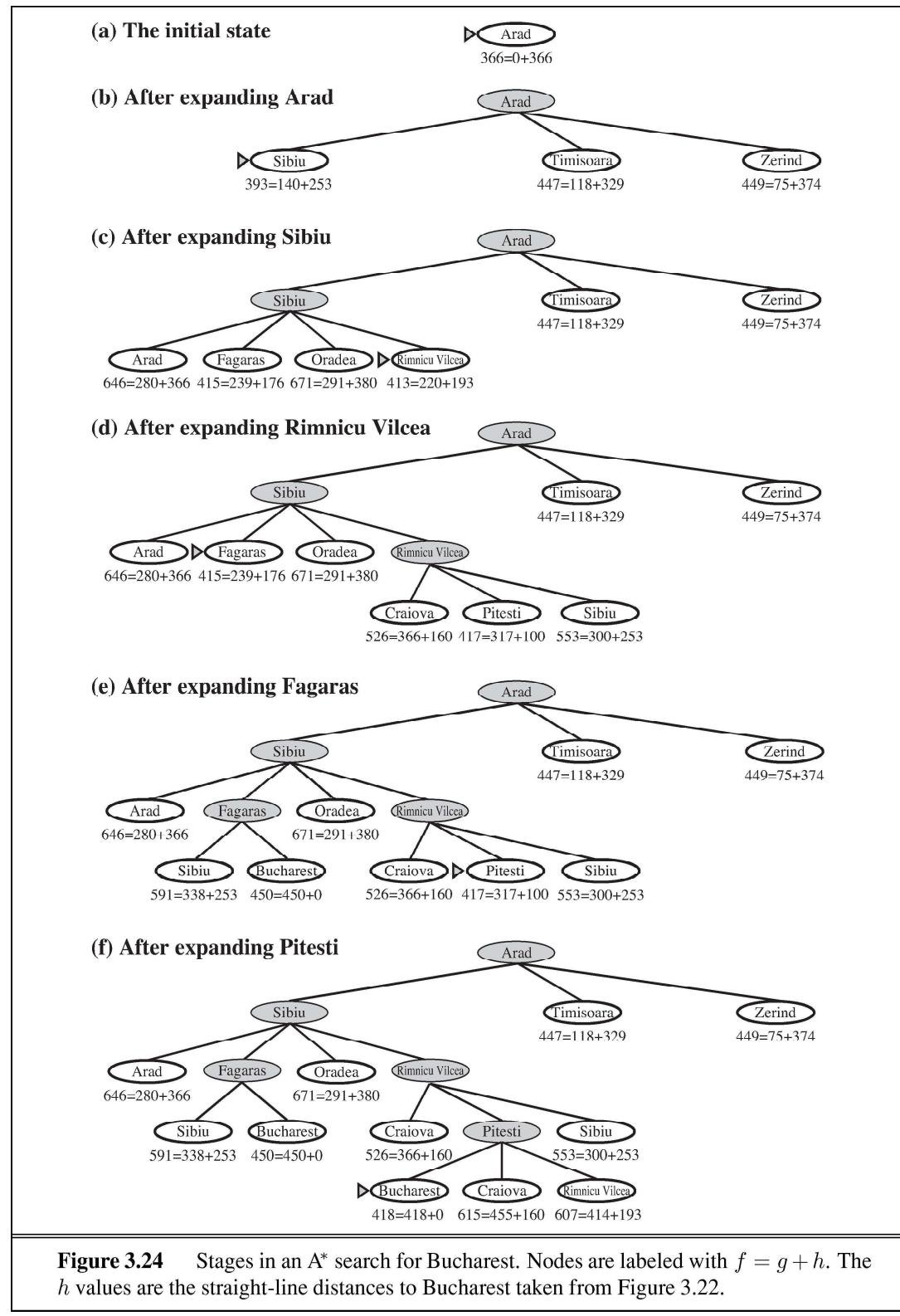
The first step is to establish the following: *if $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing*. The proof follows directly from the definition of consistency. Suppose n' is a successor of n ; then $g(n') = g(n) + c(n, a, n')$ for some action a , and we have

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



The next step is to prove that *whenever A* selects a node n for expansion, the optimal path to that node has been found*. Were this not the case, there would have to be another frontier node n' on the optimal path from the start node to n , by the graph separation property of

⁹ With an admissible but inconsistent heuristic, A* requires some extra bookkeeping to ensure optimality.



