HEURISTIC SEARCH

Ivan Bratko
Faculty of Computer and Information Sc.
University of Ljubljana

Best-first search

- Best-first: most usual form of heuristic search

- Evaluate nodes generated during search

- Evaluation function

  f: Nodes ---> R^+

- Convention: lower f, more promising node; higher f indicates a more difficult problem

- Search in directions where f-values are lower

Heuristic search algorithms

- A*, perhaps the best known algorithm in AI
- Hill climbing, steepest descent
- Beam search
- IDA*
- RBFS

Heuristic evaluation in A*

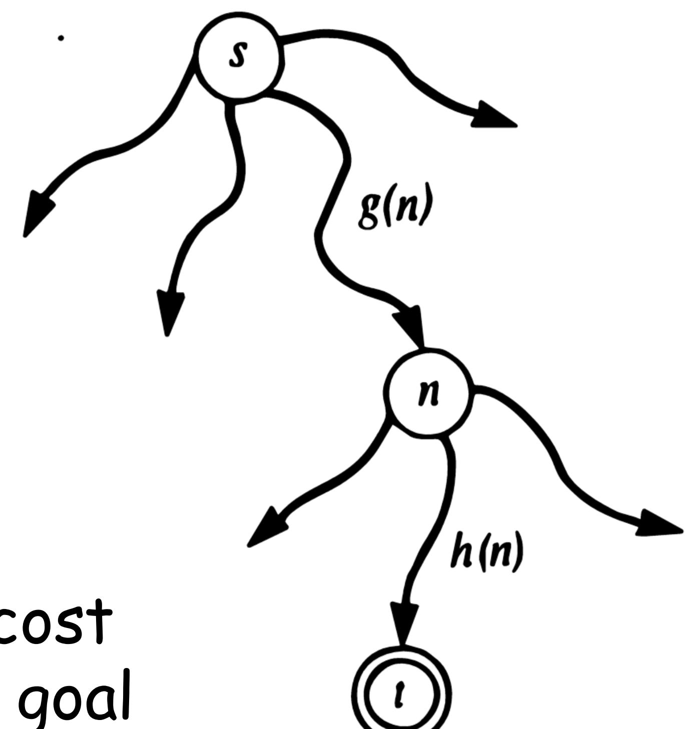
The question: How to find successful f?

Algorithm A*:
f(n) estimates cost of best solution through *n*

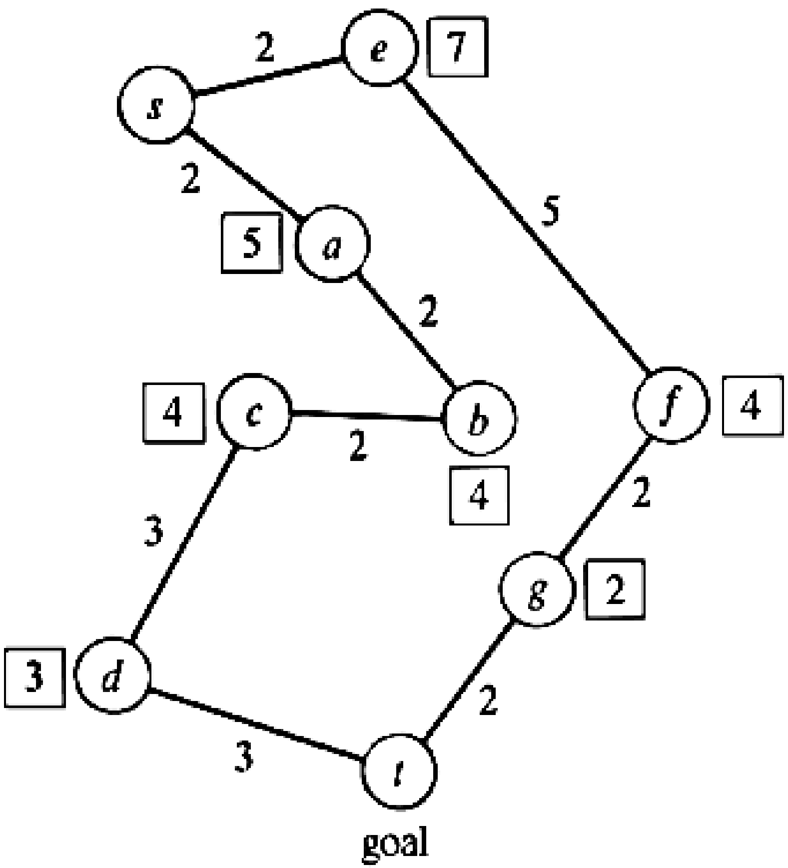
$$f(n) = g(n) + h(n)$$

known

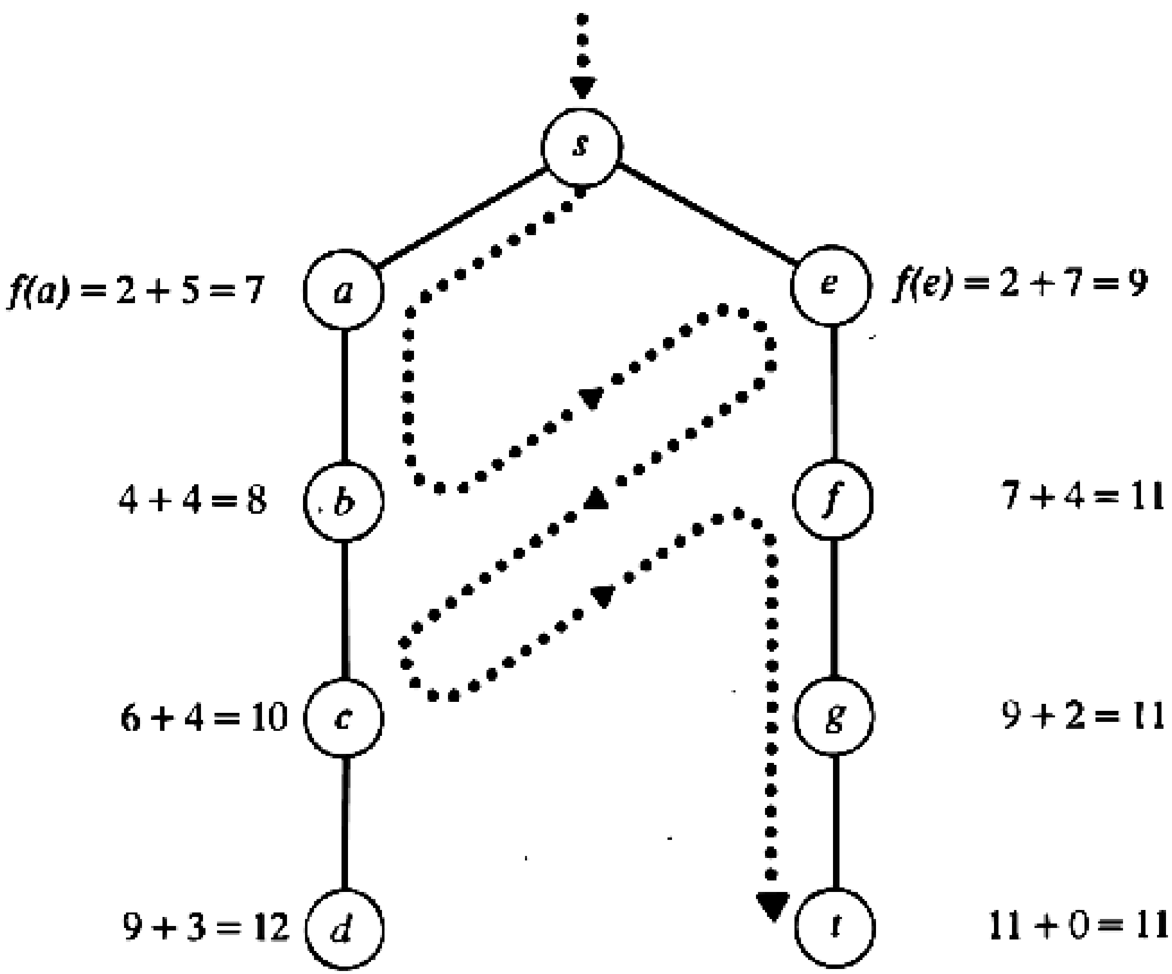
heuristic guess, estimate of path cost from *n* to nearest goal



