RBFS - Recursive Best-First Search

- Mimics best-first search with linear space
- Similar to recursive depth-first
  - Limits recursion by keeping track of the f-value of the best alternative path from any ancestor node – one step look-ahead
  - If current node exceeds this value, recursion unwinds back to the alternative path – same idea as contour
  - As recursion unwinds, replaces f-value of node with best f-value of children
  - Allows to remember whether to re-expand path at later time
- Exploits information gathered from previous searches about minimum f so as to focus further searches

RBFS - Recursive Best-First Search Algorithm

```plaintext
function RBFS(problem, node, f-bound) returns a solution, or failure
RBFS(Make-Node(Initial-State(problem)), nil)

function RBFS(problem, node, f-bound) returns a solution, or failure and a new f-bound limit
if Goal-Test(node) then return node
successor -- Expand(node, problem)
if successor is empty then return failure, nil
repeat
for each (successor, (f, path)) in Expand do
    if f > f-bound then return failure, nil
    alternative -- the search limit is increasing
    result, f-bound' = RBFS(problem, path, f-bound + Pathmax heuristic;
    if result is not failure then return result
end
```

Define contour limit

Define next highest f-contour

Recursive search on best successor, remember when to backup
All successors above limit of 415, back up value of Pitesti 417 as lowest.

Avoids going down this path again unless lowest unexplored f-contour is 417.

(c) After switching back to Rimnicu Vilcea and expanding Pitesti.
RBFS -- Pro’s and Con’s

- More efficient than IDA* and still optimal
  - Best-first Search based on next best f-contour; fewer regeneration of nodes
  - Exploit results of search at a specific f-contour by saving next f-contour associated with a node who successors have been explored.
- Like IDA* still suffers from excessive node regeneration
- IDA* and RBFS not good for graphs
  - Can’t check for repeated states other than those on current path
- Both are hard to characterize in terms of expected time complexity

SMA*(Simplified Memory-Bounded A*)

- Uses a given amount of memory to remember nodes so that they don’t have to be repeatedly regenerated
- It will utilize whatever memory is made available to it.
- It avoids repeated states as far as its memory allows.

SMA*

- Expand deepest lowest f-cost leaf-node
  - Best first search on f-cost
- Update f-cost of nodes whose successors have higher f-cost
- Drop shallowest & highest f-cost leaf node
  - remember best forgotten descendant
- Paths longer than node limit get \( \infty \) cost.

SMA* Example
SMA* Example (3-node limit)

Update A based on lowest cost f successor? Remember next lowest cost f node B that is removed

Reach goal node I but it is not the cheapest so continue search

Regenerate node B, remember that there was node with f=15 that had been removed, remember successor of G has f=24

Why don’t we need to search anymore after finding D.

SMA* Analysis

- It is complete, provided the available memory is sufficient to store the shallowest solution path.
- It is optimal, if enough memory is available to store the shallowest optimal solution path. Otherwise, it returns the best solution (if any) that can be reached with the available memory.
- Can keep switching back and forth between a set of candidate solution paths, only a few of which can fit in memory (thrashing)

Memory limitations can make a problem intractable with respect to time

- With enough memory for the entire tree, same as A*

SMA* Example (3-node limit) cont.

Sketch of SMA* Algorithm
Memory-bounded heuristic search

- **IDA** - Iterative-deepening A*
  - Use f-cost (g+h) as cutoff
  - At each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration
- **Recursive best-first search (RBFS)**
  - Best-first search with only linear space
  - Keep track of the f-value of the best alternative
  - As the recursion unwinds, it forgets the sub-tree and back-up the f-value of the best leaf as its parent’s f-value.
- **SMA** proceeds like A*
  - Expanding the best leaf until memory is full
  - Drop the worst leaf node, and back-up the value of the forgotten node to its parent.
  - Complete IF there is any reachable solution.
  - Optimal IF any optimal solution is reachable.

Approaches for Reducing Search Cost

- **Staged search** involves periodically pruning unpromising paths
  - SMA is an example of a staged search
- Node expansion may be so costly (because the branching factor is high or the cost to apply operators is high) that exhaustive node expansion is not practical.

Heuristic node expansion

- Use a generator approach to incrementally produce successors ordered by quality (must have operator-ordering function);
- Limit expansion so that only likely successors are generated (often called plausible-move generator);
- Prune unpromising successors immediately following node expansion;
- Delay state computation until expansion time when possible (must be able to compute h without state only on operator/previous state).