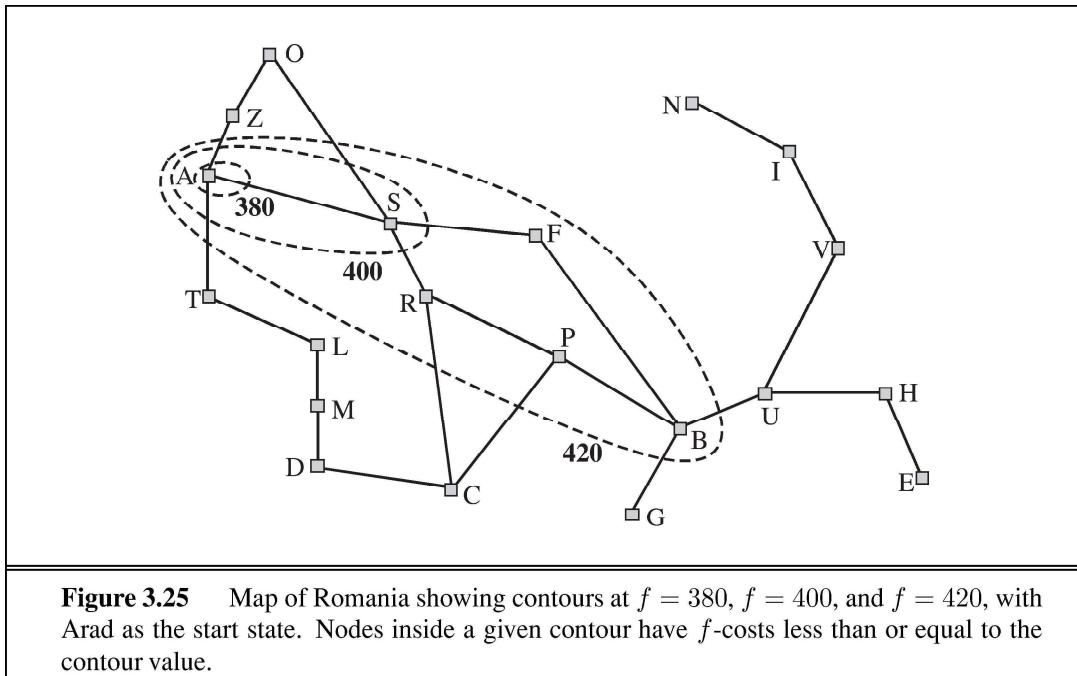


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f -costs less than or equal to the contour value.

Figure 3.9; because f is nondecreasing along any path, n' would have lower f -cost than n and would have been selected first.

From the two preceding observations, it follows that the sequence of nodes expanded by A^* using GRAPH-SEARCH is in nondecreasing order of $f(n)$. Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have $h = 0$) and all later goal nodes will be at least as expensive.

CONTOUR

The fact that f -costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map. Figure 3.25 shows an example. Inside the contour labeled 400, all nodes have $f(n)$ less than or equal to 400, and so on. Then, because A^* expands the frontier node of lowest f -cost, we can see that an A^* search fans out from the start node, adding nodes in concentric bands of increasing f -cost.

With uniform-cost search (A^* search using $h(n) = 0$), the bands will be “circular” around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path. If C^* is the cost of the optimal solution path, then we can say the following:

- A^* expands all nodes with $f(n) < C^*$.
- A^* might then expand some of the nodes right on the “goal contour” (where $f(n) = C^*$) before selecting a goal node.

Completeness requires that there be only finitely many nodes with cost less than or equal to C^* , a condition that is true if all step costs exceed some finite ϵ and if b is finite.

Notice that A^* expands no nodes with $f(n) > C^*$ —for example, Timisoara is not expanded in Figure 3.24 even though it is a child of the root. We say that the subtree below

PRUNING

Timisoara is **pruned**; because h_{SLD} is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI.

