Department of Computer Science
Course: Introduction to Artificial Intelligence
Dr. Nurit Gal-Oz

INFORMED SEARCH ALGORITHMS

Chapter 3 (3.5, 3.6)

Based on the book: Artificial Intelligence A Modern Approach
Stuart Russell & Peter Norvig

Outline

- Best-first search
- Greedy best-first search
- A^* search
- Heuristics

Review: Tree search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

- A search strategy is defined by picking the order of node expansion
-

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability"
  -

  → Expand most desirable unexpanded node

- Implementation:

  frontier is a queue sorted in decreasing order of desirability

- Special cases:
  - greedy best-first search
  - A^* search
  -

Best-first search

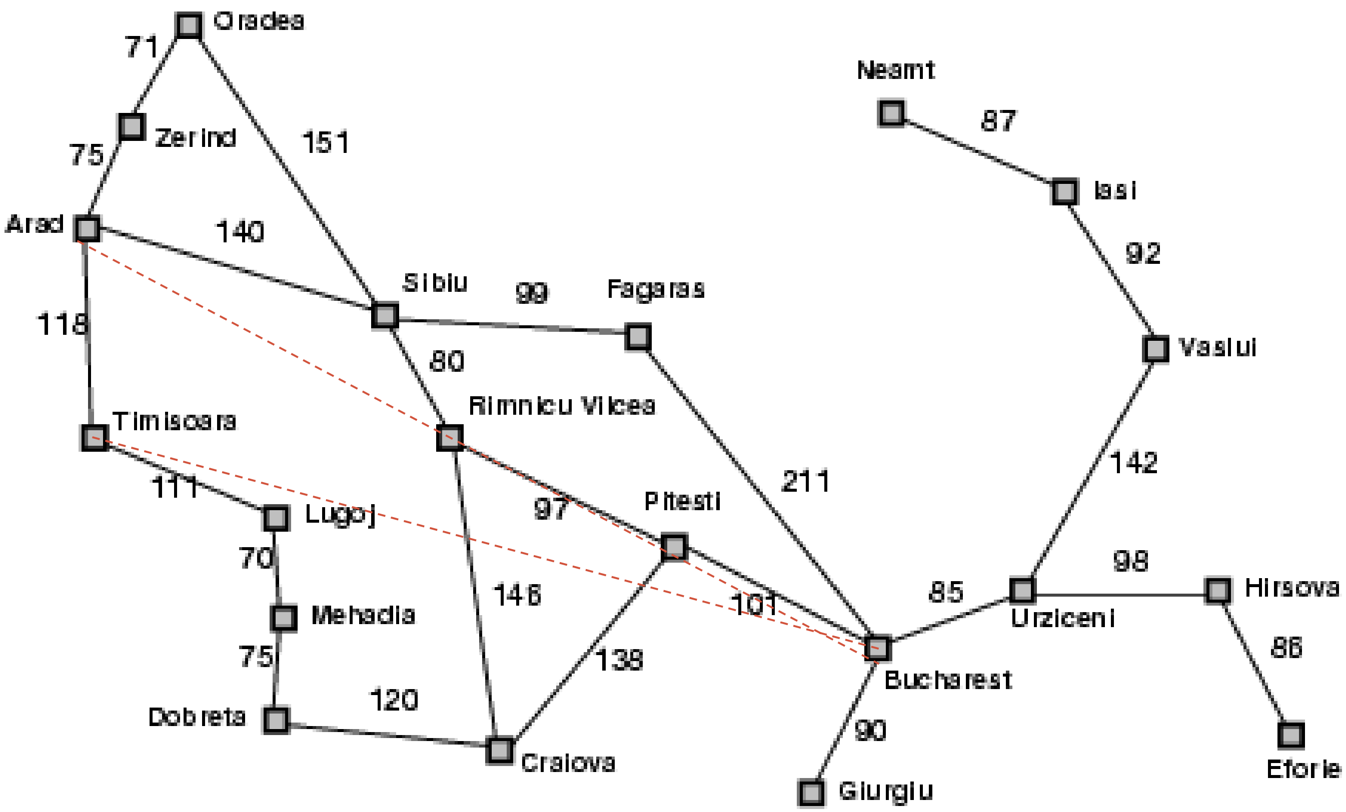
- Almost identical to that for uniform-cost search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Ordered by f(n) instead of g(n)

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Romania with step costs in km



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic)
  = estimate of cost from n to *goal*

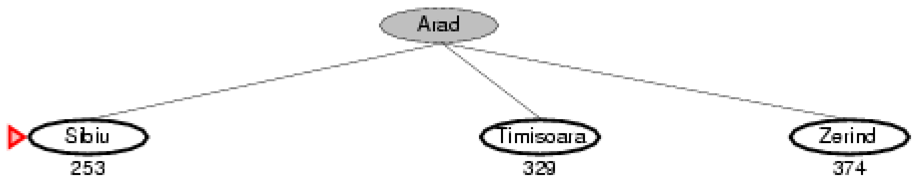
- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