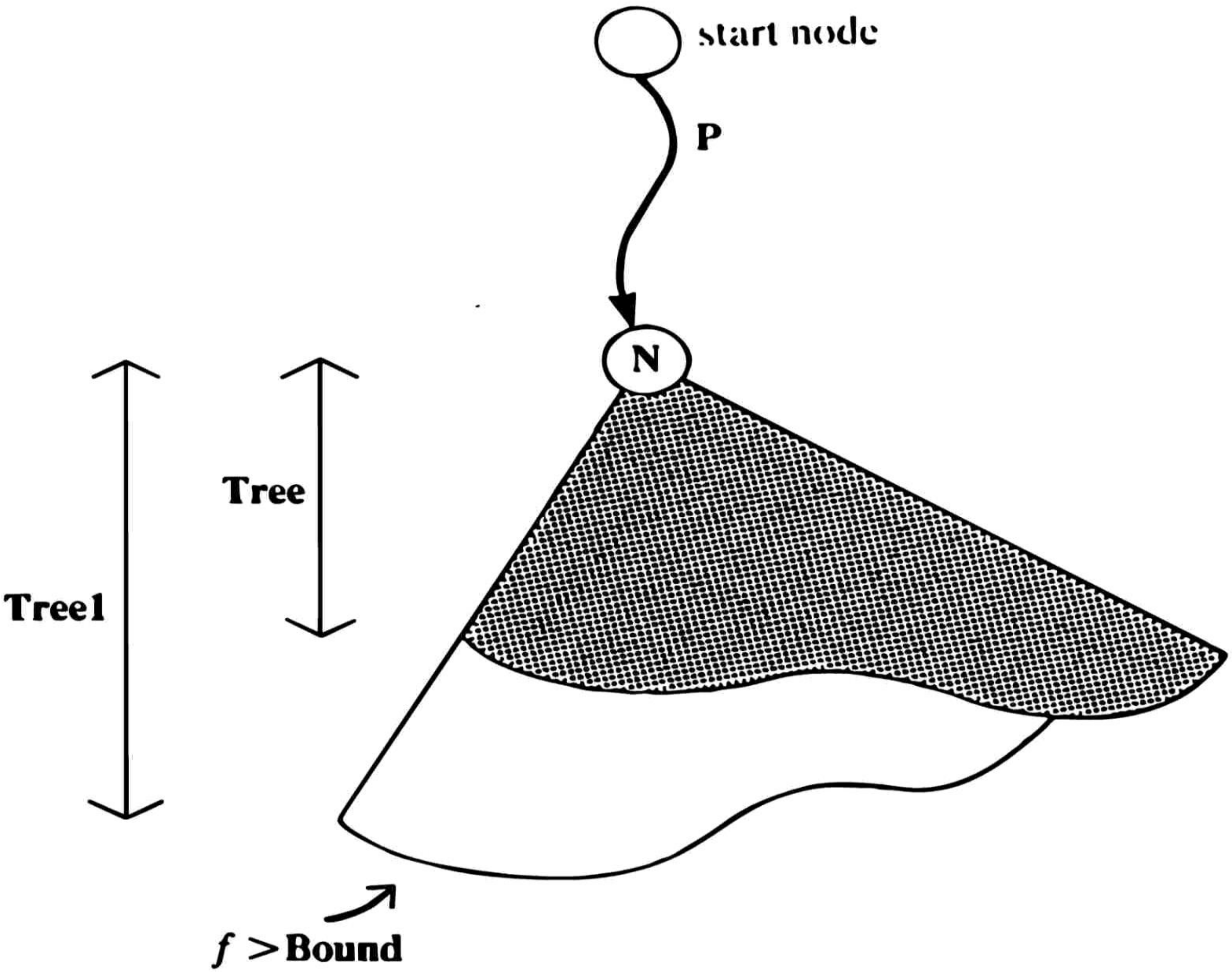
Route search in a map



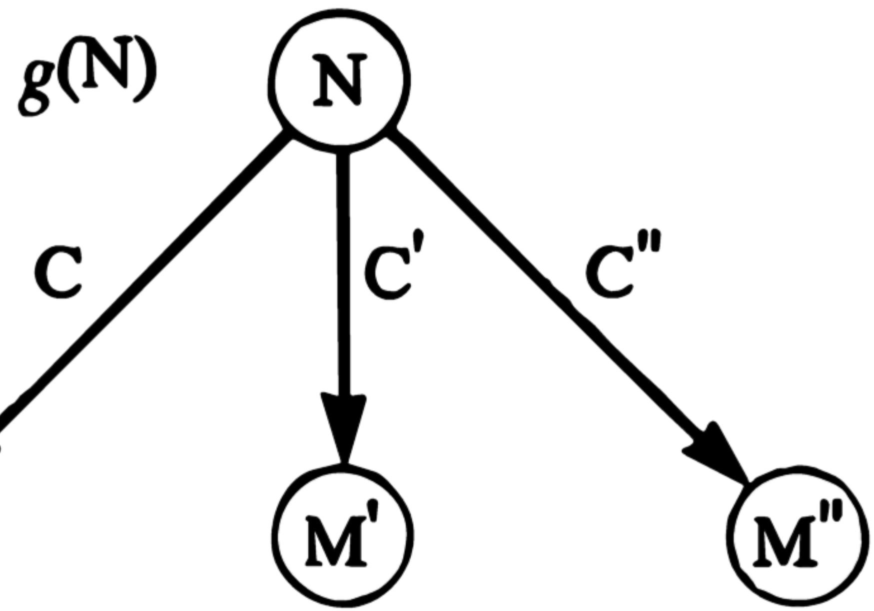
Best-first search for shortest route



Expanding tree within Bound



g-value and f-value of a node



$$g(M) = g(N) + C$$
$$f(M) = g(M) + h(M)$$

Admissibility of A*

- A search algorithm is *admissible* if it is guaranteed to always find an optimal solution
- Is there any such guarantee for A*?

- Admissibility theorem (Hart, Nilsson, Raphael 1968):

A* is admissible if $h(n) =< h^*(n)$ for all nodes n in state space. $h^*(n)$ is the actual cost of min. cost path from n to a goal node.

Admissible heuristics

- A* is admissible if it uses an optimistic heuristic estimate
- Consider $h(n) = 0$ for all n.

  This trivially admissible!
- However, what is the drawback of $h = 0$?
- How well does it guide the search?

- Ideally: $h(n) = h^*(n)$

- Main difficulty in practice: Finding heuristic functions that guide search well and are admissible

Finding good heuristic functions

- Requires problem-specific knowledge
- Consider 8-puzzle
- h = total_dist = sum of Manhattan distances of tiles from their "home" squares

