

Artificial Intelligence¹

Roman Barták

Department of Theoretical Computer Science and Mathematical Logic

Problem Solving: Informed (Heuristic) Search

Information in search

- Recall that we are looking for (the shortest) path from the initial state to some goal state.
- Which information can help the search algorithm?
 - For example, the length of path to some goal state.
 - However such information is usually not available (if it is available then we do not need to do search). Usually some **evaluation function $f(n)$** is used to evaluate „quality” of node n based on the length of path to the goal.
- best-first search**
 - The node with the smallest value of $f(n)$ is used for expansion.
- There are search algorithms with different views of $f(n)$. Usually the part of $f(n)$ is a **heuristic function $h(n)$** estimating the length of the shortest (cheapest) path to the goal state..
 - Heuristic functions are the most common form of additional information given to search algorithms
 - We will assume that **$h(n) = 0 \Leftrightarrow n$ is goal.**



- Uninformed (blind)** search algorithms can find an (optimal) solution to the problem, but they are usually not very efficient.
- Informed (heuristic)** search algorithms can find solutions more efficiently thanks to exploiting problem-specific knowledge.
 - How to use heuristics in search?**
 - BFS, A*, IDA*, RBFS, SMA*
 - How to build heuristics?**
 - relaxation, pattern databases

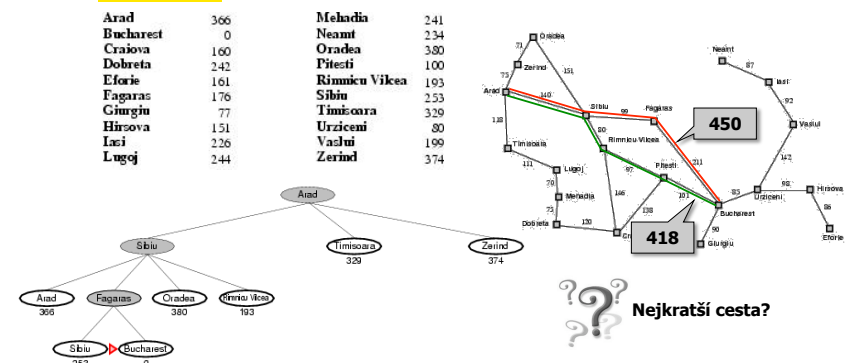


Greedy best-first search

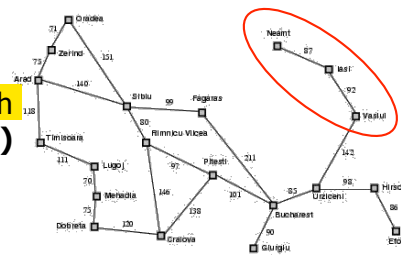
- Let us try to expand first the node that is closest to some goal state, i.e. $f(n) = h(n)$.
 - greedy best-first search algorithm**

Example (path Arad \rightarrow Bucharest):

- We have a table of direct distances from any city to Bucharest.
- Note: this information was not part of the original problem formulation!**

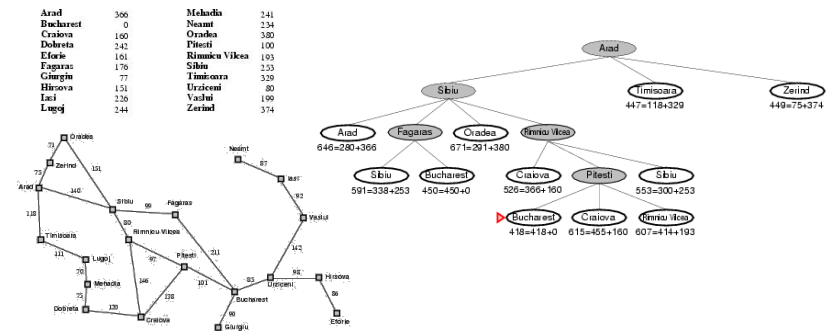


- We already know that the greedy algorithm **may not find the optimal path.**
 - Can we at least always find some path?**
 - If we expand first the node with the smallest cost then the algorithm **may not find any solution.**
- Example: path Iasi → Fagaras**
- Go to Neamt, then back to Iasi, Neamt, ...
 - We need to detect repeated visits in cities!



- Time complexity $O(b^m)$,** where m is the maximal depth
- Memory complexity $O(b^m)$**
- A good heuristic function can significantly decrease the practical complexity.