- Greedy best-first search expands the node that appears to be closest to goal

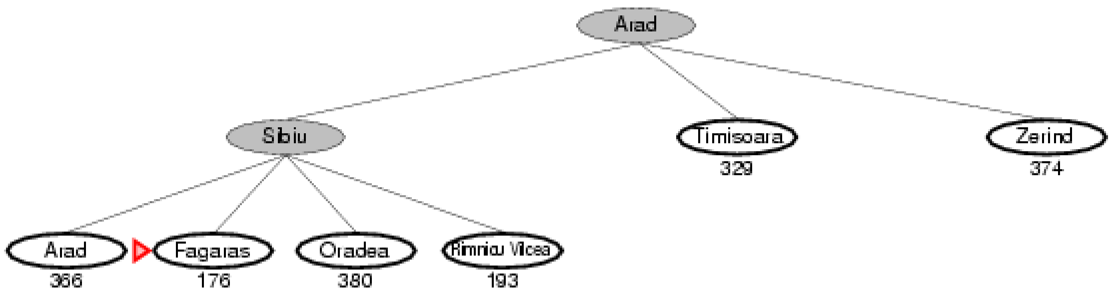
יוריסטיקה- "כלל אצבע" המכוון אותנו לקראת פתרון
הבעייה אך אינו בהכרח נכון תמיד או מדוייק

Greedy best-first search example

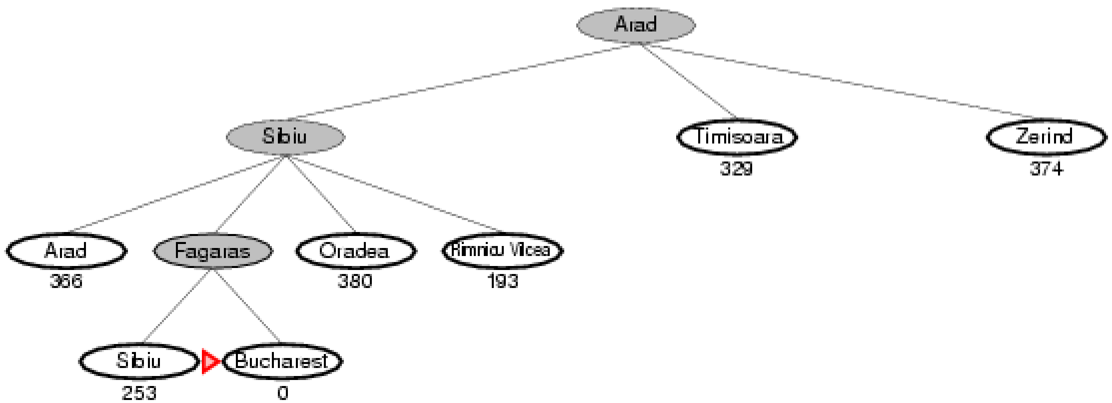
Greedy best-first search example



Greedy best-first search example



Greedy best-first search example



Properties of greedy best-first search

- Complete? No – tree version can get stuck in loops
  - e.g., Iasi → Neamt → Iasi → Neamt →
  - The graph search version *is* complete in finite spaces, but not in infinite ones
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

A^* search

- Idea: avoid expanding paths that are already expensive

- Evaluation function $f(n) = g(n) + h(n)$

- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

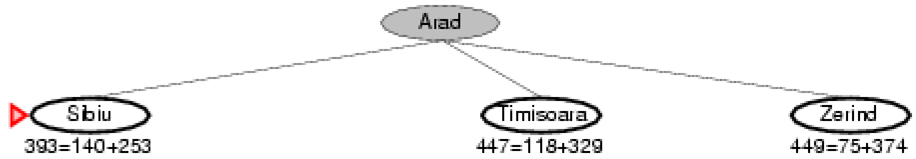
- A * is the most widely known form of best-first

A* search example

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

g(n) = cost so far to reach *n*
h(n) = estimated cost from *n* to goal
f(n) = estimated total cost of path through *n* to goal

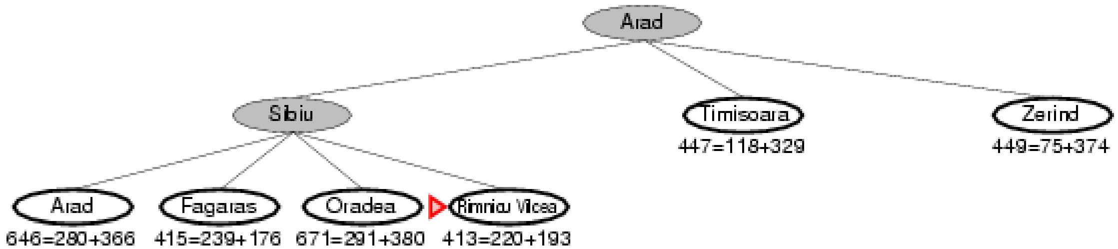
A* search example



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

g(n) = cost so far to reach *n*
h(n) = estimated cost from *n* to goal
f(n) = estimated total cost of path through *n* to goal