1	3	4
8	5	
7	6	2

total_dist = 0+3+1+1+2+0+0+0=7

- Is total_dist admissible?

Heuristics for 8-puzzle

sequence_score : assess the order of tiles;
count 1 for tile in middle, and 2 for each tile on edge not followed by proper successor clockwise

1	3	4
8	5	
7	6	2

sequence_score = 1+2+0+2+2+0+0+0 = 7

h = total_dist + 3 * sequence_score
h = 7 + 3*7 = 28
Is this heuristic function admissible?

Three 8-puzzles

1	3	4
8		2
7	6	5

(a)

2	8	3
1	6	4
7		5

(b)

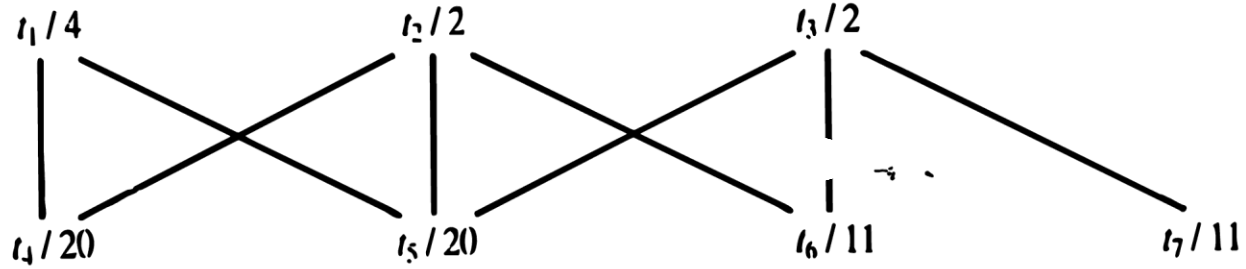
2	1	6
4		8
7	5	3

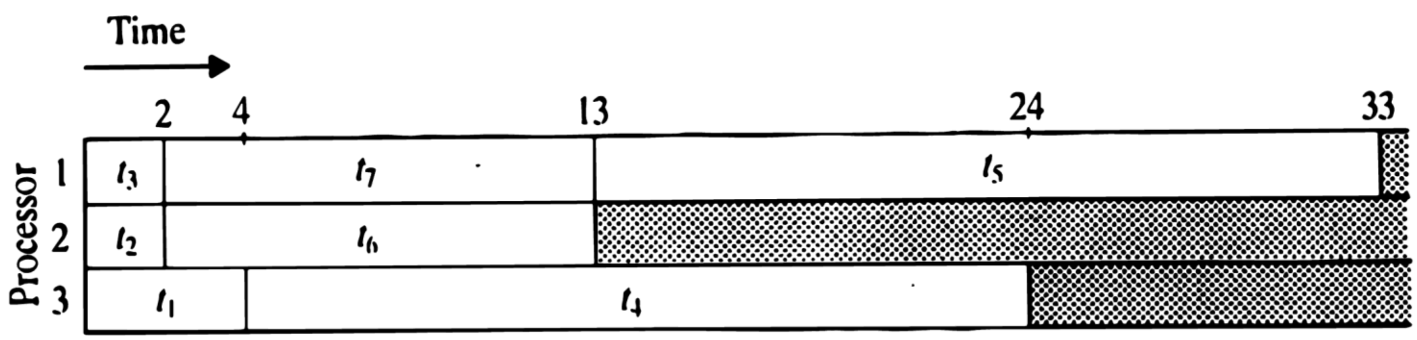
(c)

Puzzle c requires 18 steps

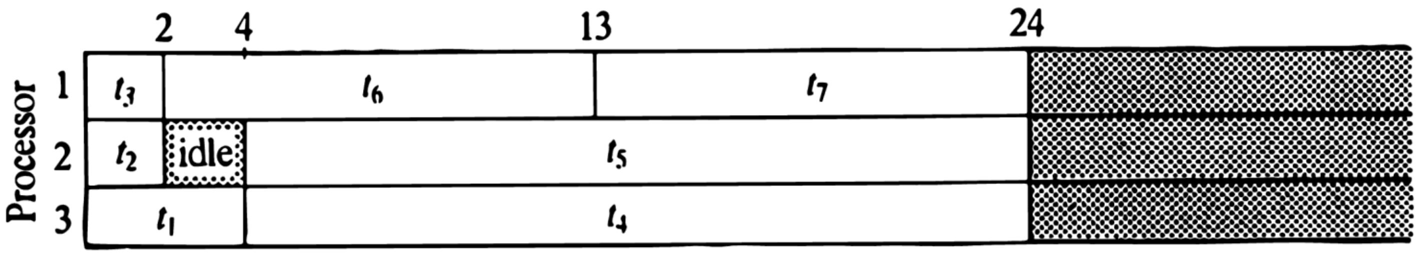
A* with this h solves (c) almost without any branching

A task-scheduling problem and two schedules





Optimal



Heuristic function for scheduling

- Fill tasks into schedule from left to right
- A heuristic function:

  For current, partial schedule:

  F_J = finishing time of processor J current engagement

  Fin = max F_J

  Finall = ($SUM_W(D_W)$ + $SUM_J(F_J)$) / m

  W: waiting tasks; D_W: duration of waiting task W

  h = max(Finall - Fin, 0)

- Is this heuristic admissible?

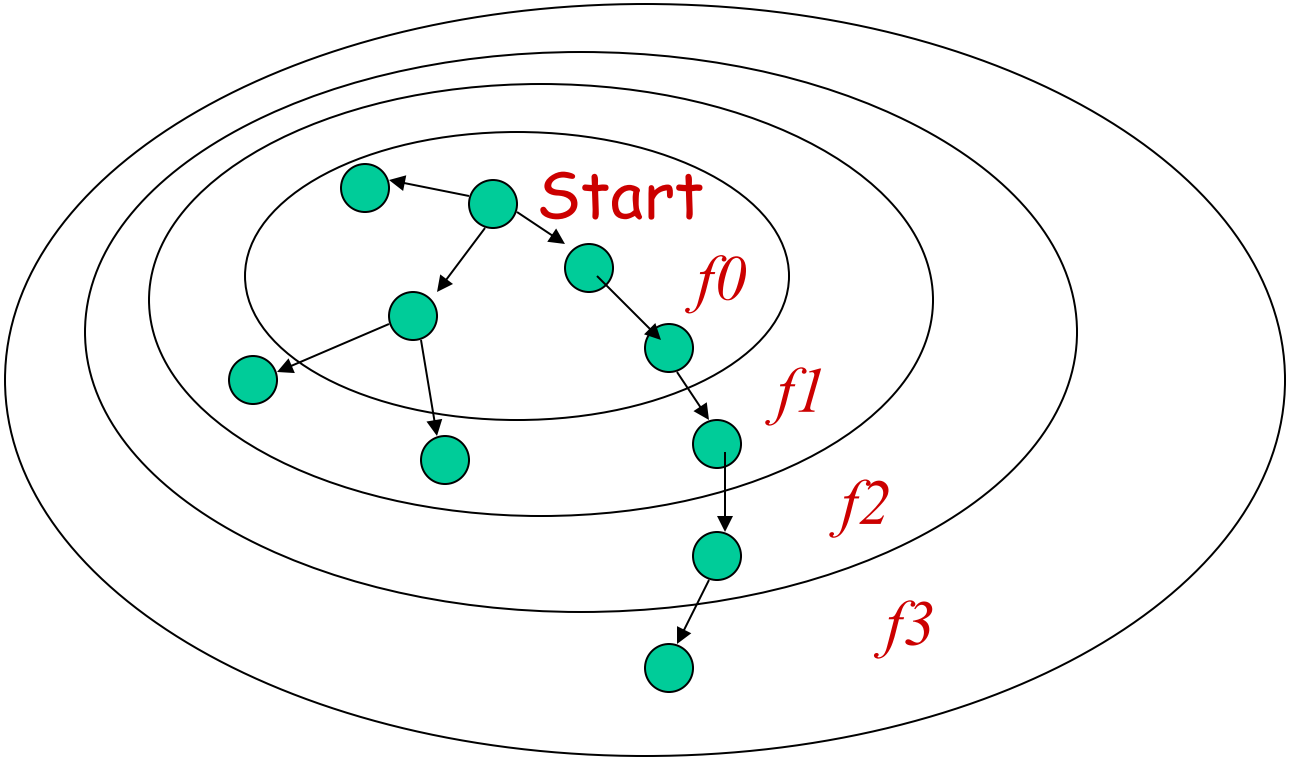
Space Saving Techniques for Best-First Search

- Space complexity of A*: depends on h, but it is roughly b^d
- Space may be critical resource
- Idea: trade time for space, similar to iterative deepening
- Space-saving versions of A*:
- IDA* (Iterative Deepening A*)
- RBFS (Recursive Best First Search)

IDA*, Iterative Deepening A*

- Introduced by Korf (1985)
- Analogous idea to iterative deepening
- Iterative deepening:
  Repeat depth-first search within increasing depth limit
- IDA*:
  Repeat depth-first search within increasing f-limit

f-values form relief



IDA*: Repeat depth-first within f-limit increasing:
$f0, f1, f2, f3, \ldots$

IDA*

Bound := f(StartNode);

SolutionFound := false;

Repeat

perform depth-first search from StartNode, so that

a node N is expanded only if f(N) $\leq$ Bound;

if this depth-first search encounters a goal node with f $\leq$ Bound

then SolutionFound = true

else

compute new bound as:

Bound = min { f(N) | N generated by this search, f(N) > Bound }

until solution found.

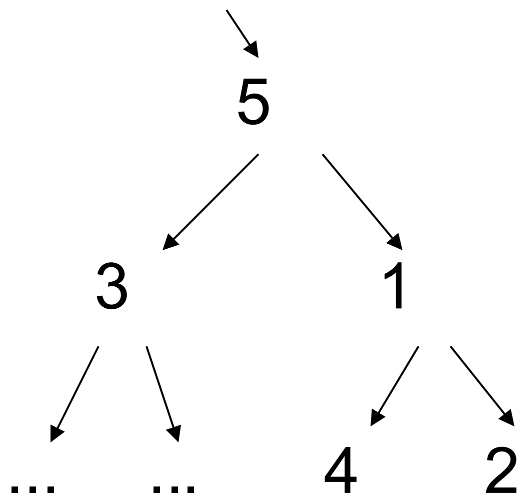
IDA* performance

- Experimental results with 15-puzzle good
  (h = Manhattan distance, many nodes same f)
- However: May become very inefficient when nodes do not tend to share f-values (e.g. small random perturbations of Manhattan distance)

IDA*: problem with non-monotonic f

- Function f is *monotonic* if:

  for all nodes n1, n2: if s(n1,n2) then f(n1) $\leq$ f(n2)
- Theorem: If f is monotonic then IDA* expands nodes in best-first order
- Example with non-monotonic f:

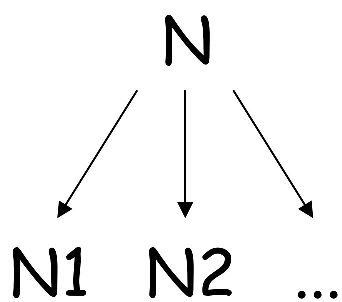


f-limit = 5, “3” expanded before “1”, “2”

Linear Space Best-First Search

- Linear Space Best-First Search
- RBFS (Recursive Best First Search), Korf 93
- Similar to A* implementation in Bratko (86; 2001), but saves space by iterative re-generation of parts of search tree

Node values in RBFS



f(N) = 'static' f-value

F(N) = backed-up f-value,
i.e. currently known lower bound
on cost of solution path through N

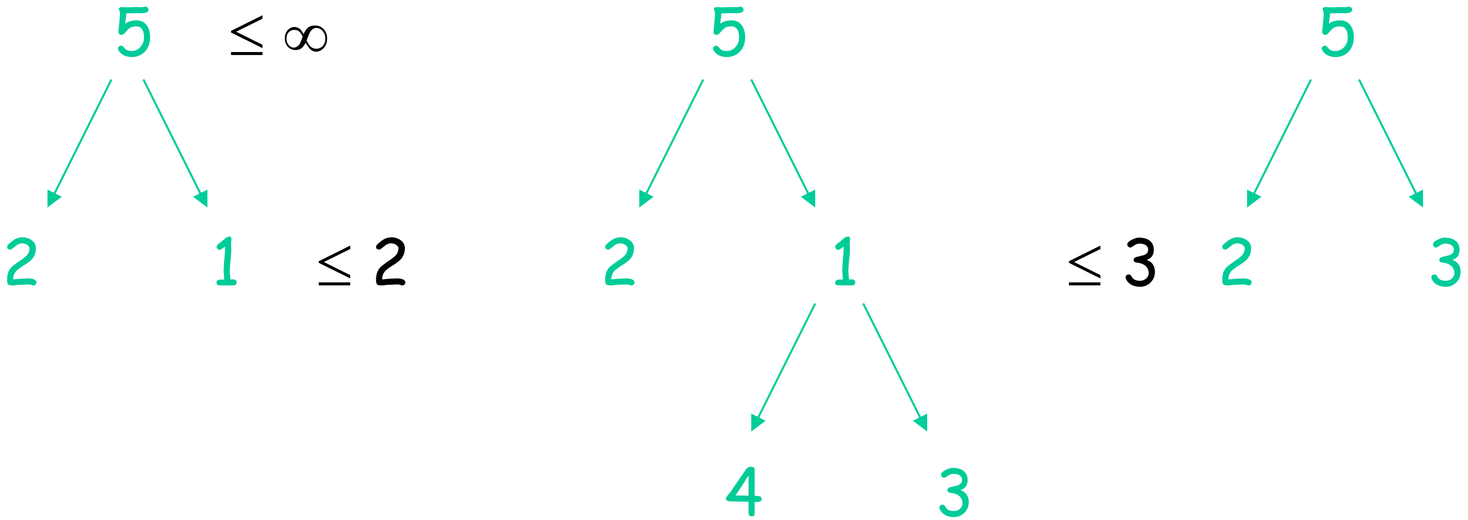
F(N) = f(N) if N has (never) been expanded

F(N) = $\min_i$ F(N_i) if N has been expanded

Simple Recursive Best-First Search SRBFS

- First consider SRBFS, a simplified version of RBFS
- Idea: Keep expanding subtree within F-bound determined by siblings
- Update node’s F-value according to searched subtree
- SRBFS expands nodes in best-first order

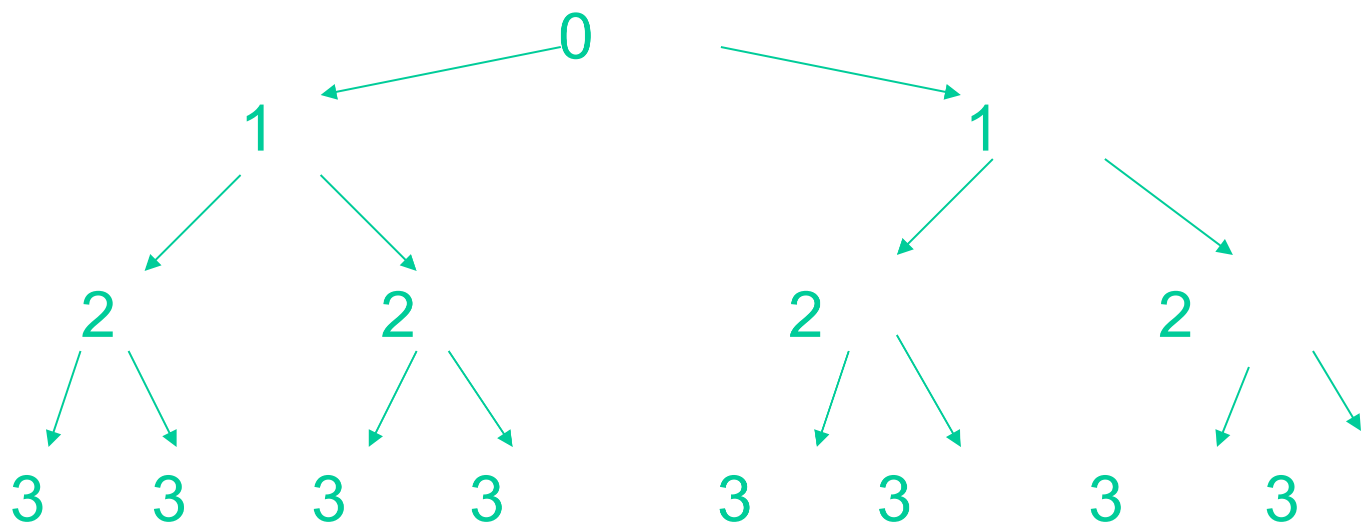
Example: Searching tree with non-monotonic f-function