- Let us now try to use **$f(n) = g(n) + h(n)$**
 - Recall that $g(n)$ is the cost of path from root to n
 - probably the most popular heuristic search algorithm
 - $f(n)$ represents the cost of path through n
 - the algorithm does not extend already long paths



What about completeness and optimality of A*?

First a few definitions:

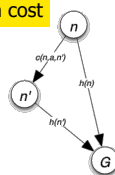
- admissible heuristic $h(n)$**
 - $h(n) \leq$ „the cost of the cheapest path from n to goal “
 - an optimistic view (the algorithm assumes a better cost than the real one)
 - function $f(n)$ in A* is a lower estimate of the cost of path through n
- monotonous (consistent) heuristic $h(n)$**
 - let n' be a successor of n via action a and $c(n,a,n')$ be the transition cost
 - $h(n) \leq c(n,a,n') + h(n')$
 - this is a form of triangle inequality

Monotonous heuristic is admissible.

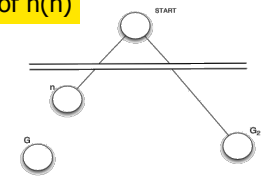
let n_1, n_2, \dots, n_k be the optimal path from n_1 to goal n_k , then
 $h(n_i) - h(n_{i+1}) \leq c(n_i, a_i, n_{i+1})$, via monotony
 $h(n_1) \leq \sum_{i=1, \dots, k-1} c(n_i, a_i, n_{i+1})$, after „sum“

For a monotonous heuristic the values of $f(n)$ are not decreasing over any path.

Let n' be a successor of n , i.e. $g(n') = g(n) + c(n,a,n')$, then
 $f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n)$

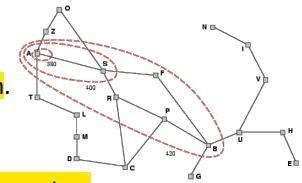


- If $h(n)$ is an admissible heuristic then the algorithm A* in TREE-SEARCH is optimal.**
 - in other words – the first expanded goal is optimal
 - Let G_2 be sub-optimal goal from the fringe and C^* be the optimal cost
 - $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$, because $h(G_2) = 0$
 - Let n be a node from the fringe and being on the optimal path
 - $f(n) = g(n) + h(n) \leq C^*$, via admissibility of $h(n)$
 - together
 - $f(n) \leq C^* < f(G_2)$,
- i.e., the algorithm must expand n before G_2 and this way it finds the optimal path.



- If $h(n)$ is a monotonous heuristic then the algorithm A* in GRAPH-SEARCH is optimal.
 - Possible problem: reaching the same state for the second time using a better path – classical GRAPH-SEARCH ignores this second path!
 - A possible solution: selection of better from both paths leading to a close node (extra bookkeeping) or using monotonous heuristic.
 - for monotonous heuristics, the values of $f(n)$ are not decreasing over any path
 - A* selects for expansion the node with the smallest value of $f(n)$, i.e., the values $f(m)$ of other open nodes m are not smaller, i.e., among all "open" paths to n there cannot be a shorter path than the path just selected (no path can shorten)
 - hence, the first closed goal node is optimal

- For non-decreasing function $f(n)$ we can draw **contours** in the state graph (the nodes inside a given contour have f -costs less than or equal to the contour value.
 - for $h(n) = 0$ we obtain circles around the start
 - for more accurate $h(n)$ we use, the bands will stretch toward the goal state and become more narrowly focused around the optimal path.
- A* expands all nodes such that $f(n) < C^*$ on the contour
- A* can expand some nodes such that $f(n) = C^*$
- the nodes n such that $f(n) > C^*$ are never expanded
- the algorithm A* is **optimality efficient** for any given consistent heuristic

**Time complexity:**

- A* can expand an exponential number of nodes
 - this can be avoided if $|h(n) - h^*(n)| \leq O(\log h^*(n))$, where $h^*(n)$ is the cost of optimal path from n to goal

Space complexity:

- A* keeps in memory all expanded nodes

A* usually runs out of space long before it runs out of time

- A simple way to decrease memory consumption is iterative deepening.