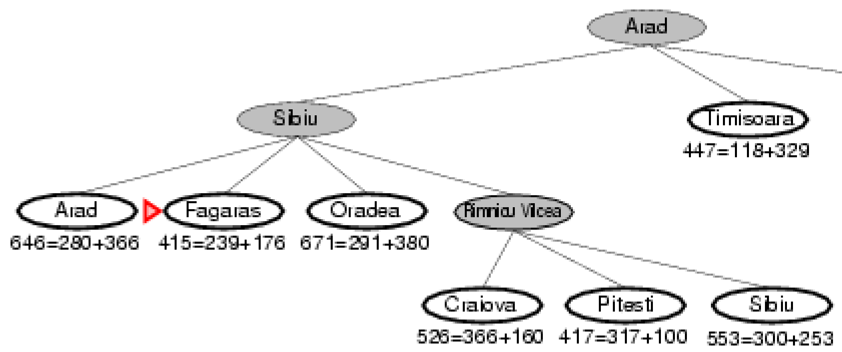
A* search example



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

g(n) = cost so far to reach *n*
h(n) = estimated cost from *n* to goal
f(n) = estimated total cost of path through *n* to goal

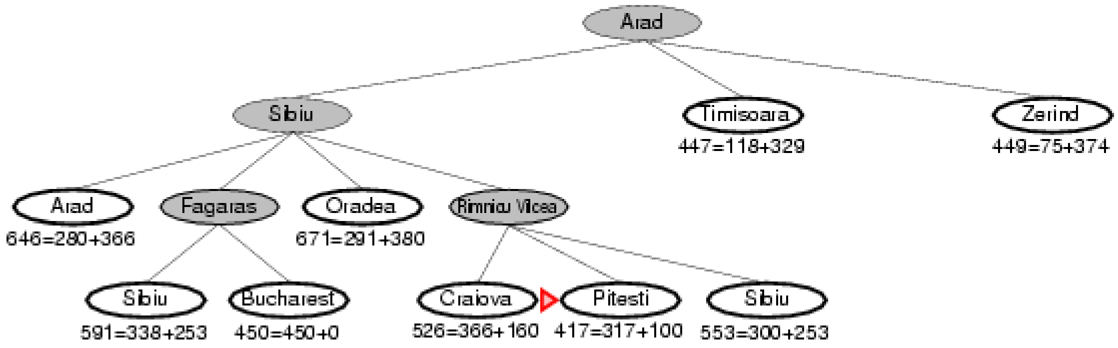
A* search example



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

g(n) = cost so far to reach *n*
h(n) = estimated cost from *n* to goal
f(n) = estimated total cost of path through *n* to goal

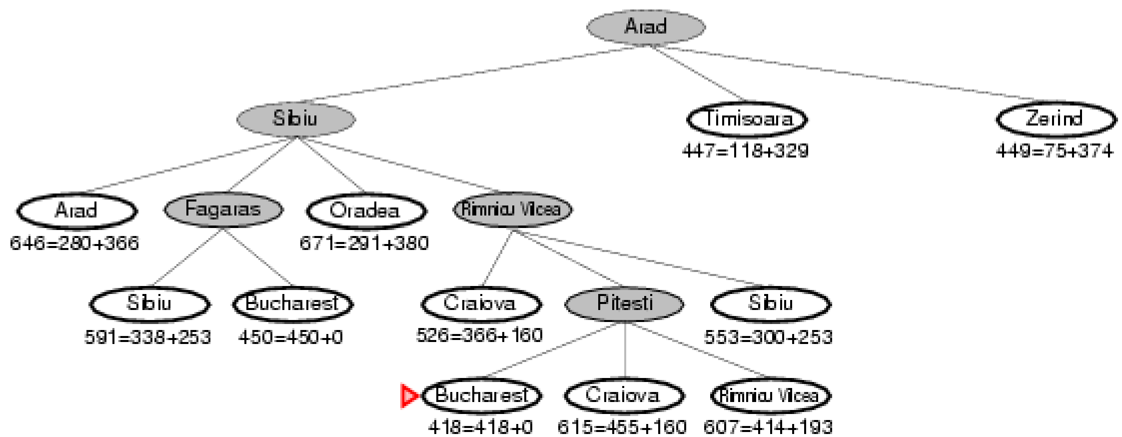
A* search example



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

g(n) = cost so far to reach *n*
h(n) = estimated cost from *n* to goal
f(n) = estimated total cost of path through *n* to goal

A* search example



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

$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost from n to goal
$f(n)$ = estimated total cost of path through n to goal