OPTIMALLY EFFICIENT

One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root and use the same heuristic information— A^* is **optimally efficient** for any given consistent heuristic. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A^* (except possibly through tie-breaking among nodes with $f(n) = C^*$). This is because any algorithm that *does not* expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution.

ABSOLUTE ERROR

RELATIVE ERROR

That A^* search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A^* is the answer to all our searching needs. The catch is that, for most problems, the number of states within the goal contour search space is still exponential in the length of the solution. The details of the analysis are beyond the scope of this book, but the basic results are as follows. For problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the the **absolute error** or the **relative error** of the heuristic. The absolute error is defined as $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal, and the relative error is defined as $\epsilon \equiv (h^* - h)/h^*$.

The complexity results depend very strongly on the assumptions made about the state space. The simplest model studied is a state space that has a single goal and is essentially a tree with reversible actions. (The 8-puzzle satisfies the first and third of these assumptions.) In this case, the time complexity of A^* is exponential in the maximum absolute error, that is, $O(b^\Delta)$. For constant step costs, we can write this as $O(b^{ed})$, where d is the solution depth. For almost all heuristics in practical use, the absolute error is at least proportional to the path cost h^* , so ϵ is constant or growing and the time complexity is exponential in d . We can also see the effect of a more accurate heuristic: $O(b^{ed}) = O((b^\epsilon)^d)$, so the effective branching factor (defined more formally in the next section) is b^ϵ .

When the state space has many goal states—particularly *near-optimal* goal states—the search process can be led astray from the optimal path and there is an extra cost proportional to the number of goals whose cost is within a factor ϵ of the optimal cost. Finally, in the general case of a graph, the situation is even worse. There can be exponentially many states with $f(n) < C^*$ even if the absolute error is bounded by a constant. For example, consider a version of the vacuum world where the agent can clean up any square for unit cost without even having to visit it: in that case, squares can be cleaned in any order. With N initially dirty squares, there are 2^N states where some subset has been cleaned and all of them are on an optimal solution path—and hence satisfy $f(n) < C^*$ —even if the heuristic has an error of 1.

The complexity of A^* often makes it impractical to insist on finding an optimal solution. One can use variants of A^* that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search. In Section 3.6, we look at the question of designing good heuristics.

Computation time is not, however, A^* 's main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A^* usually runs out of space long

```

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

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  $\leftarrow$  []
    for each action in problem.ACTIONS(node.STATE) do
        add CHILD-NODE(problem, node, action) into successors
    if successors is empty then return failure,  $\infty$ 
    for each s in successors do /* update f with value from previous search, if any */
        s.f  $\leftarrow$  max(s.g + s.h, node.f)
    loop do
        best  $\leftarrow$  the lowest f-value node in successors
        if best.f > f-limit then return failure, best.f
        alternative  $\leftarrow$  the second-lowest f-value among successors
        result, best.f  $\leftarrow$  RBFS(problem, best, min(f-limit, alternative))
        if result  $\neq$  failure then return result

```

Figure 3.26 The algorithm for recursive best-first search.

before it runs out of time. For this reason, A* is not practical for many large-scale problems. There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time. We discuss these next.

3.5.3 Memory-bounded heuristic search

ITERATIVE-
DEEPENING
A*

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the **iterative-deepening A*** (IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the *f*-cost ($g + h$) rather than the depth; at each iteration, the cutoff value is the smallest *f*-cost of any node that exceeded the cutoff on the previous iteration. IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes. Unfortunately, it suffers from the same difficulties with real-valued costs as does the iterative version of uniform-cost search described in Exercise 3.17. This section briefly examines two other memory-bounded algorithms, called RBFS and MA*.

RECURSIVE
BEST-FIRST SEARCH

BACKED-UP VALUE

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. The algorithm is shown in Figure 3.26. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the *f_limit* variable to keep track of the *f*-value of the best *alternative* path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the *f*-value of each node along the path with a **backed-up value**—the best *f*-value of its children. In this way, RBFS remembers the *f*-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth