

HEURISTIC SEARCH

Ivan Bratko

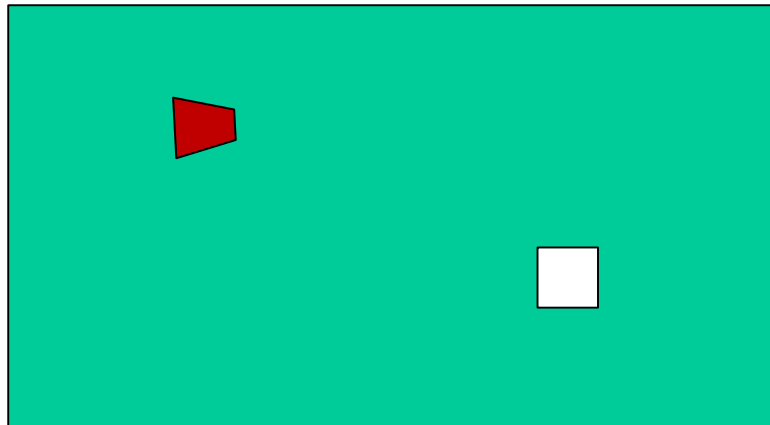
Faculty of Computer and Information Sc.

University of Ljubljana

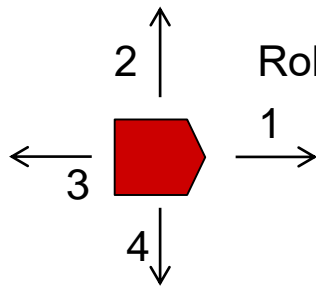
2021/22

PROBLEM OF SEARCH WITH UNINFORMED METHODS

- Example: A mobile robot searches for a path to a goal location in the plane
- Robot can only move to next cell in X-Y grid.
- How depth-first and breadth-first search work in this domain?



EXAMPLE

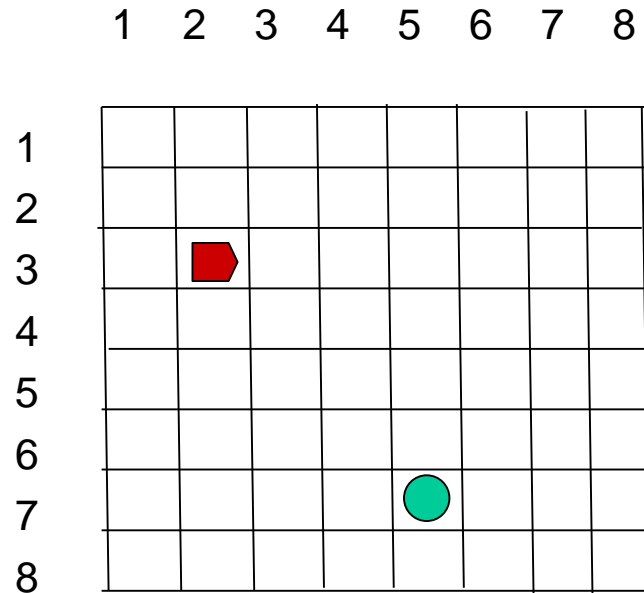


Robot's moves

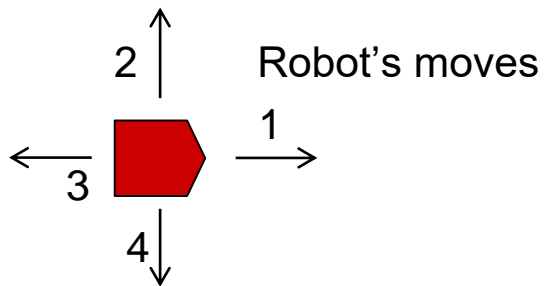
Depth-first search

Breadth-first search

Best-first (according to distance to goal)



EXAMPLE: DEPTH-FIRST SEARCH



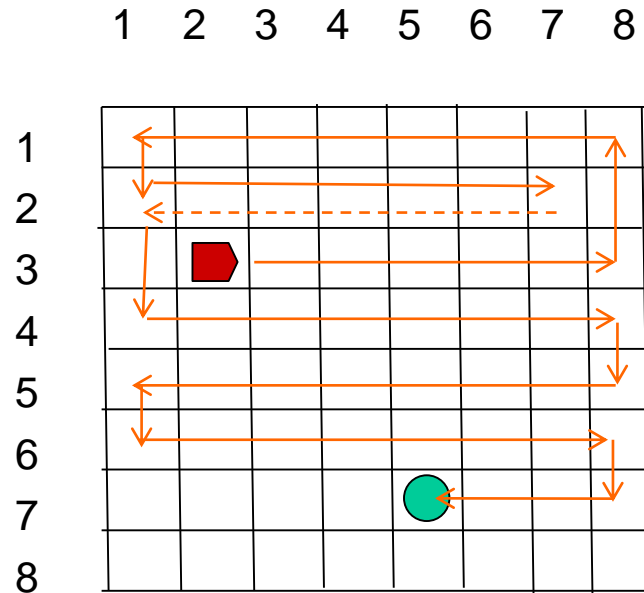
Depth-first

→ moving to depth

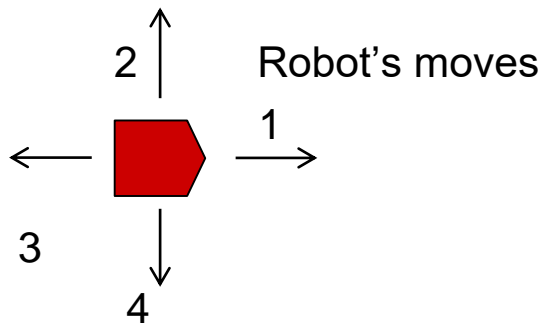
- - - - -> backtracking

Solution length = 45 (!)

Optimal sol. length = 7





EXAMPLE: BREADTH-FIRST SEARCH



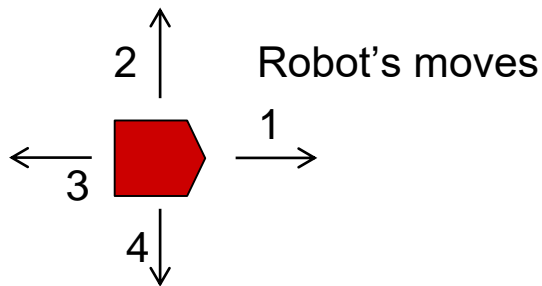
Breadth-first search:



Consecutive numbers of generated nodes 1, 2, 3, 4, ...

Length of solution found: 7

	1	2	3	4	5	6	7	8
1	17	8	15					
2	9	2	6	13				
3	3		1	5	12			
4	10	4	7	14				
5	18	11	16					
6		19						
7								
8								

EXAMPLE: BEST-FIRST SEARCH



	1	2	3	4	5	6	7
1							
2		2	6				
3	3		1	5			
4		4	7				
5							
6							
7							
8							

Nodes' evaluation: E.g. Manhattan distance to goal

Start node: (2,3)

Successor nodes' dist. to goal:

1. (3,3) 6
2. (2,2) 8
3. (1,3) 8
4. (2,4) 6

Best successors:

(3,3) and (2,4)

Choose (3,3) to expand.

Now best successors are:

"5" = (4,3) and "7" = (3,4).

Choose "5" to expand, etc.

Green numbers show order of generated nodes

Best-first search

- Best-first: most usual form of heuristic search
- Evaluate nodes generated during search
- Evaluation function
 $f: \text{Nodes} \rightarrow \mathbb{R}^+$
- 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, greedy algorithm
- Beam search
- IDA*
- RBFS
- ...

Heuristic evaluation in A*

The question: How to find successful f?

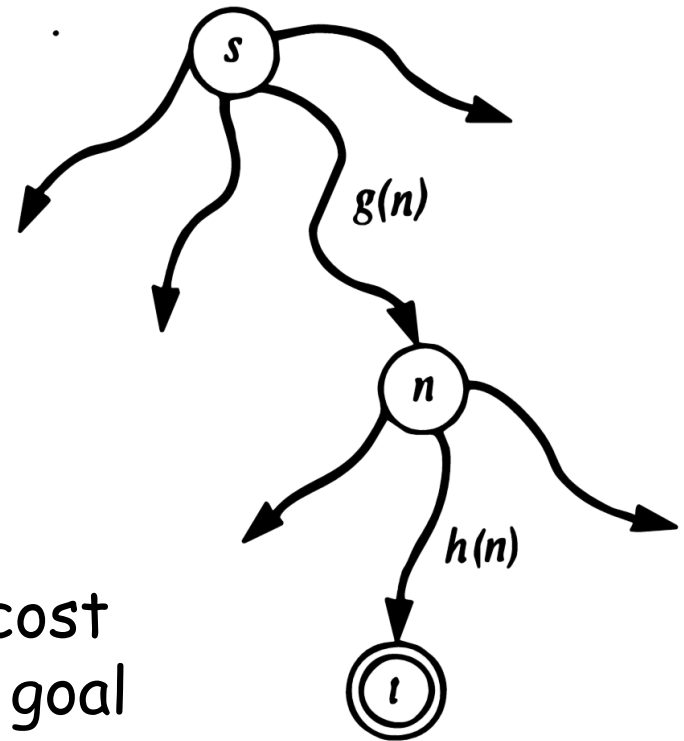
Algorithm A*:

$f(n)$ estimates cost of best solution through n

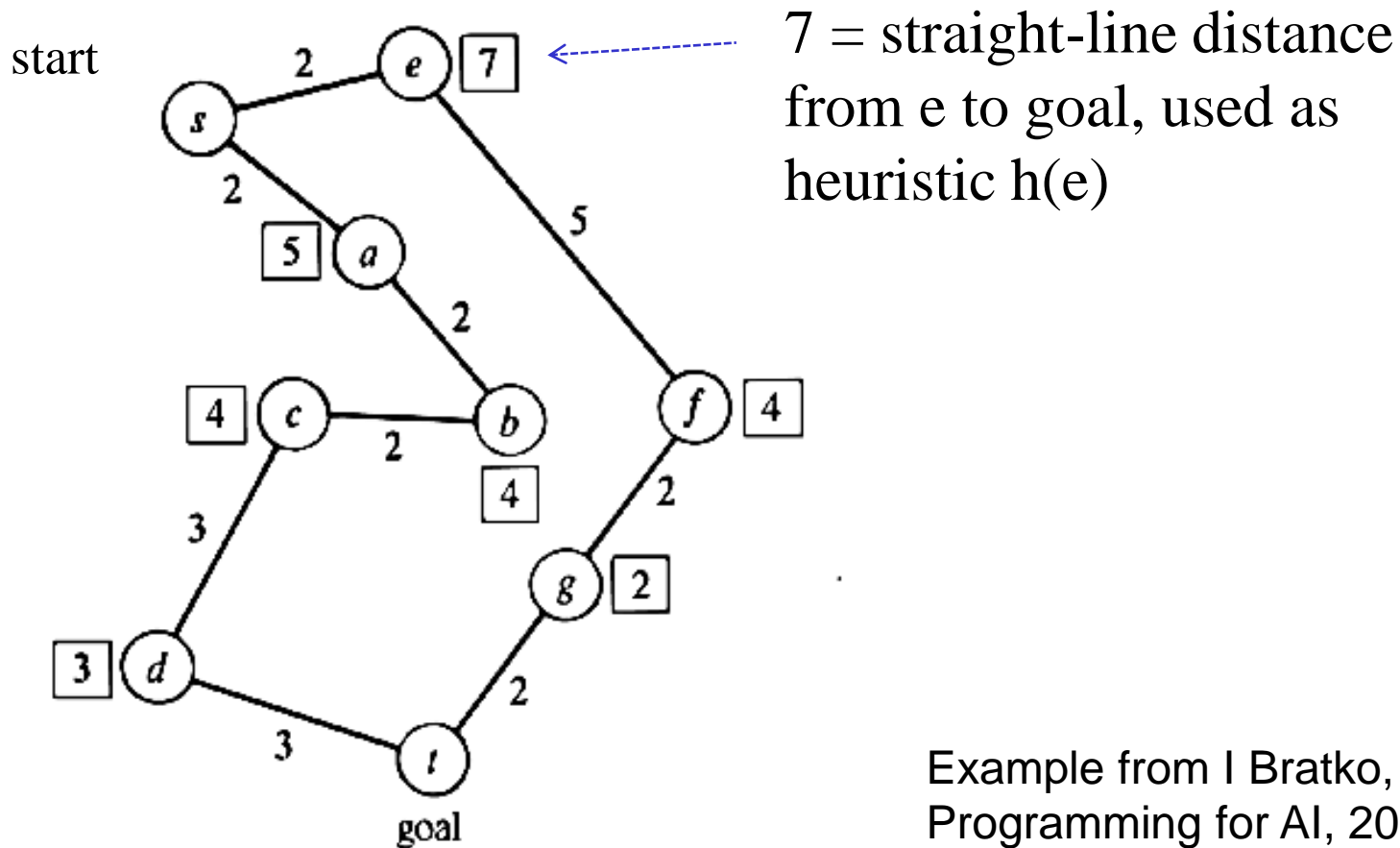
$$f(n) = g(n) + h(n)$$

known

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



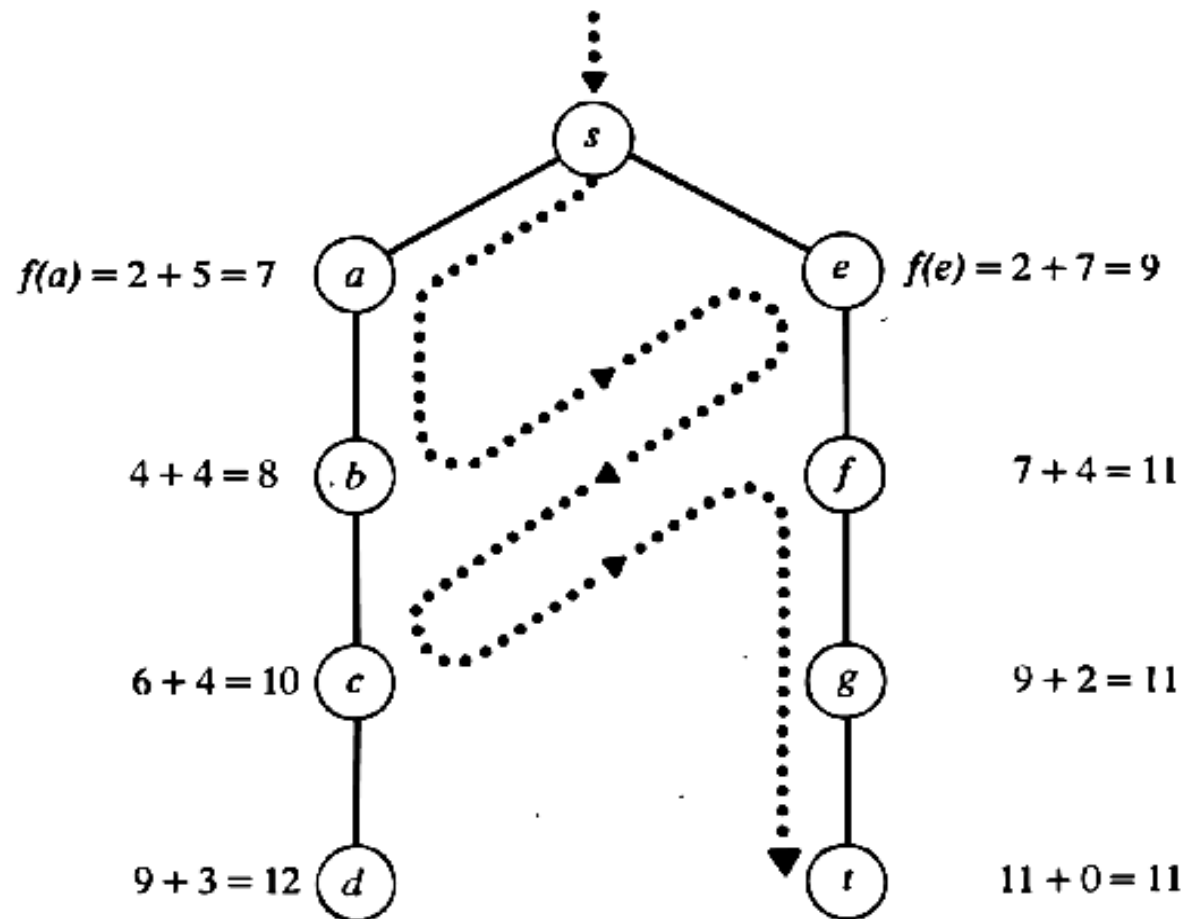
Route search in a map



Example from I Bratko, Prolog Programming for AI, 2011

Best-first search for shortest route

Dotted line shows order in which nodes are expanded



Admissibility of A*

The most famous theorem in all AI

- 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 $h(n) \leq h^*(n)$ 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 is trivially admissible!
- However, what is the drawback of $h = 0$?
- How well does it guide the search?
- Ideally: $h(n) = h^*(n)$ (but usually hard to find)
- 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
- $h = \text{total_dist} = \text{sum of Manhattan distances of tiles from their "home" squares}$

1	3	4
8	5	
7	6	2

$$\text{total_dist} = 0+3+1+1+2+0+0+0=7$$

- Is total_dist optimistic?

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

$$\text{sequence_score} = 1 + 2 + 0 + 2 + 2 + 0 + 0 + 0 = 7$$

$$h = \text{total_dist} + 3 * \text{sequence_score}$$

$$h = 7 + 3 * 7 = 28$$

Is this heuristic function optimistic?

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

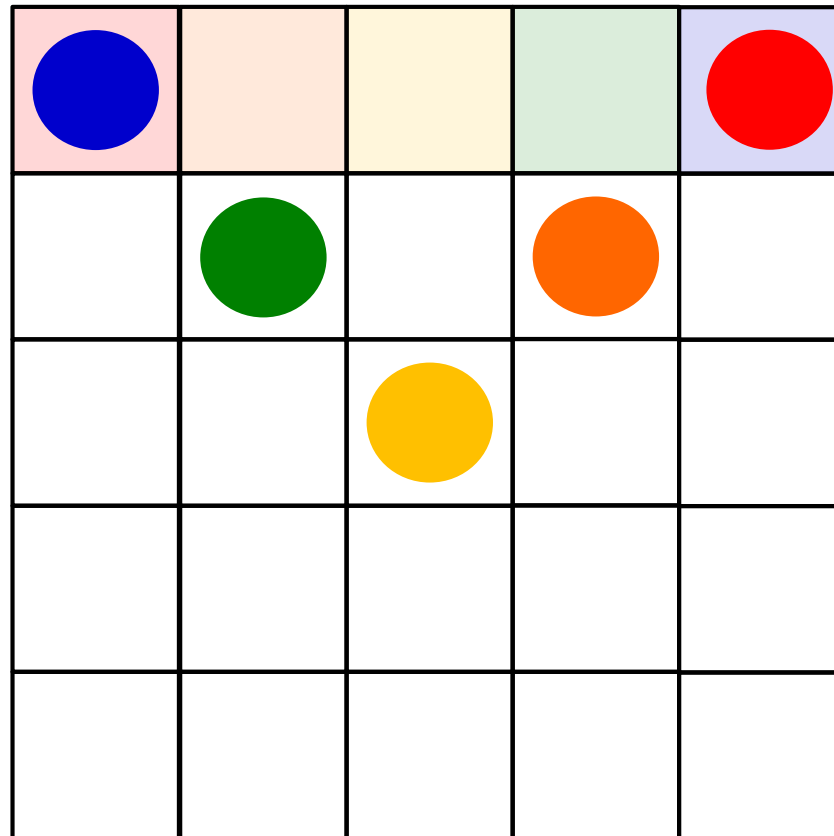
(c)