Conditions for optimality

- Conditions for optimality:
  - Admissibility קבילות
  - Consistency עקביות

Admissible heuristics

- A heuristic $h(n)$ is admissible if

  for every node n, $h(n) \leq h^*(n)$,

  where $h^*(n)$ is the true cost to reach the goal

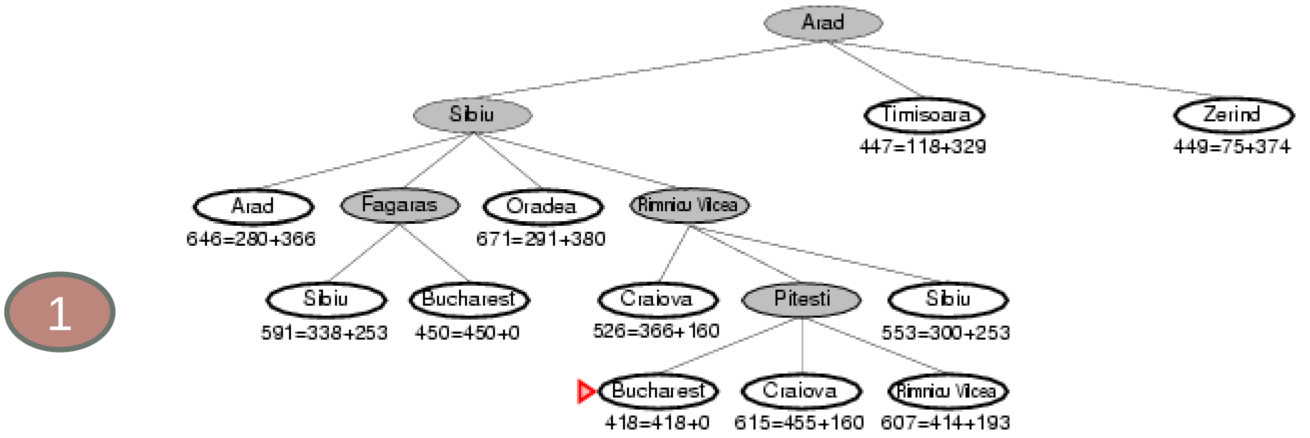
  state from n.

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

Theorem: If $h(n)$ is admissible, A^* using `TREE-SEARCH` is optimal

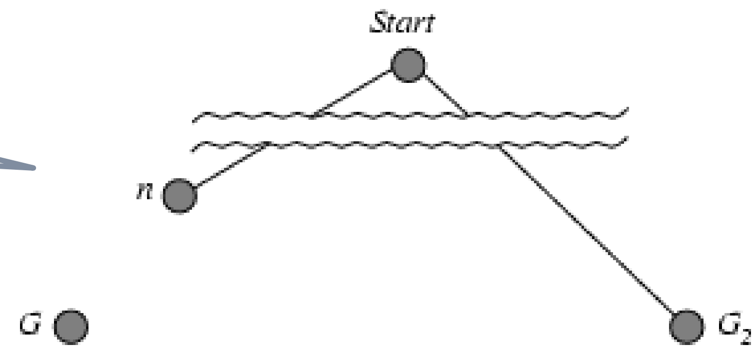
Admissible heuristics

- Bucharest first appears on the frontier at step (1)
  - It is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417).
  - Motivation: 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.



Optimality of A* (proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.
- **Proof sketch:**
  Show that f(n) < f(G2)
  (→ A* will prefer n over G2)



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$

∗ • $f(G_2) > f(G)$ from above

- $h(n) \leq h^*(n)$ since h is admissible
- $g(n) + h(n) \leq g(n) + h^*(n)$ (→ f(G) via n)

h*(n) – true of cost of getting to target from n

∗ • $f(n) \leq f(G)$

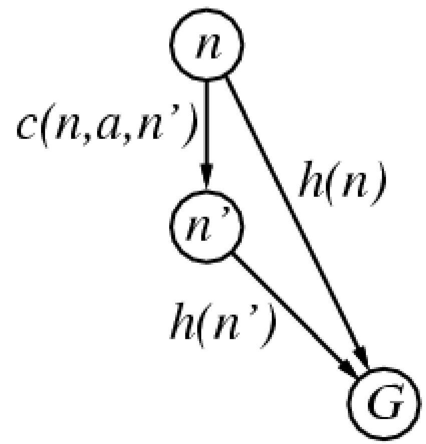
∗ • Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion

Consistent heuristics

- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,

$$h(n) \le c(n,a,n') + h(n')$$

$c(n,a,n')$
העלות הנמוכה ביותר
להגיע מ n ל n'