SRBFS can be very inefficient

Example: search tree with

f(node)=depth of node



Parts of tree repeatedly re-explored;

f-bound increases in unnecessarily small steps

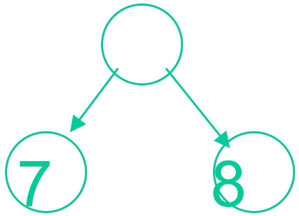
RBFS, improvement of SRBFS

- F-values not only backed-up from subtrees, but also *inherited* from parents

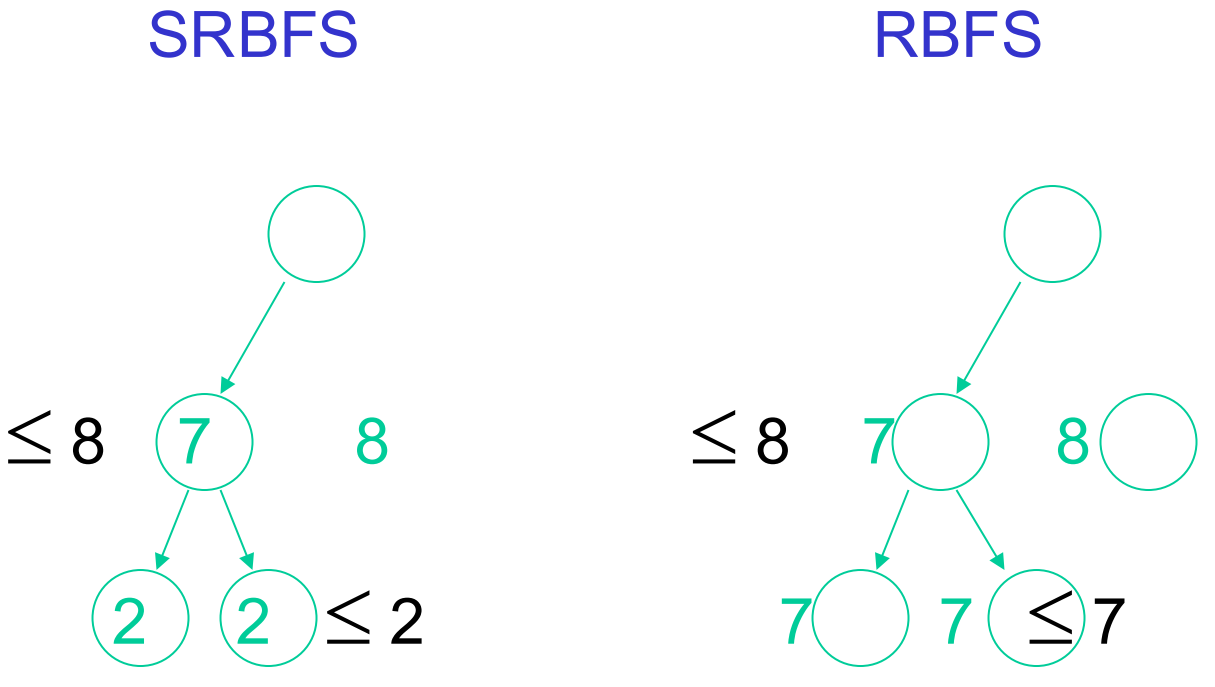
- Example: searching tree with f = depth

- At some point, F-values are:

≤ 8



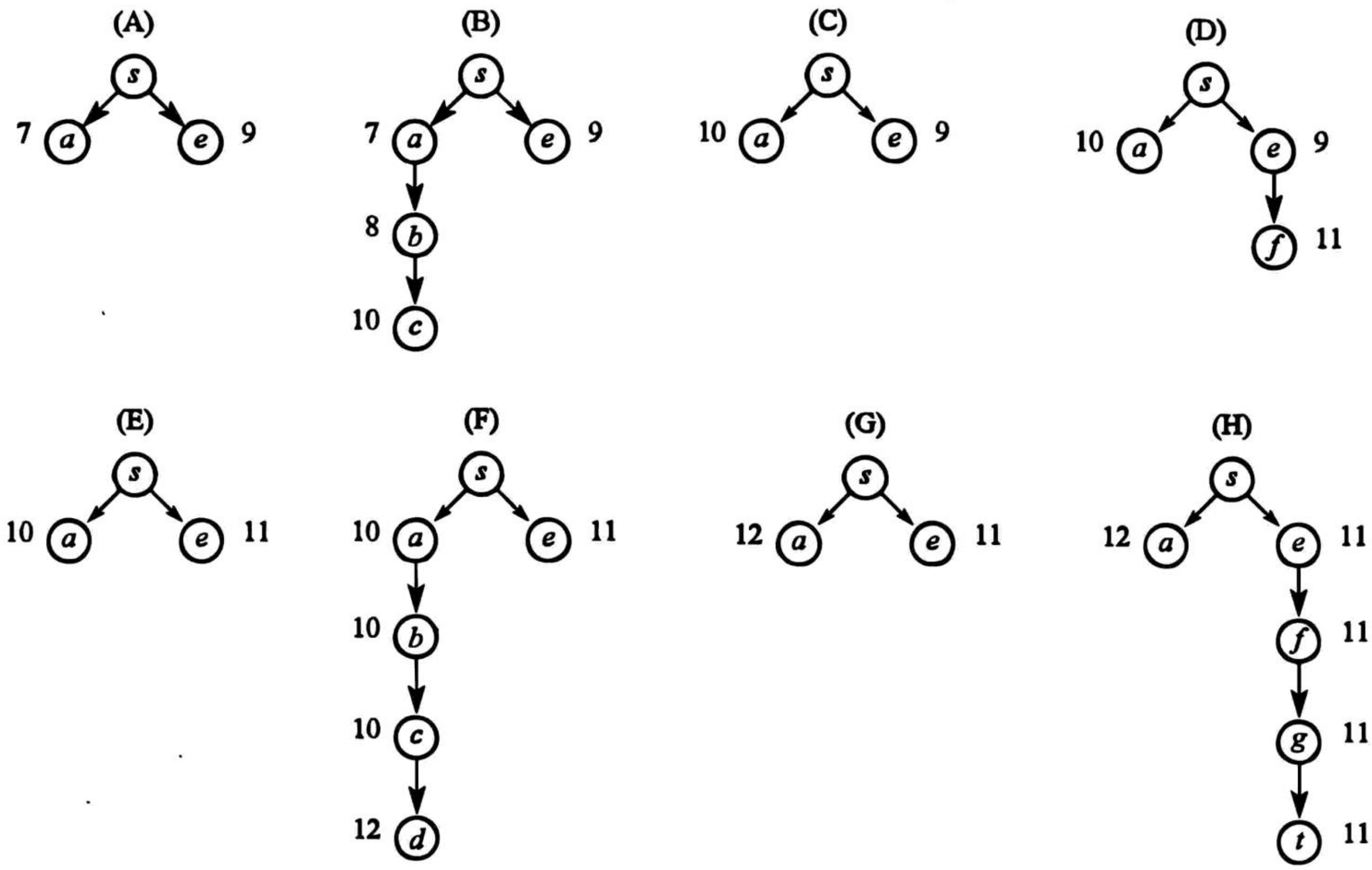
In RBFS, “7” is expanded, and F-value inherited:



F-values of nodes

- To save space RBFS often removes already generated nodes
- But: If N1, N2, ... are deleted, essential info. is retained as F(N)
- Note: If $f(N) < F(N)$ then (part of) N's subtree must have been searched

Searching the map with RBFS



Algorithm RBFS

```
RBFS( N, F(N), Bound)
    if F(N) > Bound then return F(N);
    if goal(N) then exit search;
    if N has no children then return ∞;
    for each child Ni of N do
        if f(N) < F(N) then F(Ni) := max(F(N),f(Ni)
                       else F(Ni) := f(Ni);
    Sort Ni in increasing order of F(Ni);
    if only one child then F(N2) := ∞;
    while F(N1) ≤ Bound and F(N1) < ∞ do
          F(N1) := RBFS(N1,F(N1),min(Bound,F(N2)))
          insert N1 in sorted order
    return F(N1).
```

Properties of RBFS

- RBFS(Start, f(Start), ∞) performs complete best-first search

- Space complexity: O(bd)
  (linear space best-first search)

- RBFS explores nodes in best-first order even with non-monotonic f

  That is: RBFS expands "*open* " nodes in best-first order

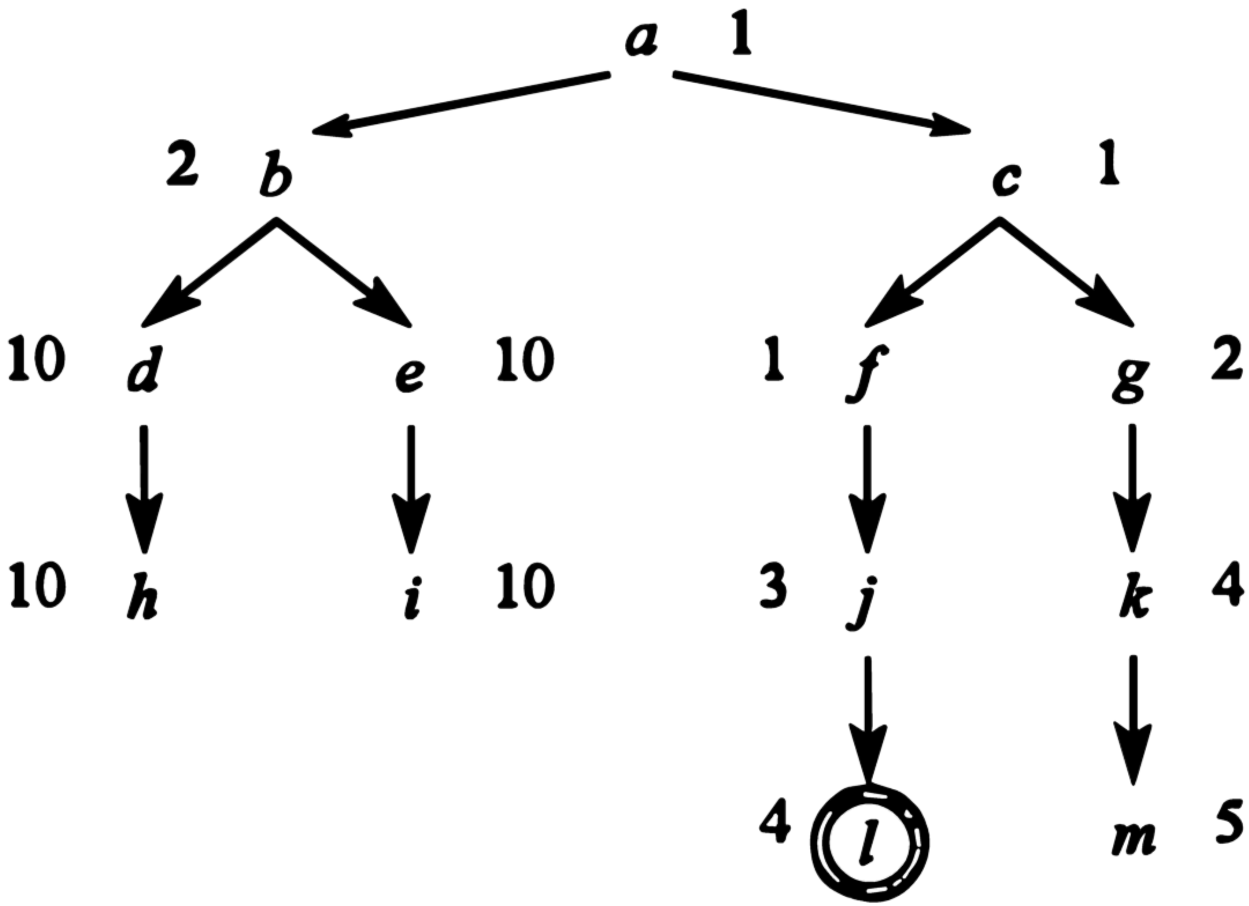
Summary of concepts in best-first search

- Evaluation function: f
- Special case (A*): f(n) = g(n) = h(n)
- h(n) admissible if h(n) =< h*(n)
- Sometimes in literature: A* defined with admissible h
- Algorithm “respects” best-first order if already generated nodes are expanded in best-first order according to f
- f is defined with intention to reflect the “goodness” of nodes; therefore it is desirable that an algorithm respects best-first order
- f is monotonic if for all n1, n2: s(n1,n2) => f(n1) =< f(n2)

Best-first order

- Algorithm “respects” best-first order if generated nodes are expanded in best-first order according to f
- f is defined with intention to reflect the “goodness” of nodes; therefore it is desirable that an algorithm respects best-first order
- f is monotonic if for all n1, n2: s(n1,n2) => f(n1) =< f(n2)
- A* and RBFS respect best-first order
- IDA* respects best-first order if f is monotonic

Problem. How many nodes are generated by A*, IDA* and RBFS? Count also re-generated nodes