Puzzle c requires 18 steps

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

FIVE ROBOTS ON GRID 5 x 5

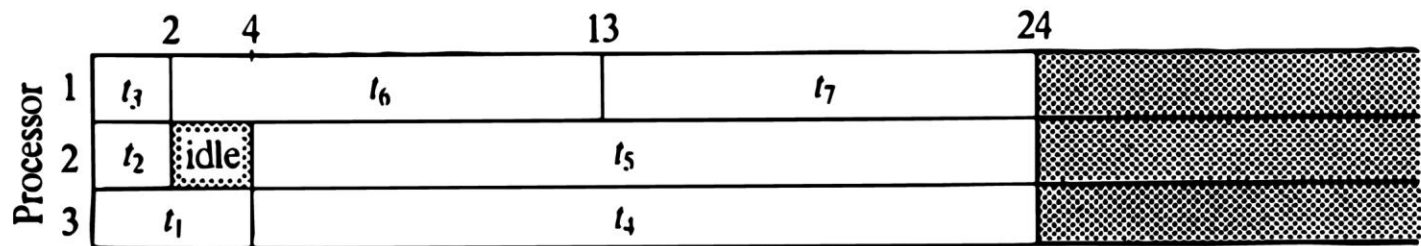
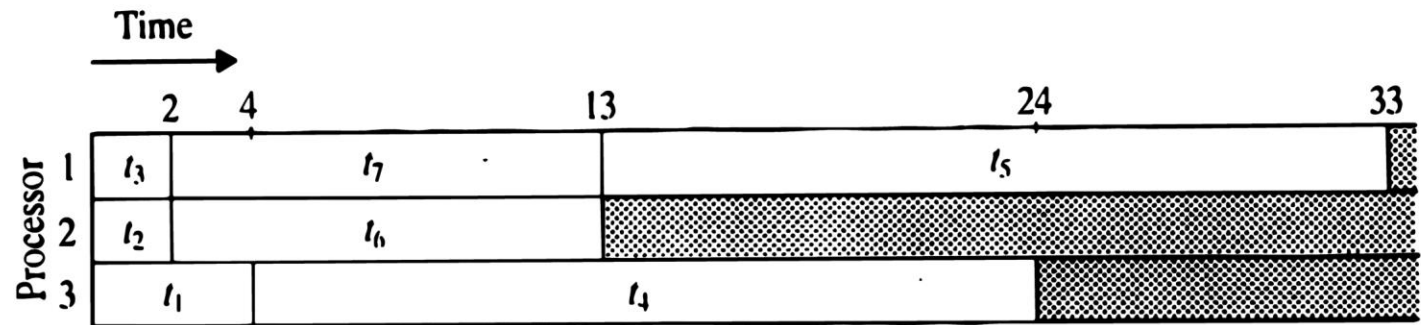
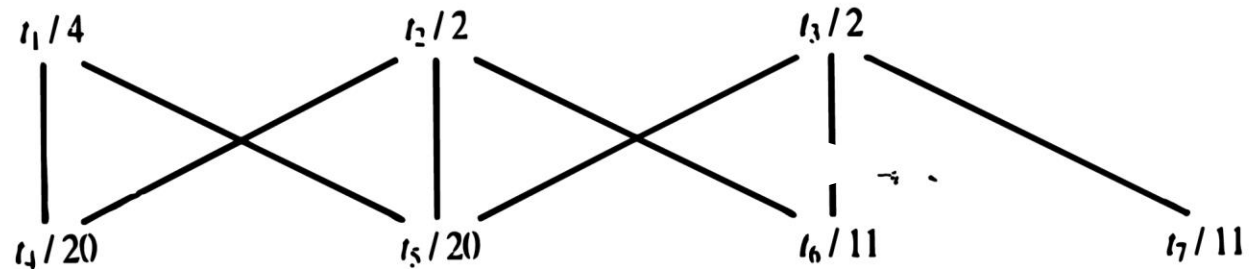
Robots creating a rainbow in top row



Possible heuristics for robots on grid

- $h1 = \max_R (\text{dist_from_goal}(R))$
- $h2 = \text{SUM}_R (\text{dist_from_goal}(R))$
- $h3 = K * \text{SUM}_R (\text{dist_from_goal}(R))$
 - K parameter
 - K may depend on density of robots
- $h4 = \max(h1, h3)$
- Experiments with h3 in 8-puzzle and robots on grid
- Project: Learning h-function

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

$Fin = \max F_J$

$Finall = (\text{SUM}_W(D_W) + \text{SUM}_J(F_J)) / m$

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

$h = \max(Finall - Fin, 0)$

- Is this heuristic admissible?

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 retained after expanding a node

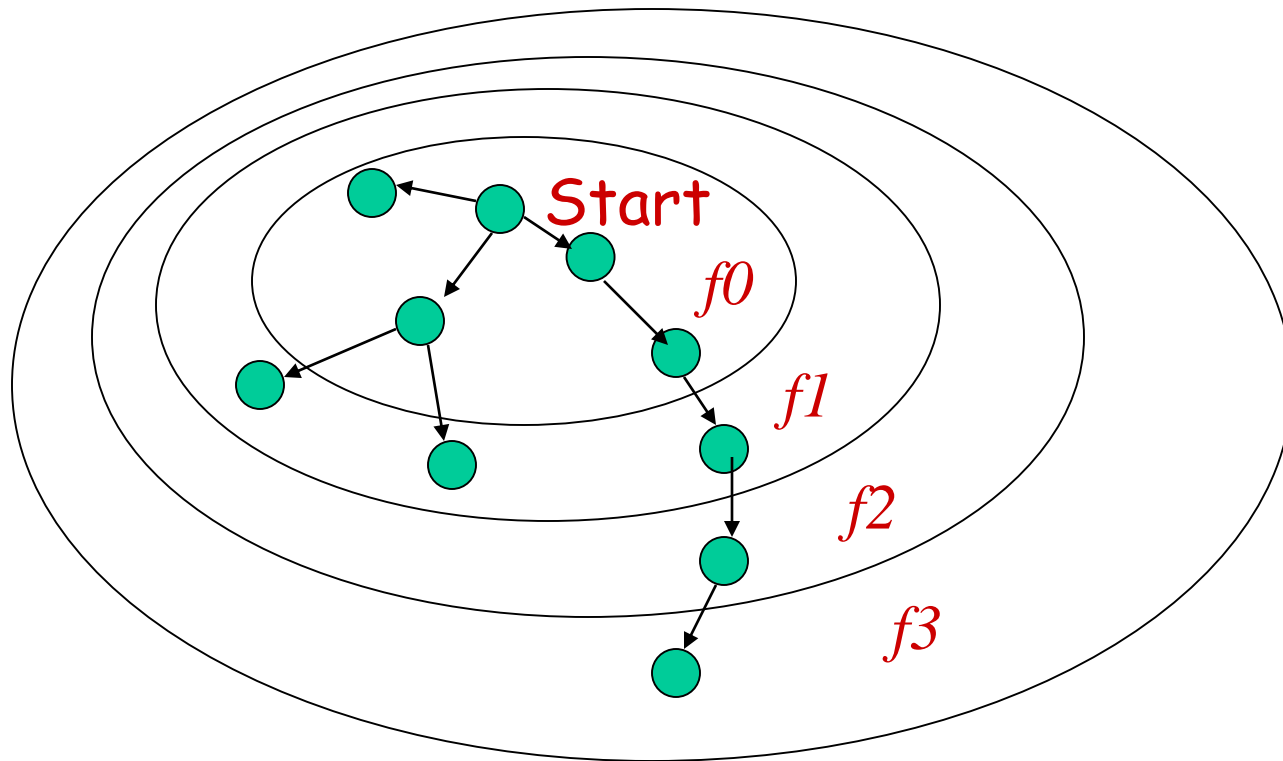
Space Saving Techniques for Best-First Search

- 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:
 $f_0, f_1, f_2, f_3, \dots$

IDA*

Bound := f(StartNode);

SolutionFound := false;

Repeat

perform depth-first search from StartNode, so that

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

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

then SolutionFound = true

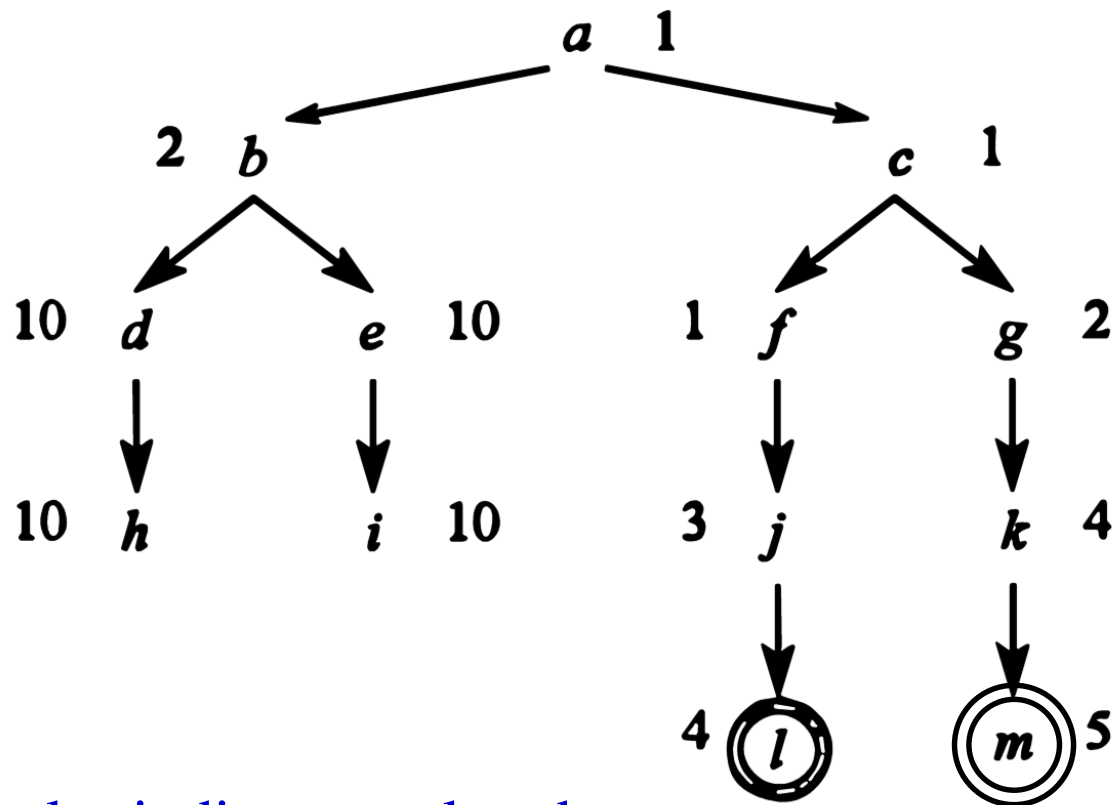
else

compute new bound as:

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

until solution found.