- If h is consistent, we have

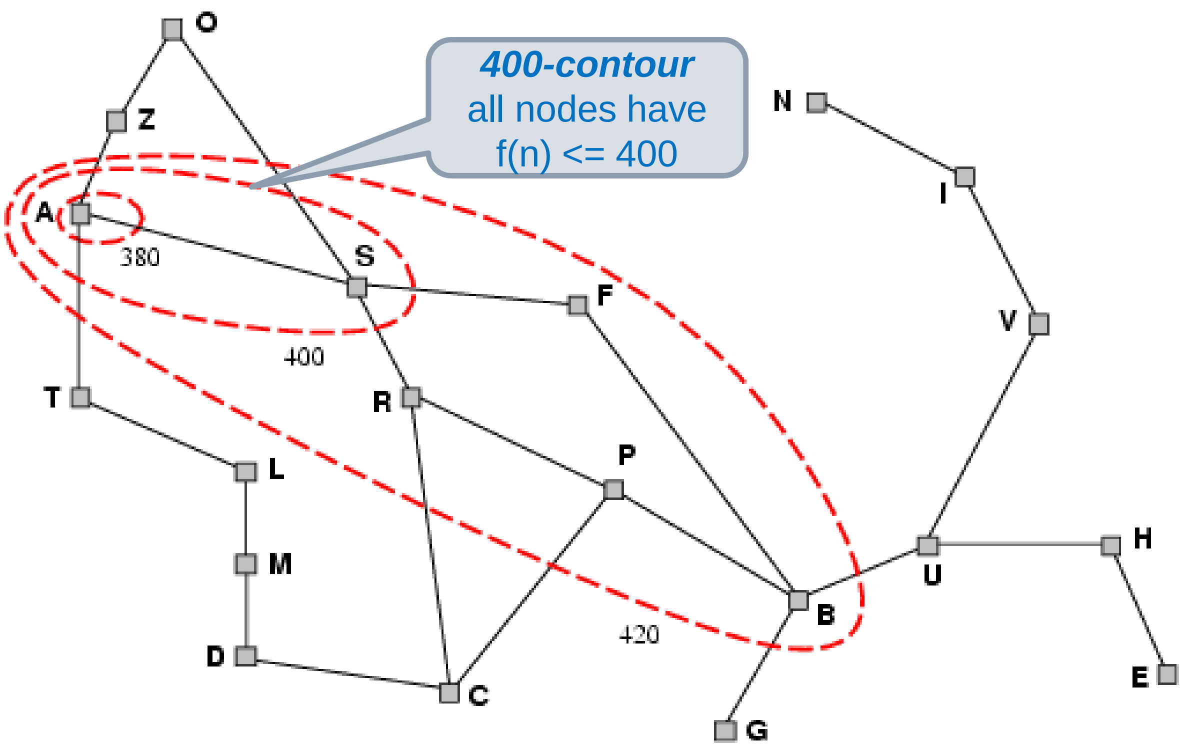
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\ge g(n) + h(n) \\ &= f(n) \end{aligned}$$

בחיפוש גרף קבילות לא מבטיחה חיפוש אופטימלי.
חיפוש גרף עלול לא לפתח צומת שפותח בעבר גם אם הדרך אליו לא היתה אופטימלית.

Theorem: If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal

Optimality of A*

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$



Properties of A*

- If C* is the cost of the optimal solution path:
  - 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*
    - true if all step costs exceed some finite Ɛ and if b is finite.

  - A* is optimally efficient for any given consistent heuristic
    - 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
  - **Pruning גיזום**
    - A∗ expands no nodes with f(n) > C* - for example Timisoara

Properties of A*

- Complete? Yes

(unless there are infinitely many nodes with f ≤ *f(G)*)

  - A* expands nodes in order of increasing f
  - Must find goal state unless
    - infinitely many nodes with f(n) < f*
      - infinite branching factor OR
      - finite path cost with infinite nodes on it

המקרה הגרוע ביותר h(n)=0 לכל n .
זהה ל חיפוש מונחה מחיר $O(b^{\frac{c}{\varepsilon}})$
המקרה הטוב ביותר h(n)=h*(n) לכל n.
סיבוכיות זמן ליניארית O(bd)

- Time? Exponential (depends on h)
      - Many heuristics lead to exponential number of nodes
      - Good heuristic – less nodes

- Space? : $O(b^m)$, Keeps all nodes in memory (!!)

- Optimal? Yes

Memory bounded heuristic search

- Iterative-deepening A* (IDA*)
  - Using f-cost(g+h) rather than the depth for cutoff
  - Cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration
  - Space complexity O(bd)