Algorithm IDA*

```

function IDA*(problem) returns a solution sequence
inputs: problem, a problem
static: f-limit, the current f-COST limit
root, a node

root ← MAKE-NODE(INITIAL-STATE[problem])
f-limit ← f-COST(root)
loop do
  solution, f-limit ← DFS-CONTOUR(root, f-limit)
  if solution is non-null then return solution
  if f-limit = ∞ then return failure; end

function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f-COST limit
inputs: node, a node
      f-limit, the current f-COST limit
static: next-f, the f-COST limit for the next contour, initially ∞

if f-COST[node] > f-limit then return null, f-COST[node]
if GOAL-TEST[problem](STATE[node]) then return node, f-limit
for each node s in SUCCESSORS(node) do
  solution, new-f ← DFS-CONTOUR(s, f-limit)
  if solution is non-null then return solution, f-limit
  next-f ← MIN(next-f, new-f); end
return null, next-f
  
```

- the search limit is defined using the cost $f(n)$ instead of depth
- for the next iteration we use the smallest value $f(n)$ of node n that exceeded the limit in the last iteration
- frequently used algorithm

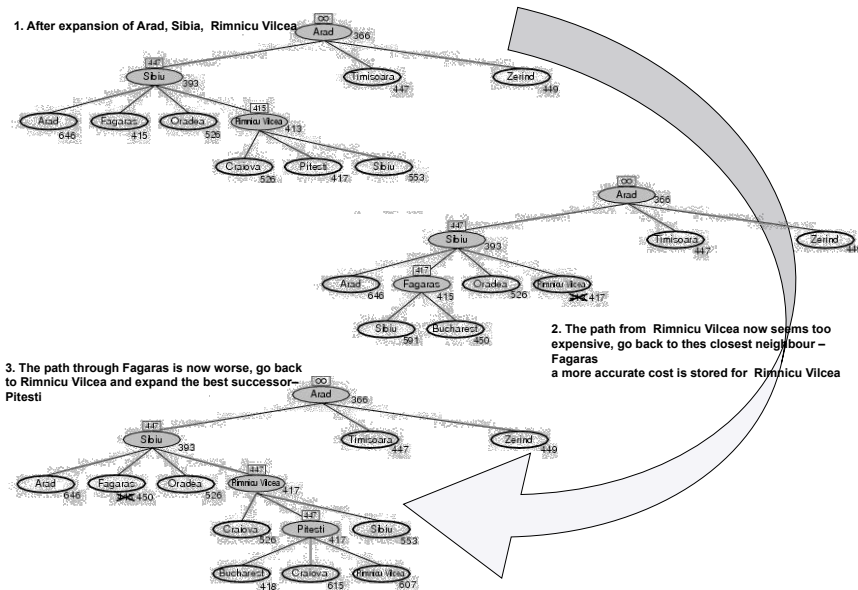
- Let us try to mimic standard best-first search, but using only linear space
 - the algorithm stops exploration if there is an alternative path with better cost $f(n)$
 - when the algorithm goes back to node n , it replaces the value $f(n)$ using the cost of successors (remembers the best leaf in the forgotten subtree)
- If $h(n)$ is an admissible heuristic then the algorithm is optimal.
- Space complexity $O(bd)$
- Time complexity is still exponential (suffers from excessive node re-generation)

```

function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
  RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f-limit) returns a solution, or failure and a new f-cost limit
  if GOAL-TEST[problem](STATE[node]) then return node
  successors ← EXPAND(node, problem)
  if successors is empty then return failure, ∞
  for each s in successors do
    f[s] ← max(g(s) + h(s), f[node])
  repeat
    best ← the lowest f-value node in successors
    if f[best] > f-limit then return failure, f[best]
    alternative ← the second-lowest f-value among successors
    result, f[best] ← RBFS(problem, best, min(f-limit, alternative))
  if result ≠ failure then return result
  
```

Recursive BFS - example



Simplified memory-bounded A*

- IDA* and RBFS do not exploit available memory!
- This is a pity as the already expanded nodes are re-expanded again (waste of time)
- Let us try to modify classical A*