Problem. Trace the execution of A* when searching the tree bellow. How many nodes are generated by A* and IDA*? Count also re-generated nodes. How about memory used by A* and IDA*? The number next to a node is the node's f value.



Double circles indicate goal nodes

Answ:

$\#(A^*)=11$, $\#(IDA^*)=33$, $\#(RBFS) = 14(?)$

Memory: A^* : 11 states, IDA^* : 5 states

IDA* performance

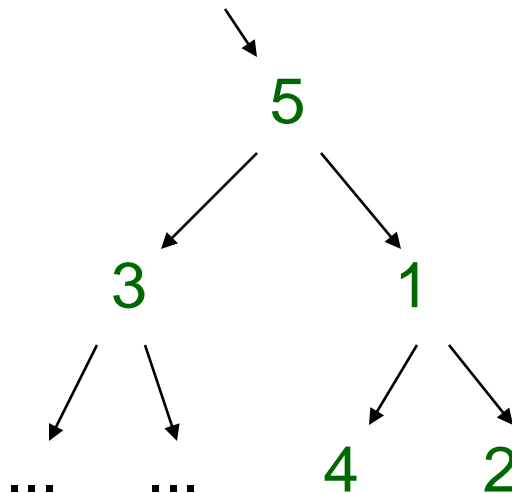
- Experimental results with 15-puzzle good
(h = Manhattan distance, many nodes same f)
- However: May become very inefficient when nodes do not tend to share f -values (e.g. small random perturbations of Manhattan distance)

COMPLEXITY OF IDA*

- Time: despite repetitions, normally same order as A^*
- Unfortunate case: when all f-values tend to be different from each other; then IDA* becomes very slow

IDA*: problem with non-monotonic f

- Function f is *monotonic* if:
for all nodes n_1, n_2 : if $s(n_1, n_2)$ then $f(n_1) \leq f(n_2)$
- Theorem: If f is monotonic then IDA* expands nodes in best-first order
- Example with non-monotonic f :

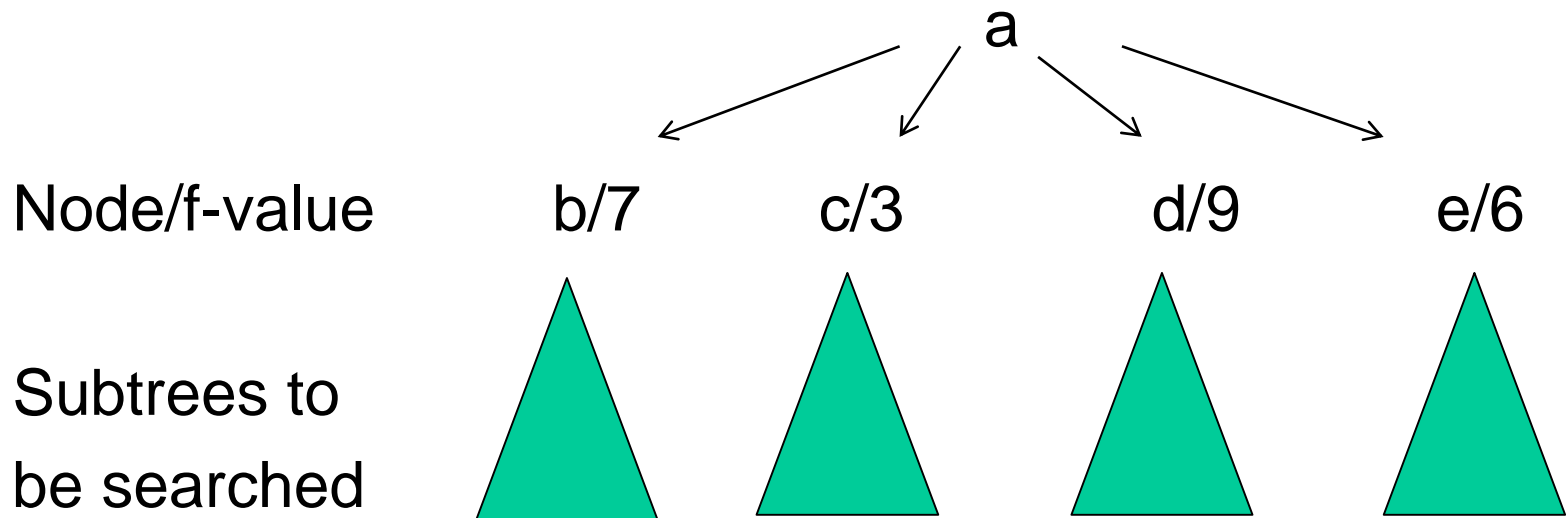


Green numbers denote f -values
 f -limit = 5,
“3” expanded before “1”, “2”

Linear Space Best-First Search (RBFS)

- Linear Space Best-First Search (linear in depth of search)
- RBFS (Recursive Best First Search), Korf 1993
- Similar to A* implementation in Bratko (1986; 2011), but saves space by iterative re-generation of parts of search tree

A* as competing search processes

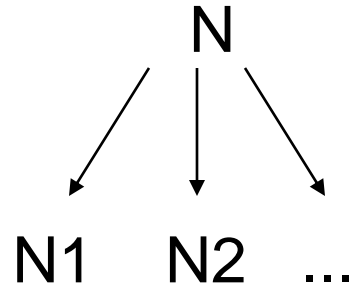


Idea: Choose the best candidate (c) and search its subtree until either (1) goal found, or (2) f-values become greater than nearest competitor (e, $f(e)=6$), then switch to next best candidate etc.

Insight of RBFS

- To save space, when switching to next best candidate, remove the just explored subtree from memory, but retain its backed-up f value

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

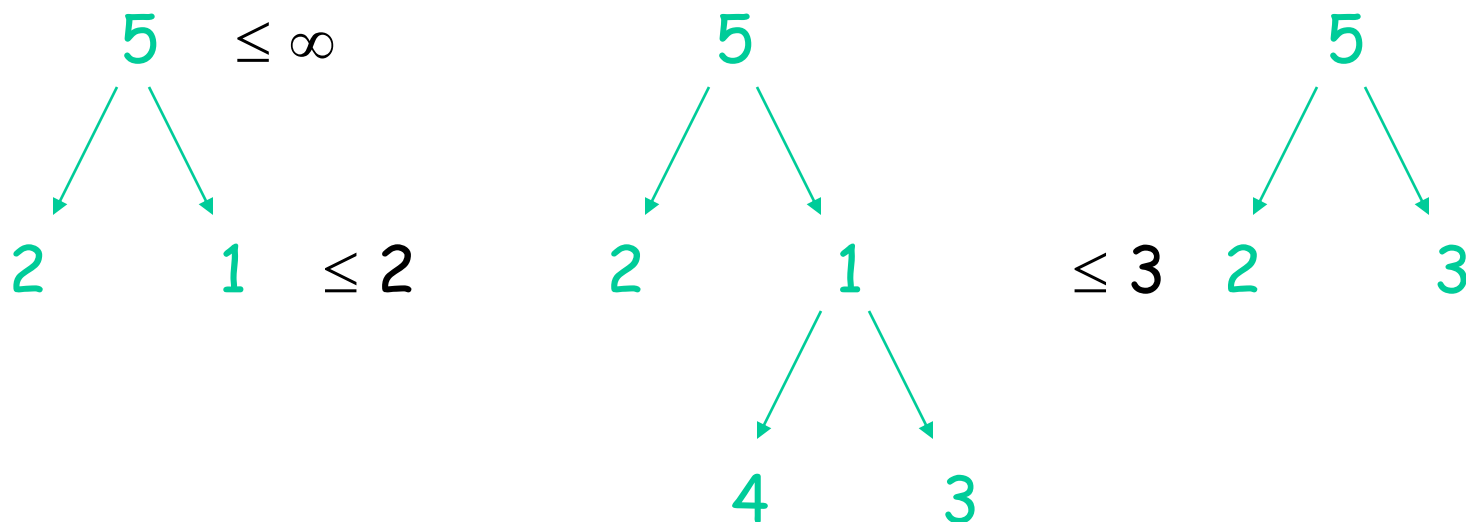
$F(N) = \min_i F(N_i)$ if N has been expanded