אין צורך לשמור תור
ממוין של צמתים

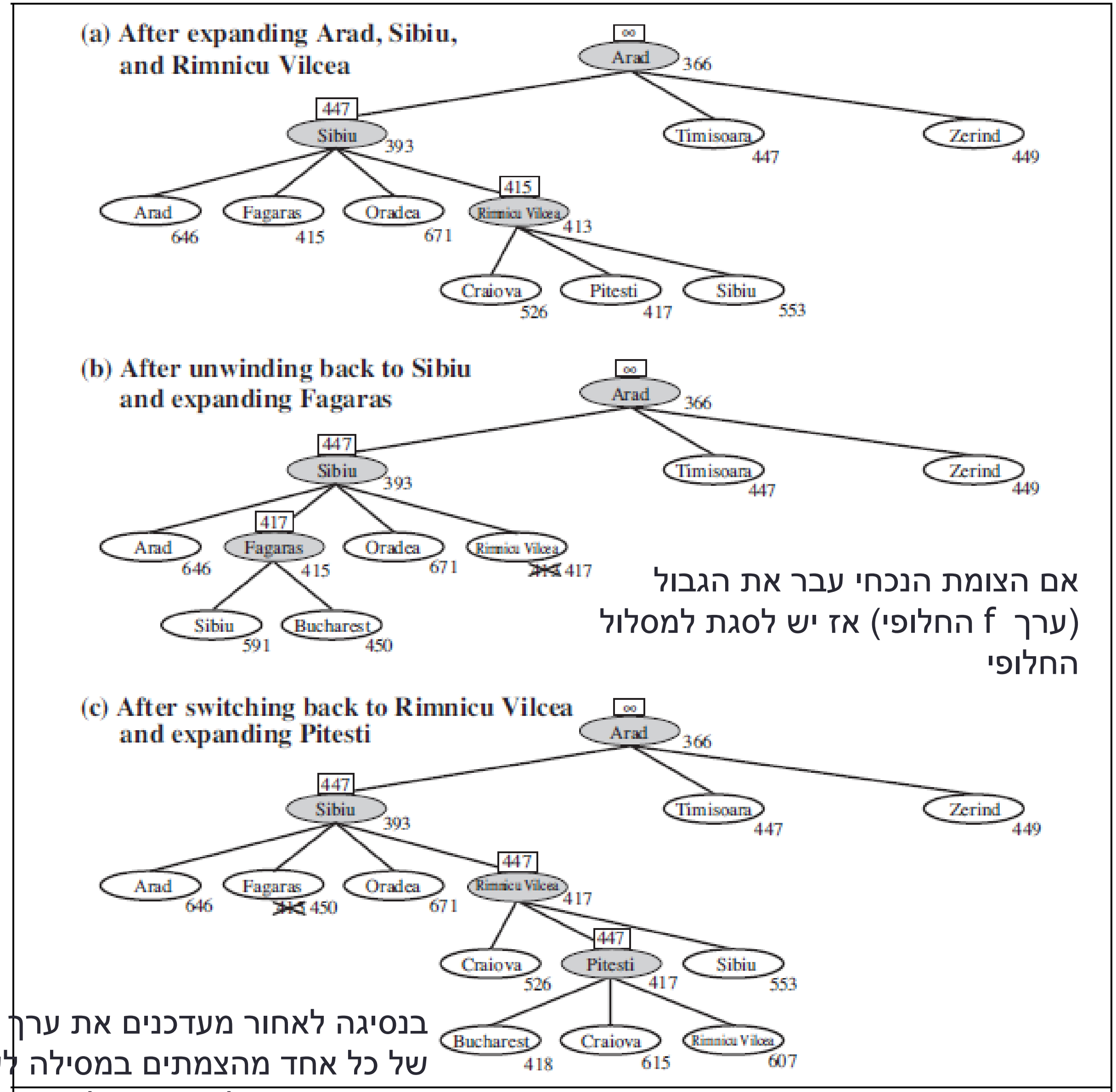
- Recursive best-first search (RBFS)
  - Best-first search using only linear space
  - It replaces the f-value of each node along the path with the best f-value of its children
  - Space complexity O(bd)

- Simplified memory bounded A* (SMA*)
  - IDA* and RBFS use too little memory – excessive node regeneration
  - Expanding the best leaf until memory is full
  - Dropping the worst leaf node (highest f-value) by backing up to its parent

מחקה את *best first* אלא שבמקום לרדת לעומק באופן בלתי מבוקר
שומרים על ערך f של המסלול החילופי הכי טוב שנמצא עד כה

RBFS

- **Figure 3.27**
- The f-limit value for each recursive call is shown on top of each current node
- every node is labeled with its f-cost.
- (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).
- (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450.
- (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.



אם הצומת הנכחי עבר את הגבול
(ערך f החלופי) אז יש לסגת למסלול
החלופי

בנסיגה לאחור מעדכנים את ערך f
של כל אחד מהצמתים במסילה לערך
הטוב ביותר של הבנים של הצומת

Simple Memory Bounded A* (SMA*)

- This is like A*, but -

  when memory is full we delete the worst node (largest f-value).
- Like RBFS

  we remember the best descendent in the branch we delete.
- If there is a tie (equal f-values) we delete the oldest nodes first.
- simple-MBA* finds the optimal *reachable* solution given the memory constraint.
- Time can still be exponential.

A Solution is not reachable
if a single path from root to goal
does not fit into memory

If there is enough
memory to store the
whole search tree
A*=SMA*

Simple Memory Bounded A* (SMA*)

```
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 in Queue
  end
```

מרחיב את הצומת העמוקה ביותר בעלת המחיר הנמוך ביותר.

המחיר של צומת שאינה צומת מטרה בעומק מקסימלי הוא ∞

"שוכח" את העלה בעל המחיר הגבוה ביותר .

Simple Memory-bounded A*

Progress of SMA*. Each node is labeled with its *current* *f*-cost. Values in parentheses show the value of the best forgotten descendant.