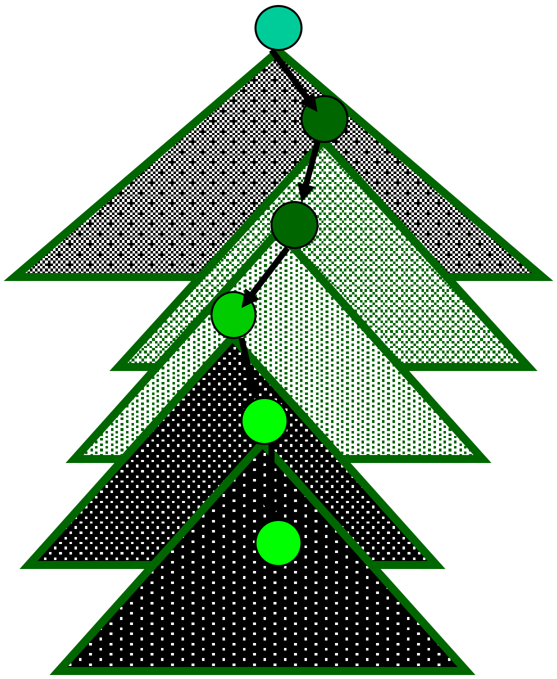
SOME OTHER BEST-FIRST TECHNIQUES

- Hill climbing, steepest descent, greedy search: special case of A* when the successor with best F is retained only; no backtracking
- Beam search: special case of A* where only some limited number of open nodes are retained, say W best evaluated open nodes (W is beam width)
- In some versions of beam search, W may change with depth. Or, the limitation refers to number of successors of an expanded node retained

REAL-TIME BEST FIRST SEARCH

- With limitation on problem-solving time, agent (robot) has to make decision before complete solution is found
- RTA*, real-time A* (Korf 1990):
  - Agent moves from state to next best-looking successor state after fixed depth lookahead
  - Successors of current node are evaluated by backing up f-values from nodes on lookahead-depth horizon
  - $f = g + h$

RTA* planning and execution

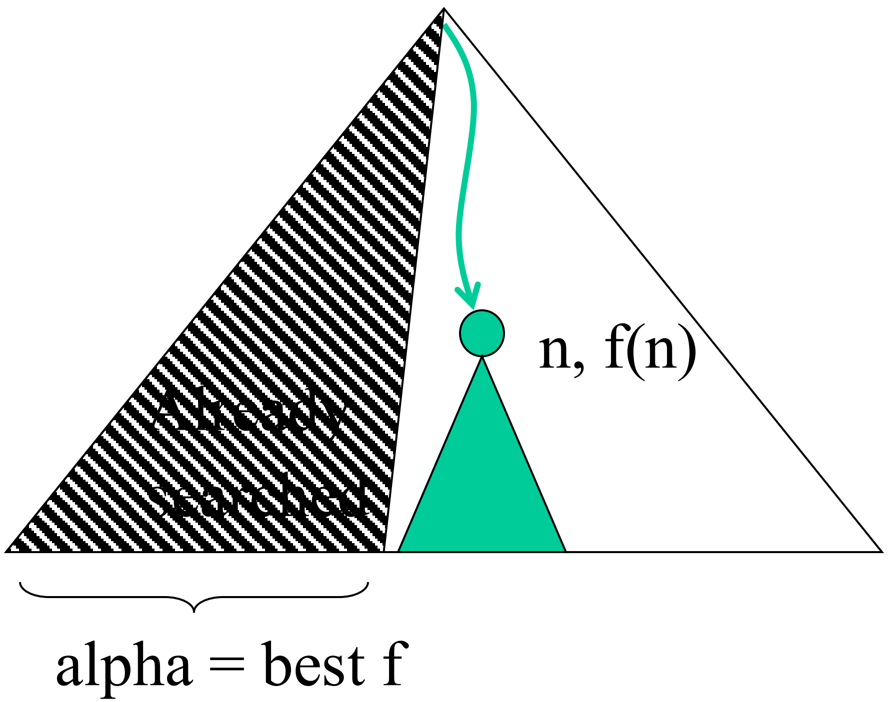


RTA* and alpha-pruning

- If f is monotonic, *alpha-pruning* is possible (with analogy to alpha-beta pruning in minimax search in games)
- Alpha-pruning: let best_f be the best f-value at lookahed horizon found so far, and n be a node encountered before lookahead horizon; if $f(n) \geq$ best_f then subtree of n can be pruned
- Note: In practice, many heuristics are monotonic (Manhattan distance, Euclidean distance)

Alpha pruning

- Prune n's subtree if f(n) ≥ alpha
- If f is monotonic then all descendants of n have f ≥ alpha



RTA*, details

- Problem solving = planning stage + execution stage
- In RTA*, planning and execution interleaved
- Distinguished states: start state, current state
- Current state is understood as actual physical state of robot reached aftre physically executing a move
- Plan in current state by fixed depth lookahead, execute one move to reach next current state

RTA*, details

- $g(n)$, $f(n)$ are measured relative to current state (not start state)
- $f(n) = g(n) + h(n)$, where $g(n)$ is cost from current state to n (not from start state)

RTA* main loop, roughly

- current state s := start_state
- While goal not found:
  - Plan: evaluate successors of s by fixed depth lookahead;
  - best_s := successor with min. backed-up f
  - second_best_f := f of second best successor
  - Store s among “visited nodes” and store

    f(s) := f(second_best_f) + cost(s,best_s)
  - Execute: current state s := best_s

RTA*, visited nodes

- Visited nodes are nodes that have been (physically) visited (i.e. robot has moved to these states in past)
- Idea behind storing f of a visited node s as:

  f(s) := f(second_best_f) + cost(s,best_s)
- If best_s subsequently turns out as bad, problem solver will return to s and this time consider s's second best successor; cost(s, best-s) is added to reflect the fact that problem solver had to pay the cost of moving from s to best_s, in addition to later moving from best_s to s

RTA* lookahead

For node n encountered by lookahead:

- if goal(n) then return h(n) = 0,
  don't search beyond n
- if visited(n) then return h(n) = stored f(n),
  don't search beyond n
- if n at lookahead horizin then evaluate n statically by heuristic function h(n)
- if n not at lookahead horizon then generate n's successors and back up f value from them

LRTA*, Learning RTA*

- Useful when successively solving multiple problem instance with the same goal
- Trial = Solving one problem instance
- Save table of visited nodes with their backed-up h values
- Note: In table used by LRTA*, store the best successors' f (rather than second best f as in RTA*). Best f is appropriate info. to be transferred *between* trials, second best is appropriate *within* a single trial

In RTA*, pessimistic heuristics better than optimistic (Sadikov, Bratko 2006)

- Traditionally, optimistic heuristics preferred in A* search (admissibility)
- Surprisingly, in RTA* pessimistic heuristics perform better than optimistic (solutions closer to optimal, less search, no pathology)