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



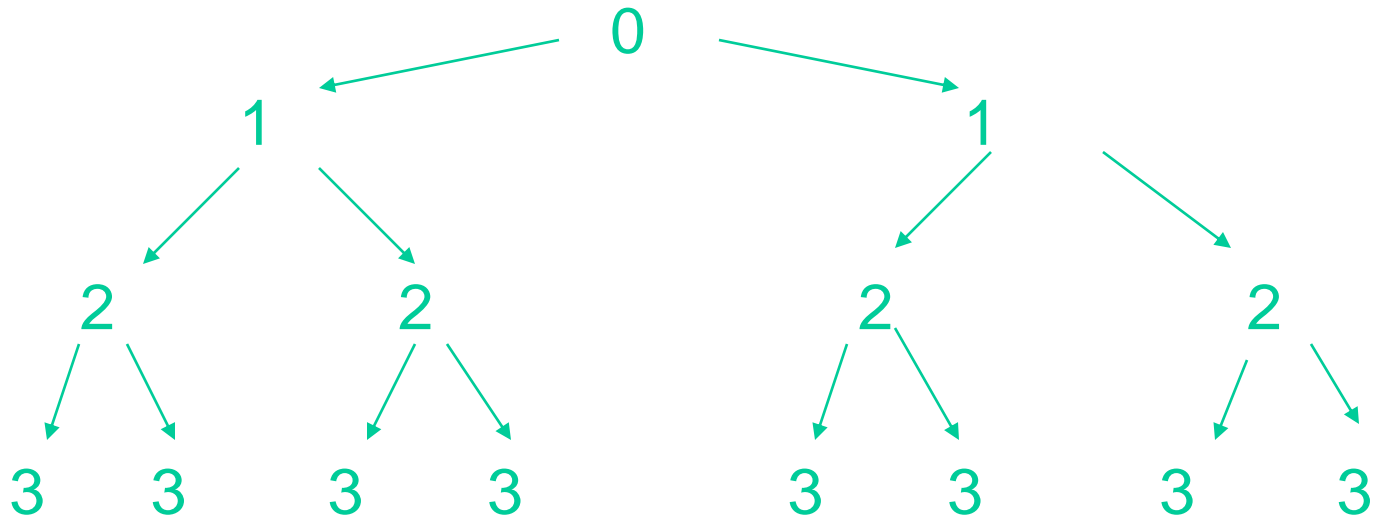
Green numbers are nodes' F-values, black are F-bounds

Note: Nodes are searched in best-first order (unlike IDA*)

SRBFS can be very inefficient

Example: search tree with

$f(\text{node}) = \text{depth of node}$, numbers below are f-values

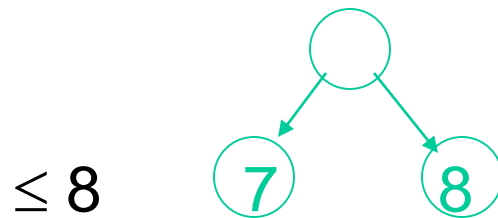


Parts of tree repeatedly re-explored;

f-bound increases in unnecessarily small steps

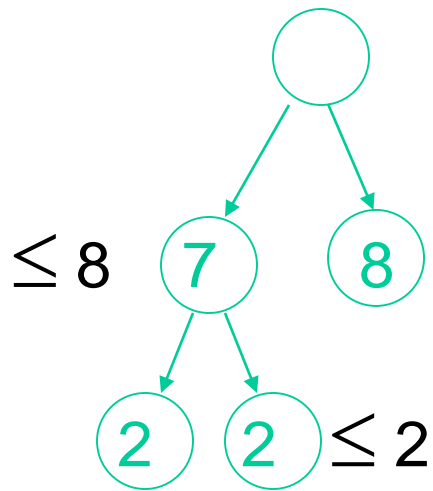
RBFS, improvement of SRBFS

- F-values not only backed-up from subtrees, but also *inherited* from parents
- Example: searching tree with $f = \text{depth}$
- At some point, F-values are:

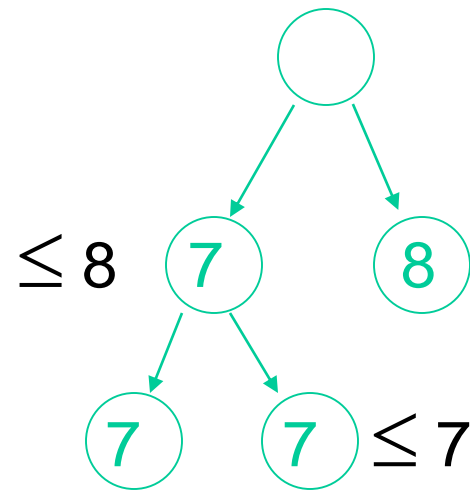


In RBFS, “7” is expanded, and F-value inherited:

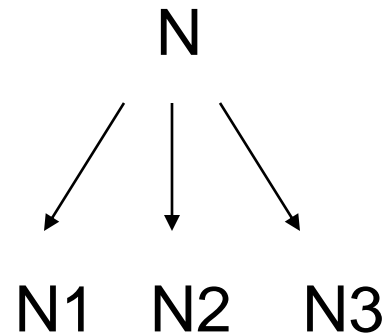
SRBFS



RBFS

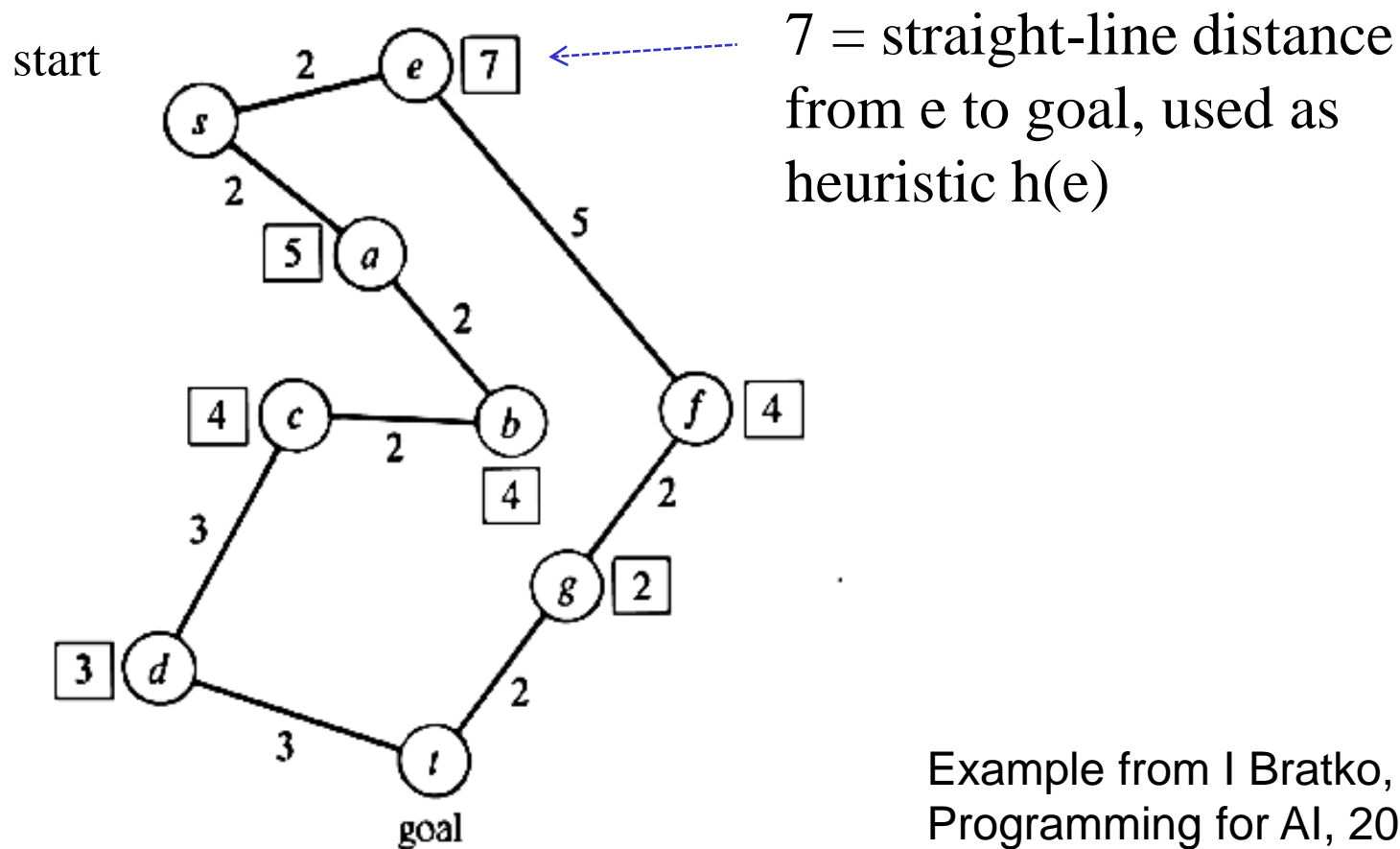


F-values of nodes



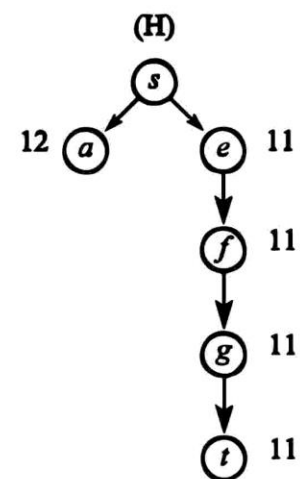
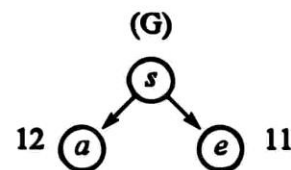
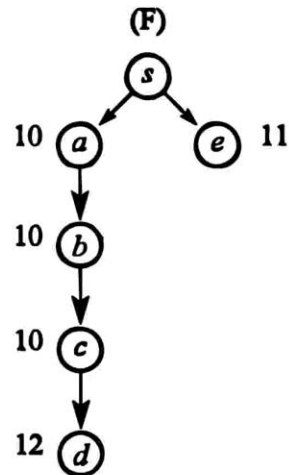
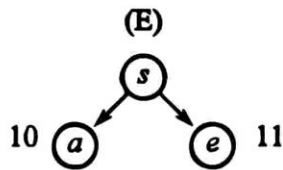
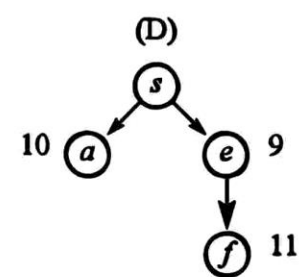
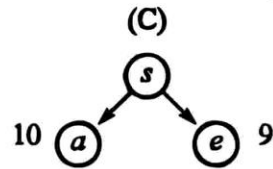
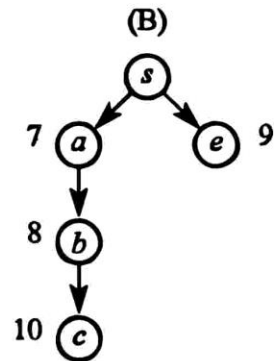
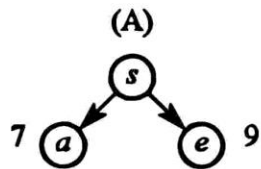
- To save space RBFS often removes already generated nodes
- But: If $N1, N2, \dots$ are deleted, essential info. is retained as $F(N)$
- Note: If $f(N) < F(N)$ then (part of) N 's subtree must have been searched and $F(N_i) \geq F(N)$ for all i