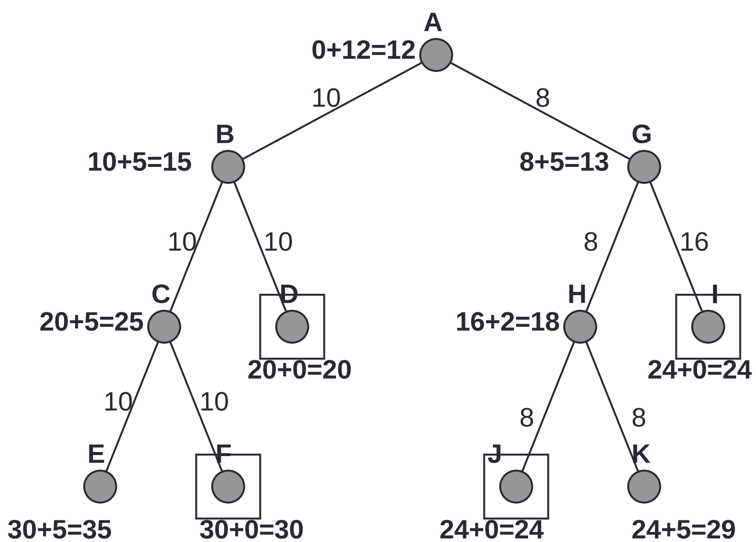
maximal depth is 3, since memory limit is 3. This branch is now useless.

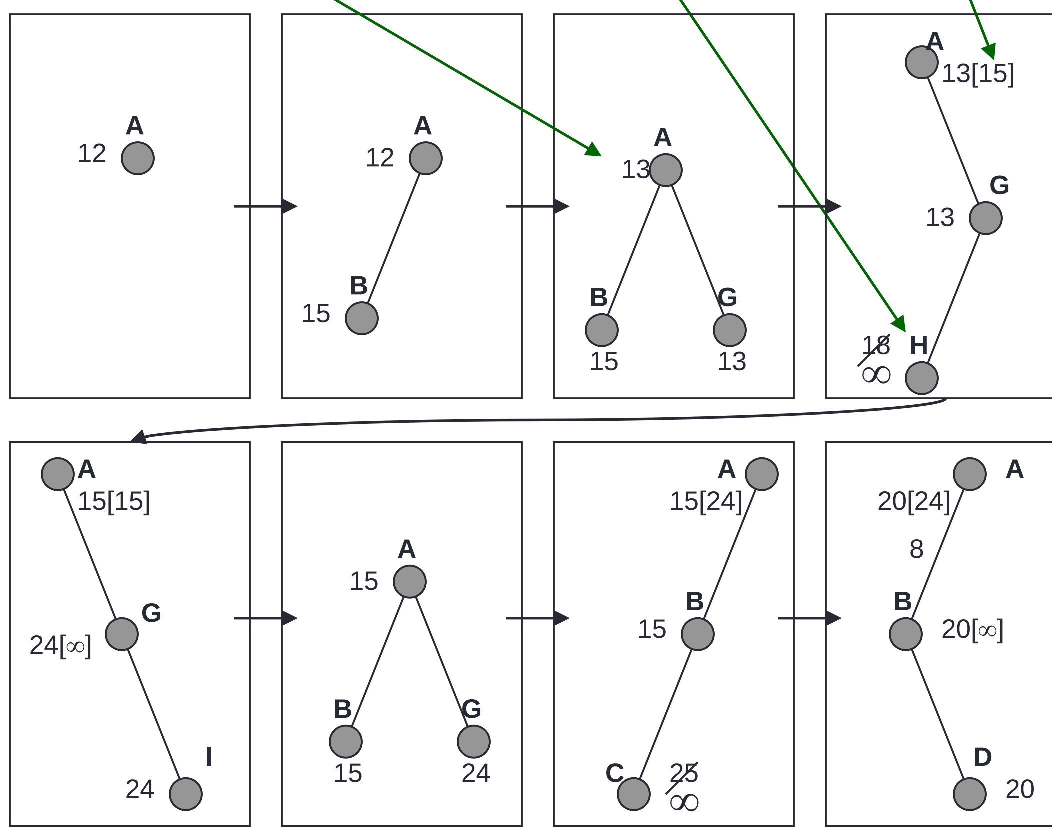
best forgotten node

best estimated solution so far for that node

Search space

f = g+h   □ = goal





(Example with 3-node memory)

אופטימלי אם יש פתרון אופטימלי שניתן להגיע אליו במגבלות הזיכרון = יש מספיק זכרון לשמור את המסלול

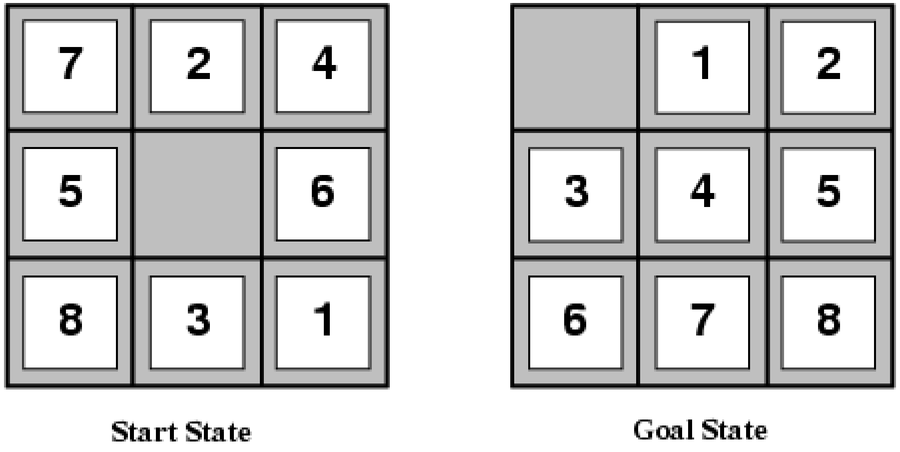
Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)

the distance between two points is the sum of the absolute differences of their Cartesian coordinates