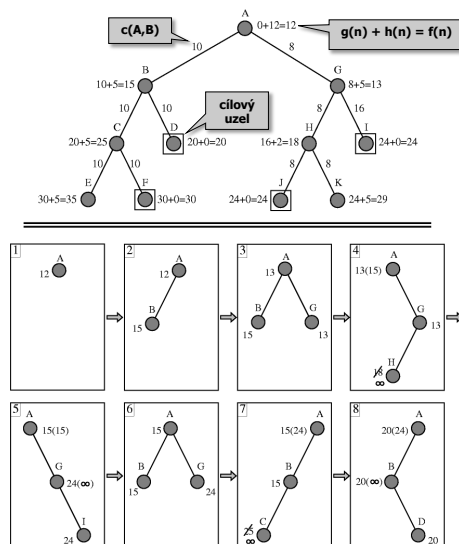
```

function SMA*(problem) returns a solution sequence
inputs: problem, a problem
static: Queue, a queue of nodes ordered by f-cost
Queue ← MAKE-QUEUE((MAKE-NODE(INITIAL-STATE[problem])))
loop do
  if Queue is empty then return failure
  n ← deepest least-f-cost node in Queue
  if GOAL-TEST(n) then return success
  s ← NEXT-SUCCESSOR(n)
  if s is not a goal and is at maximum depth then
    f(s) ← ∞
  else
    f(s) ← MAX(f(n), g(s)+h(s))
  if all of n's successors have been generated then
    update n's f-cost and those of its ancestors if necessary
  if SUCCESSORS(n) all in memory then remove n from Queue
  if memory is full then
    delete shallowest, highest-f-cost node in Queue
    remove it from its parent's successor list
    insert its parent on Queue if necessary
  insert s on Queue
end
    
```

Path from root to this non-goal node can be stored in memory, hence no optimal path through this node can be found.

- when memory is full, drop the worst leaf node – the node with the highest f-value (if there are such nodes then drop the shallowest node)
- similarly to RBFS back up the value of the forgotten node to its parent

Simplified memory-bounded A* - example

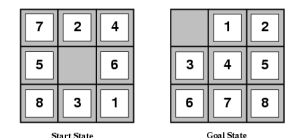


Looking for heuristics

How to find admissible heuristics?

Example: 8-puzzle

- 22 steps to goal in average
- branching factor around 3
- (complete) search tree: $3^{22} \approx 3,1 \times 10^{10}$ nodes
- the number of reachable states is only $9!/2 = 181\,440$
- for 15-puzzle there are 10^{13} states
- We need some heuristic, preferable admissible
 - h_1 = „the number of misplaced tiles” = 8
 - h_2 = „the sum of the distances of the tiles from the goal positions” = $3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$
a so called **Manhattan heuristic**
 - the optimal solution needs 26 steps



How to characterize the quality of a heuristic?

Effective branching factor b^*

- Let the algorithm needs N nodes to find a solution in depth d
- b^* is a branching factor of a uniform tree of depth d containing $N+1$ nodes

$$N+1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

Example:

- 15-puzzle
- the average over 100 instances for each of various solution lengths

d	Search Cost			Effective Branching Factor		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	30	25	2.80	1.33	1.24
10	47127	93	30	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- Is h_2 (from 8-puzzle) always better than h_1 and how to recognize it?
 - notice that $\forall n \ h_2(n) \geq h_1(n)$
 - We say that h_2 dominates h_1
 - A^* with h_2 never expands more nodes than A^* with h_1
 - A^* expands all nodes such that $f(n) < C^*$, t.j. $h(n) < C^* - g(n)$
 - In particular if it expands a node using h_2 , then the same node must be expanded using h_1
- It is always better to use a heuristic function giving higher values provided that
 - the limit $C^* - g(n)$ is not exceeded (then the heuristic would not be admissible)
 - the computation time is no too long

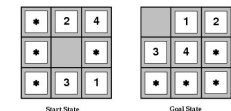
Can an agent construct admissible heuristics for any problem?

Yes via problem relaxation!

- relaxation is a simplification of the problem such that the solution of the original problem is also a solution of the relaxed problem (even if not necessarily optimal)
- we need to be able to solve the relaxed problem fast
- the cost of optimal solution to a relaxed problem is a lower bound for the solution to the original problem and hence it is an admissible (and monotonous) heuristic for the original problem
- Example (8-puzzle)**
 - A tile can move from square A to square B if:
 - A is horizontally or vertically adjacent to B
 - B is blank
 - possible relaxations (omitting some constraints to move a tile):
 - a tile can move from square A to square B if A is adjacent to B (Manhattan distance)
 - a tile can move from square A to square B if B is blank
 - a tile can move from square A to square B (heuristic h_1)

Another approach to admissible heuristics is using a pattern database

- based on solution of specific sub-problems (patterns)
- by searching back from the goal and recording the cost of each new pattern encountered
- heuristic is defined by taking the worst cost of a pattern that matches the current state
- Beware! The "sum" of costs of matching patterns need not be a **admissible** (the steps for solving one pattern may be used when solving another pattern).



If there are **more heuristics**, we can always use the **maximum** value from them (such a heuristic dominates each of used heuristics).