Example: Route search in a map with RBFS



Example from I Bratko, Prolog Programming for AI, 2011

Searching the map with RBFS



Example from I Bratko, Prolog
Programming for AI, 2011

Algorithm RBFS

RBFS(N, F(N), Bound)

Explore tree rooted in node N until either:

- (1) F-values exceed Bound, or
- (2) goal node with $F \leq \text{Bound}$ encountered, or
- (3) tree ends

Depending on these outcomes, return either:

- (1) New, updated F(N)
- (2) Stop search, solution found
- (3) New, updated $F(N) = \infty$ (Never return here again)

Algorithm RBFS

RBFS(N, F(N), Bound)

if $F(N) > \text{Bound}$ then return $F(N)$;

if goal(N) then exit search;

if N has no children then return ∞ ;

for each child N_i of N do

 if $f(N) < F(N)$ then $F(N_i) := \max(F(N), f(N_i))$

 else $F(N_i) := f(N_i)$;

Sort N_i in increasing order of $F(N_i)$;

if only one child then $F(N_2) := \infty$;

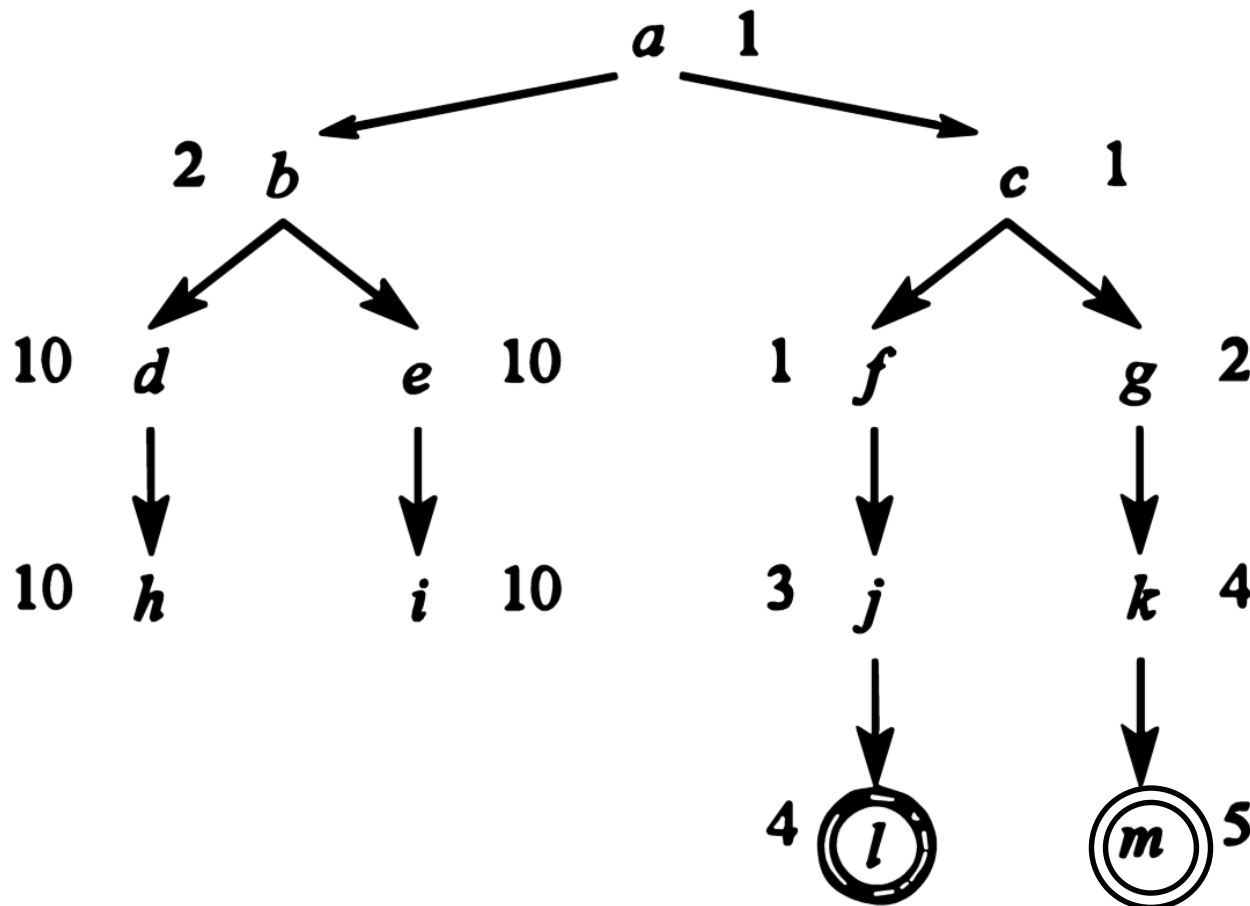
while $F(N_1) \leq \text{Bound}$ and $F(N_1) < \infty$ do

$F(N_1) := \text{RBFS}(N_1, F(N_1), \min(\text{Bound}, F(N_2)))$

 insert N_1 in sorted order

return $F(N_1)$.

Problem. Trace RBFS on the tree bellow. How many nodes are generated by RBFS? Count also re-generated nodes. Compare this with A* and IDA*.



Properties of RBFS

- RBFS(Start, $f(\text{Start})$, ∞) 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 is optimistic: $h(n) \leq h^*(n)$ for all n
- Sometimes in literature: A^* defined with admissible h ; then A^* is admissible by definition (but under this terminology, we have no name for A^* with non-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 n_1, n_2 : $s(n_1, n_2) \Rightarrow f(n_1) \leq f(n_2)$

Best-first order

- A* and RBFS respect best-first order
- IDA* respects best-first order if f is monotonic

QUESTIONS

The following questions are about A^* , IDA* and RBFS:

- (a) When do we say that f function is monotonic? Give an example when f is not monotonic.
- (b) How is evaluation function composed in A^* ?
- (c) Give the well-known sufficient condition that guarantees admissibility of A^* .
- (d) Does the condition from the admissibility theorem also guarantee that the f -function is monotonic? Explain your answer.
- (e) The following statement is valid for which of the algorithms A^* , IDA* and RBFS : if function f is not monotonic then it is not guaranteed that the algorithm expands generated nodes in best-first order.
- (f) How would you rank algorithms A^* , IDA* and RBFS from the most to the least efficient one with respect to:
 - space efficiency
 - time efficiency

Of course relative success of the algorithms depends on the concrete search problem, therefore your answers are expected for the “average” case.

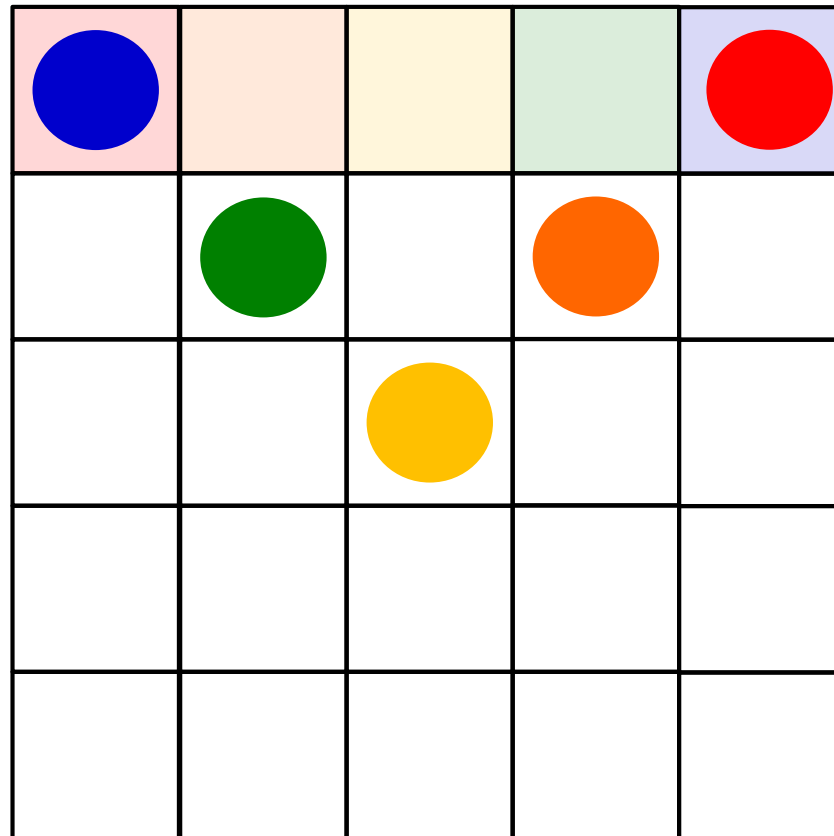
- (g) Do monotonic evaluation functions also necessarily satisfy the condition from the admissibility theorem?
- (h) Does admissibility theorem also apply to IDA*? Yes