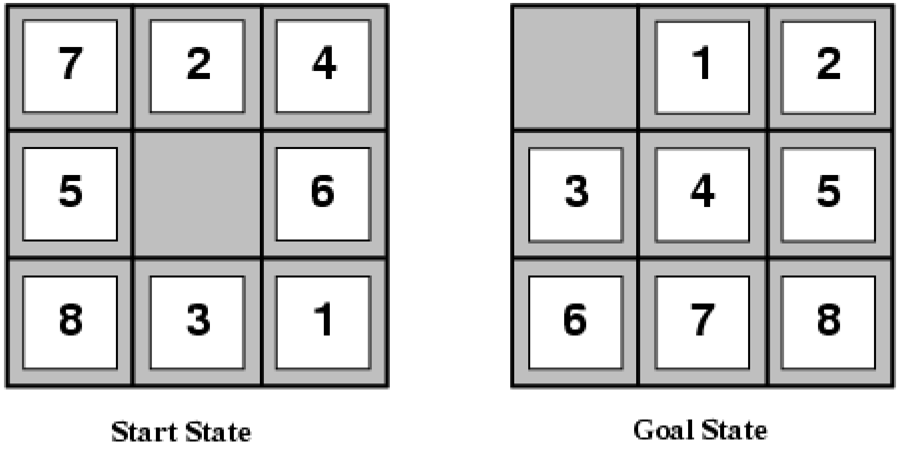
- $h_1(S) = ?$
- $h_2(S) = ?$

Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



- $h_1(S) = ?$ 8
- $h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18

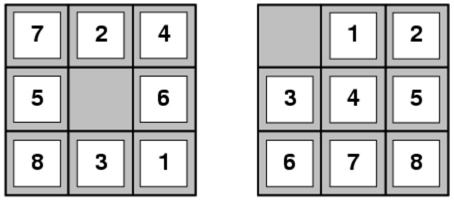
Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
  then h_2 dominates h_1
  → h_2 is better for search

- Typical search costs (average number of nodes expanded):

- *d=12* IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes

- *d=24* IDS = too many nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes



Start State Goal State

- IDS -Iterative-deepening-search

Dominance

	Search Cost (nodes generated)			Effective Branching Factor		
d	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	39	25	2.80	1.33	1.24
10	47127	93	39	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

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

HEURISTIC FUNCTIONS

Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem

(a) A tile can move from square A to square B if A is adjacent to B.
(b) A tile can move from square A to square B if B is blank.
(c) A tile can move from square A to square B.

- **The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem**

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance

- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution

- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Pattern databases

- Admissible heuristics can be derived from the solution cost of a **subproblem** of a given problem.
- Example:

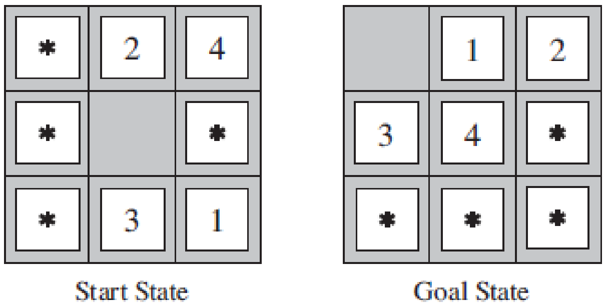


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

- The cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem.
- **Pattern databases** store the exact solution to for every possible subproblem instance.
  - The complete heuristic is constructed using the patterns in the DB

Pattern databases

- EXAMPLE:

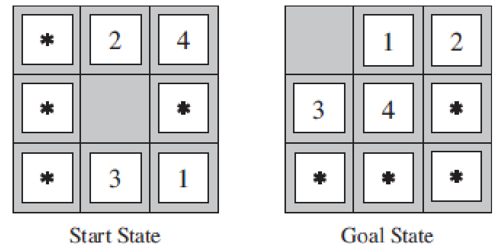


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

- The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions.
  - It turns out to be more accurate than Manhattan distance in some cases.
- The pattern database keeps every possible configuration of the four tiles and the blank.
  - The locations of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count toward the cost.)
  - Then we compute an admissible heuristic hDB for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database.

Learning heuristics from experience

- A heuristic function h(n) is supposed to estimate the cost of a solution beginning from the state at node n.
- How could an agent construct such a function?
  - devise relaxed problems for which an optimal solution can be found easily.
  - learn from experience.
    - Solve lots of 8-puzzles.
    - Each optimal solution to an 8-puzzle problem provides examples from which h(n) can be learned.
    - Each example consists of a state from the solution path and the actual cost of the solution from that point.
    - From these examples, a learning algorithm can be used to construct a function h(n) that can (with luck) predict solution costs for other states that arise during search.
    - Techniques for doing this - later on…