Real-Time Concerns
Real-time problem solving

- Practical and theoretical difficulties:
  - Agents have limited computational power.
  - They must react within an acceptable time.
  - Computation time normally reduces the value of the result.
  - There is a high degree of uncertainty regarding the rate of progress.
  - The “appropriate” level of deliberation is situation dependent.

Simon’s “Bounded-Rationality”

“...a theory of rationality that does not give an account of problem solving in the face of complexity is badly incomplete. It is worse than incomplete; it can be seriously misleading by providing “solutions” that are without operational significance.”

“The global optimization problem is to find the least-cost or best-return decision, net of computational costs.”

-- Herbert Simon, 1958

Satisficing

- A Scottish word which means satisfying.
- Denotes decision making that searches until an alternative is found that is satisfactory by the agent's aspiration level criterion.
- Heuristic search as satisficing.
- Formalizing the notion of satisficing.

Satisficing versus Optimizing

“It appears probable that, however adaptive the behavior of organisms in learning and choice situations, this adaptiveness falls far short of the ideal “maximizing” postulated in economic theory. Evidently, organisms adapt well enough to “satisfice”; they do not, in general, “optimize.”
Optimizing in the Real-World

“In complex real-world situations, optimization becomes approximate optimization since the description of the real-world is radically simplified until reduced to a degree of complication that the decision maker can handle. Satisficing seeks simplification in a somewhat different direction, retaining more of the detail of the real-world situation, but settling for a satisfactory, rather than approximate-best, decision.”

Which approach is preferable?

Anytime algorithms

- Ideal (maximal quality in no time)
- Traditional (quality maximizing)
- Anytime (utility maximizing)

Value is a combination of quality of solution and amount of time to arrive at solution
Anytime A*

- A* is best first search with $f(n) = g(n) + h(n)$

- Three changes make it an anytime algorithm:
  1. Use a non-admissible heuristic so that sub-optimal solutions are found quickly.
  2. Continue the search after the first solution is found using it to prune the open list.
  3. When the open list is empty, the best solution generated is optimal.

- How to choose a non-admissible heuristic?

Weighted evaluation functions

- Use $f'(n) = (1 - w)g(n) + w * h(n)$
- Higher weight on $h(n)$ tends to search deeper.
- Admissible if $h(n)$ is admissible and $w \leq 0.5$
  - Same relative node ordering as admissible heuristic $h(n) \geq w * h(n) / (1 - w)$ and $f(n) = f'(n) / (1 - w)$
  - Otherwise, the search is non-admissible, but it normally finds solutions much faster.
  - *An appropriate w makes possible a tradeoff between the solution quality and the computation time.*

### Algorithm 1: Anytime WA*

```
Input: A* starts here
Output: Best solution found and error bound

begin
  $g(start) = 0$, $f(start) = h(start)$, $f'(start) = w * h(start)$
  OPEN = [start], CLOSED = [] (incumbent)
  while OPEN ≠ [] and not incumbent do
    $n$ = select min-cost node
    if incumbent = null or $f(n) < f'(incumbent)$ then
      incumbent = new incumbent
      $f'(incumbent) = f(n)$
      if $n$ is a goal node then
        return incumbent
      else if $n$ ∈ CLOSED and $g(n) + g(n) = g(n)$ then
        return incumbent
      else if $n$ ∈ OPEN and $g(n) + g(n) < g(n)$ then
        $g(n)$ = $g(n)$ + $h(n)$, $f(n)$ = $g(n) + w * h(n)$
        $n$ = push $n$ into OPEN
        if have found shorter path to node update
        CLOSED = CLOSED \ {n}
      end
    end
  end

if OPEN = [] then error = 0
else error = $|f'(incumbent) - f'(node)|$ : bound error

Could you prune open list after
output incumbent solution and error bound

end
```

Pruning States in Anytime A*

- For each node, store real $f(n) = g(n) + h(n)$
  - $f(n)$ is the lower bound on the cost of the best solution path through $n$
- When find solution/goal node $n_1$
  - $f(n_1)$ is an upper bound of the cost of the optimal solution
  - Prune all nodes $n$ on the open list that have $f(n) >= f(n_1)$
Adjusting W Dynamically

Suppose you had the following situations, how would you adjust w.

- the open list has gotten so large that you are running out of memory?
- you are running out of time and you have not yet reached an answer?
- there are a number of nodes on the open list whose h value is very small?