EXPERIMENTS WITH HEURISTIC FUNCTIONS FOR ROBOTS ON GRID

KIND OF “BEIJING POST OFFICE”

FIVE ROBOTS ON GRID 5 x 5

The robots want to form a rainbow in top row



GRID 5x5, 5 ROBOTS

Goal: all robots in top row ordered a, b, c, d, e

A start state

e	-	-	-	a
-	d	-	b	-
-	-	c	-	-
-	-	-	-	-
-	-	-	-	-

A* ON RAINBOW PROBLEM

$$h(\text{node}) = k * \text{total_dist}(\text{node})$$

k	#nodes generated	Time [sec]	Solution length
---	---------------------	---------------	-----------------

0.2 Resource error, insufficient memory, program stopped after about 110 sec

0.3 Resource error, "-", program stopped after about 283 sec, generated 1,269,545 states

0.4 676,416 35.406 6 steps

0.5 330,822 16.107 6 steps Effective branching = 8.32

0.6 22,661 1.015 8 steps Effective branching = 5.32

0.7 58,284 2.422 9

1.0 56,748 2.250 10

1.5 14,093 0.547 14 Effective branching = 1.98

2.0 4,971 0.219 15 Effective branching = 1.76

RBFS ON RAINBOW PROBLEM

k	#nodes generated	Time [sec]	Solution length
---	---------------------	---------------	-----------------

0.2	~20 million	~23 minutes	Program stopped manually, no solution found in 23min
-----	-------------	-------------	--

0.3	18,540,928	~21 min	6 steps Note: This was better than A*
-----	------------	---------	---------------------------------------

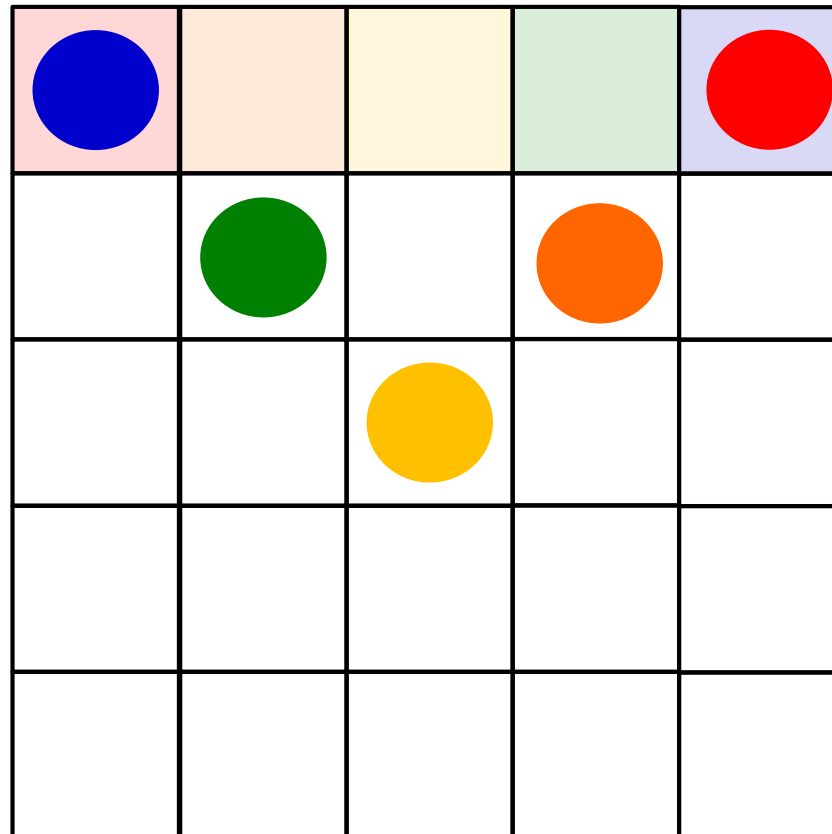
#nodes generated by A*

0.4	1,648,776	114 sec	6	676,416
0.5	831,674	56 sec	6	330,822
0.6	29,429	1.469 sec	8	22,661
0.7	93,743	5.301 sec	9	58,284
1.0	128,301	7.797	10	56,748
1.5	28,823	1.328	14	14,093
2.0	6,778	0.328	15	4,971

FIVE ROBOTS ON GRID 5 x 5

The robots want to form a rainbow in top row

Some moving patterns are formed – one way streets?



A* on 8-puzzle (8 robots on grid 3x3)

$$h = k * \text{TotalDist}$$

- $h = k * \text{totaldist}$
- What are sensible values of parameter k ?
- Intuitively, how will performance change with changing k ?

START	GOAL	Optimal 18 steps
216	123	
4-8	8-4	
753	765	

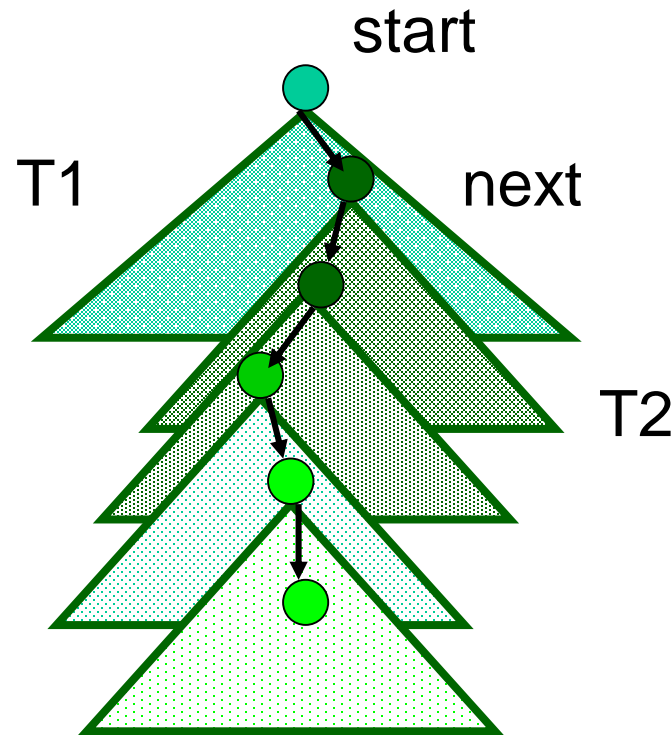
K	# Nodes generated	Time [msec]	Solution length	
0.8	43,177	11609	18	
0.9	17,833	4733	18	
1.0	10,102	2719	18	Here effective branching = 1.67
1.1	4,465	1141	18	
1.2	4,447	1141	18	
1.3	3,622	937	18	
1.35	3,304	827	18	
1.4	3,144	780	18	
1.42	3,081	796	18	
1.45	3,078	781	18	Here effective branching = 1.56
1.47	3,078	781	18	
1.48	3,078	781	18	
1.49	3,078	781	18	
1.5	3,173	813	18	
1.6	4,068	1016	18	
1.7	7,406	1970	18	
1.8	8,700	2269	18	
1.9	15,666	4157	18	
2.0	15,705	4156	18	
2.1	82,135	21693	18	
2.5	Execution aborted (run out of memory)			

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

By splitting the task into planning and execution, RTA* can solve much larger problems than A*, although suboptimally. First, search lookahead tree T1 from start; this gives most promising successor next state of start. Then execute move start-next. Then search lookahead tree T2 from next, etc.

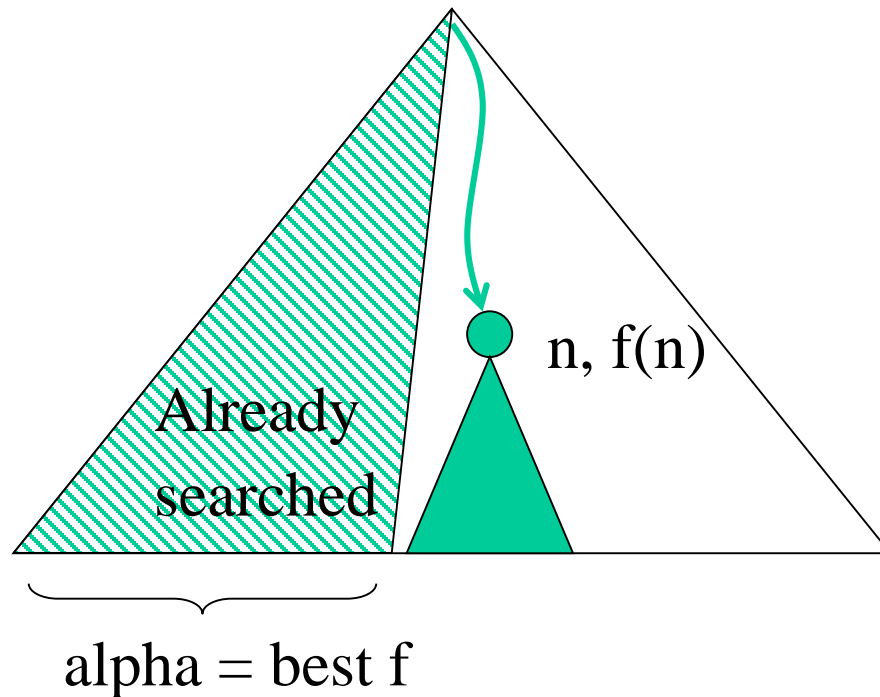


RTA* and alpha-pruning

- Lookahead is done in depth-first manner
- 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 lookahead 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 $f(n) \geq \alpha$
- If f is monotonic then all descendants of n have $f \geq \alpha$



RTA*, details

- Problem solving = planning stage + execution stage
- In RTA*, planning and execution are interleaved
- Distinguished states: start state, current state
- Current state is understood as actual physical state of robot reached after 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 $s := \text{start_state}$
- While goal not found:
 - Plan: evaluate successors of s by fixed depth lookahead;
 - $\text{best_s} :=$ successor with min. backed-up f
 - $\text{second_best_f} := f$ of second best successor
 - Store s among “visited nodes” and store $f(s) := \text{second_best_f}$
 - Execute: current state $s := \text{best_s}$

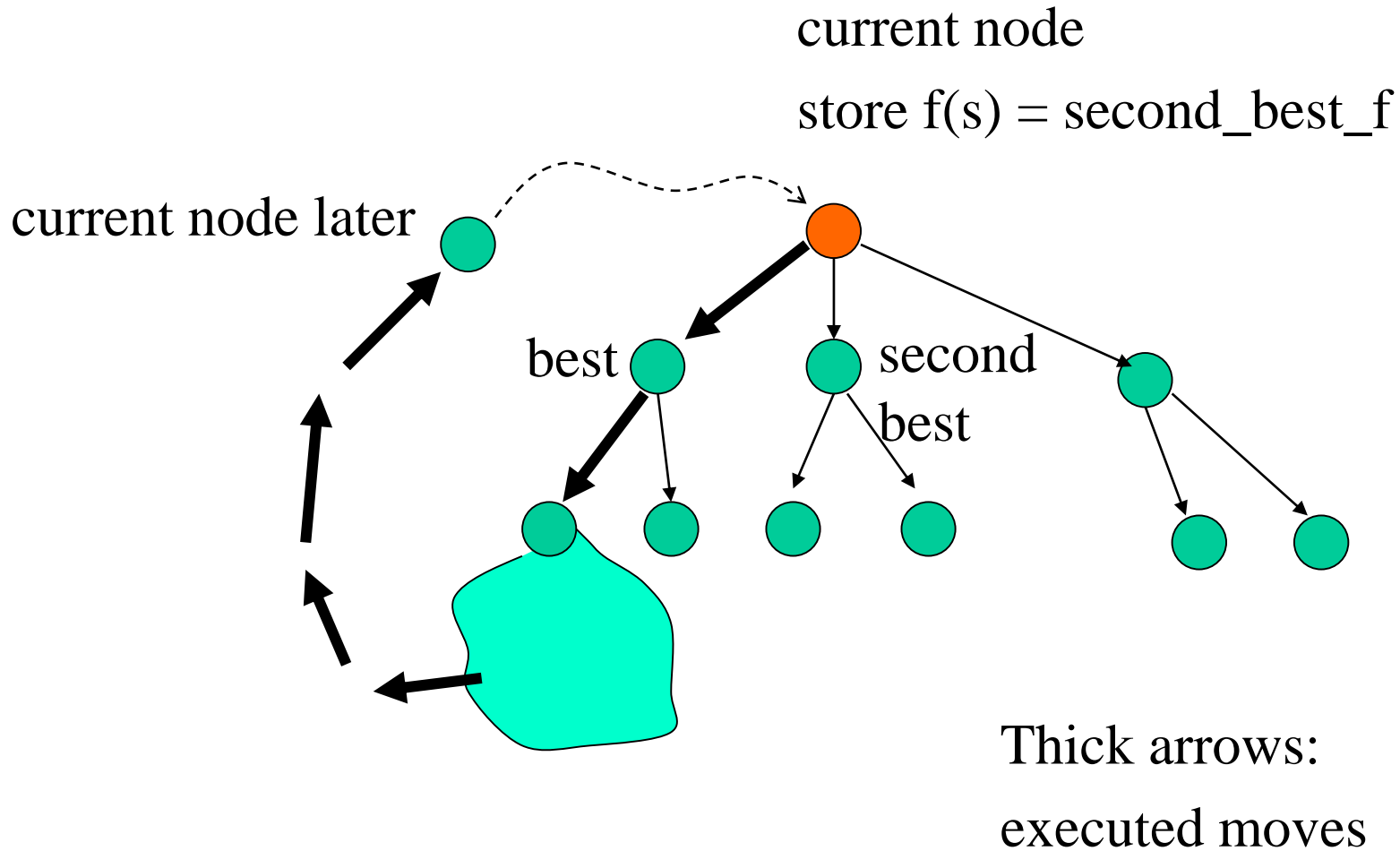
RTA*, visited nodes

- 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:

$$f(s) := \text{second_best_f}$$

If best_s subsequently turns out as bad, problem solver may at some later point return to s and this time consider s 's second best successor; otherwise, if second_best_f was not stored, problem-solver would find best_s as best again and repeat the same loop

Why store second best f, illustration



Ideas behind storing f-values of current state

- If current state S is encountered in a later lookahead, we want to heuristically evaluate S based on previous lookahead, not by static $h(S)$.
- The effect of this is as if lookahead depth was increased – hopefully more reliable evaluation
- Using `second_best_f` is more realistic than `best_f` which has led to a cycle

RTA* lookahead

For node n encountered by lookahead:

- if $\text{goal}(n)$ then return $h(n) = 0$,
don't search beyond n
- if $\text{visited}(n)$ then return $h(n) = \text{stored } f(n)$,
don't search beyond n
- if n at lookahead horizon 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

EXPERIMENTS 8-PUZZLE A*

- A*
 - Eight puzzle 30 steps, using Manh dist
 - Solution length = 30, Time = 204 msec
-
- A*
 - Eight puzzle 30 steps, Manh dist + 3 * sequence score
 - Solution length = 36, Time = 47 msec

SAME 8-PUZZLE RTA*

RTA*

Eight puzzle 30 steps Manh dist

Lookahead	Solution_length	Time [msec]
1	66	16
2	62	15
3	356	562
4	418	1985
5	120	515
6	66	500
7	46	750
8	136	12078
9	98	20047
10	44	14469
11	32	24344

Note: Search pathology! (cf lookahead depths 2 and 3)

SAME 8-PUZZLE RTA*

RTA*

Eight puzzle 30 steps Manh dist + 3 * sequence score

Look. depth	Solution_length	Time
1	108	16
2	44	15
3	50	31
4	44	47
5	42	110
6	42	265
7	94	1750
8	34	1563
9	44	5547
10	44	14657
11	56	50906

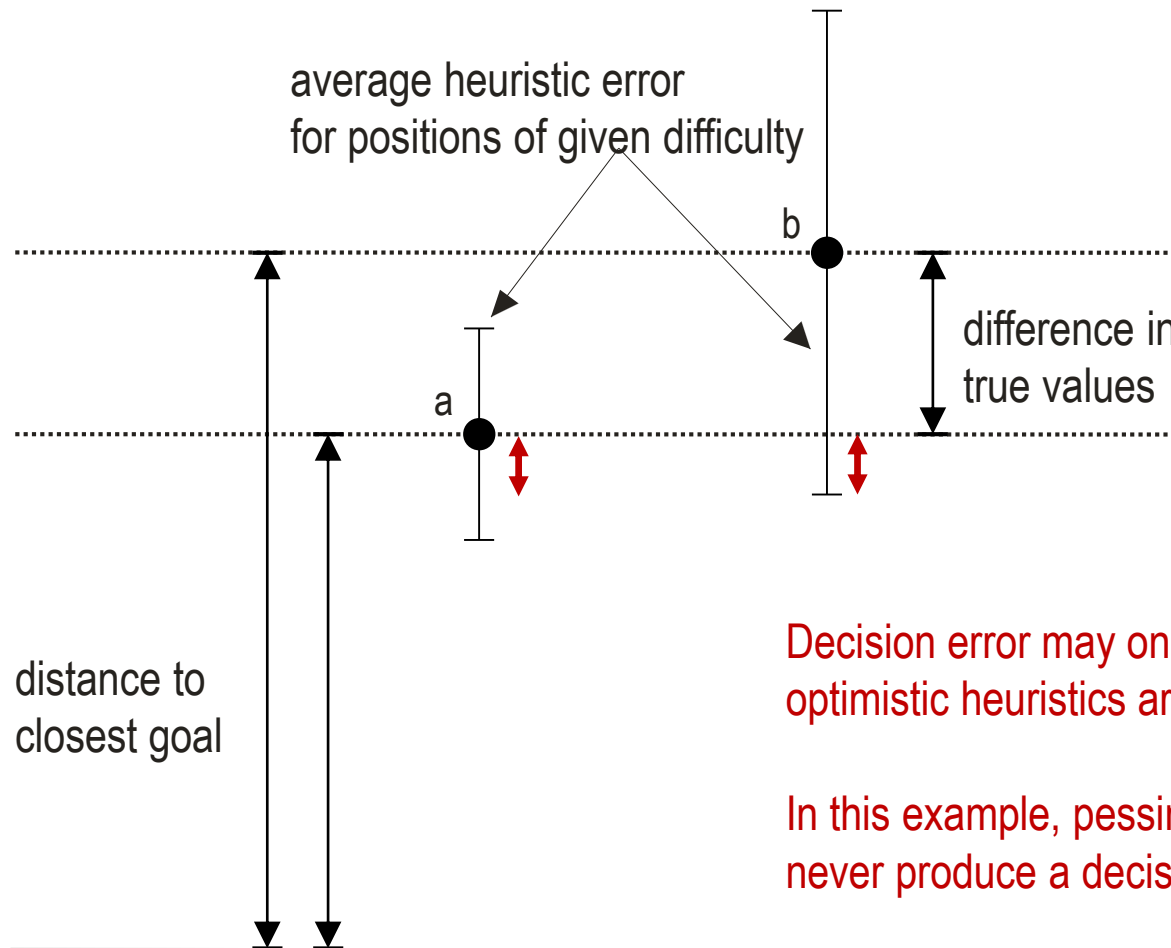
LRTA*, Learning RTA*

- Useful when successively solving multiple problem instances with the same goal, or the same problem in multiple trials to improve the solution
- Trial = Solving one problem instance
- Save table of visited nodes with their backed-up f 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

- Although optimistic heuristics traditionally preferred in A* search
- Surprisingly, in RTA* and LRTA* pessimistic heuristics perform better than optimistic (Sadikov, Bratko 2006)
- Solutions closer to optimal, less search, no “pathology”

Who is more likely to commit a decision error?



A natural assumption

harder problems
give rise to larger heuristic errors

