Informed search algorithms

This lecture topic

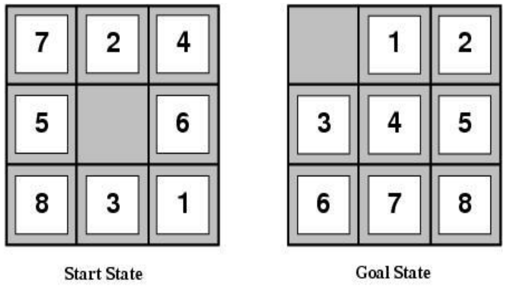
Chapter 3.3

Outline

- Review limitations of uninformed search methods
- **Informed (or heuristic) search uses problem-specific heuristics to improve efficiency**
  - Best-first, A* (and if needed for memory limits, RBFS, SMA*)
  - Techniques for generating heuristics
  - A* is optimal with admissible (tree)/consistent (graph) heuristics
  - A* is quick and easy to code, and often works ***very*** well
- **Heuristics**
  - A structured way to add "smarts" to your solution
  - Provide ***significant*** speed-ups in practice
  - Still have worst-case exponential time complexity
- In AI, "NP-Complete" means "Formally interesting"

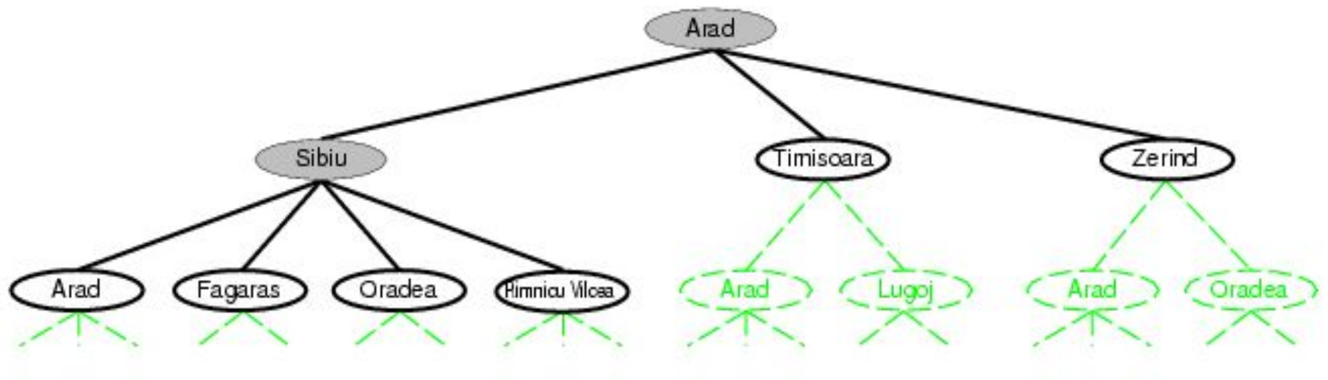
Limitations of uninformed search

- Search Space Size makes search tedious
  - **Combinatorial Explosion**
- For example, 8-puzzle
  - Avg. solution cost is about 22 steps
  - branching factor ~ 3
  - Exhaustive search to depth 22:
    - 3.1 x 10^{10} states
  - E.g., d=12, IDS expands 3.6 million states on average

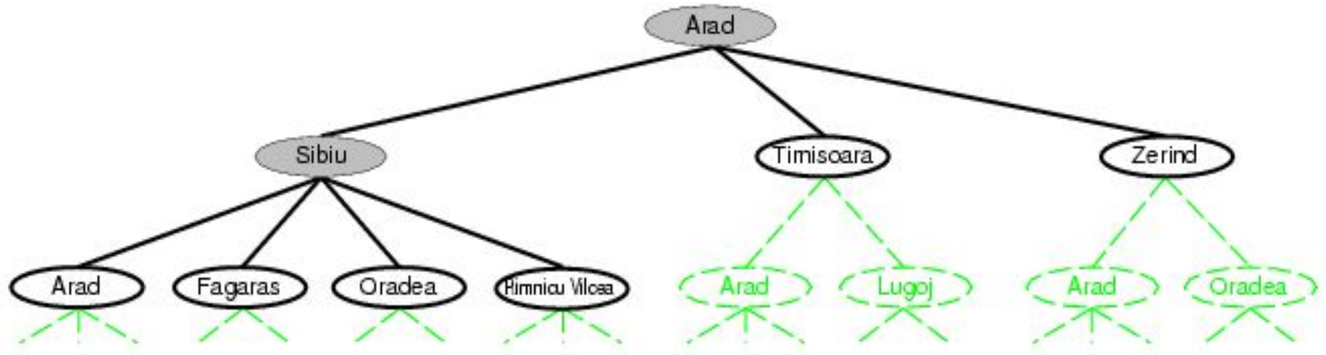


[24 puzzle has 10^{24} states (much worse)]

Recall tree search…



Recall tree search…



```
function TREE-SEARCH( problem, strategy) returns a solut
    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
```

This “strategy” is what differentiates different search algorithms

Heuristic search

- Idea: use an evaluation function *f(n)* for each node
  and a heuristic function *h(n)* for each node
  - g(n) = known path cost so far to node n.
  - h(n) = estimate of (optimal) cost to goal from node n.
  - f(n) = g(n)+h(n) = estimate of total cost to goal through node n.
  - f(n) provides an estimate for the total cost:
  - Expand the node n with smallest f(n).

- Implementation:
  Order the nodes in frontier by increasing estimated cost.

- Evaluation function is an estimate of node quality
  - ⇒ More accurate name for "best first" search would be "seemingly best-first search"

⇒ ***Search efficiency depends on heuristic quality!***
  - ⇒ ***The better your heuristic, the faster your search!***

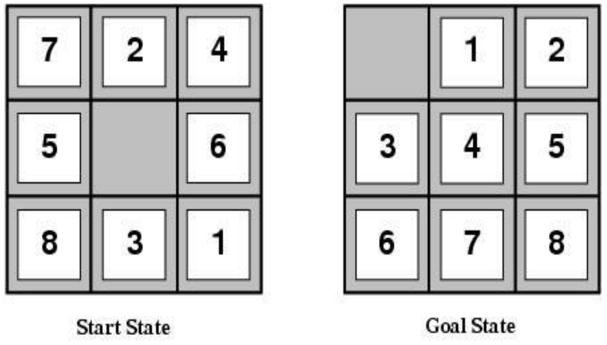
Heuristic function

- Heuristic:
  - Definition: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  - Same linguistic root as "Eureka" = "I have found it"
  - "using rules of thumb to find answers"

- Heuristic function *h(n)*
  - Estimate of (optimal) remaining cost from *n* to *goal*
  - Defined using only the ***state*** of node *n*
  - h(n) = 0 if n is a goal node
  - Example: straight line distance from n to Bucharest
    - Note that this is not the true state-space distance
    - It is an estimate – actual state-space distance can be higher

  - Provides problem-specific knowledge to the search algorithm

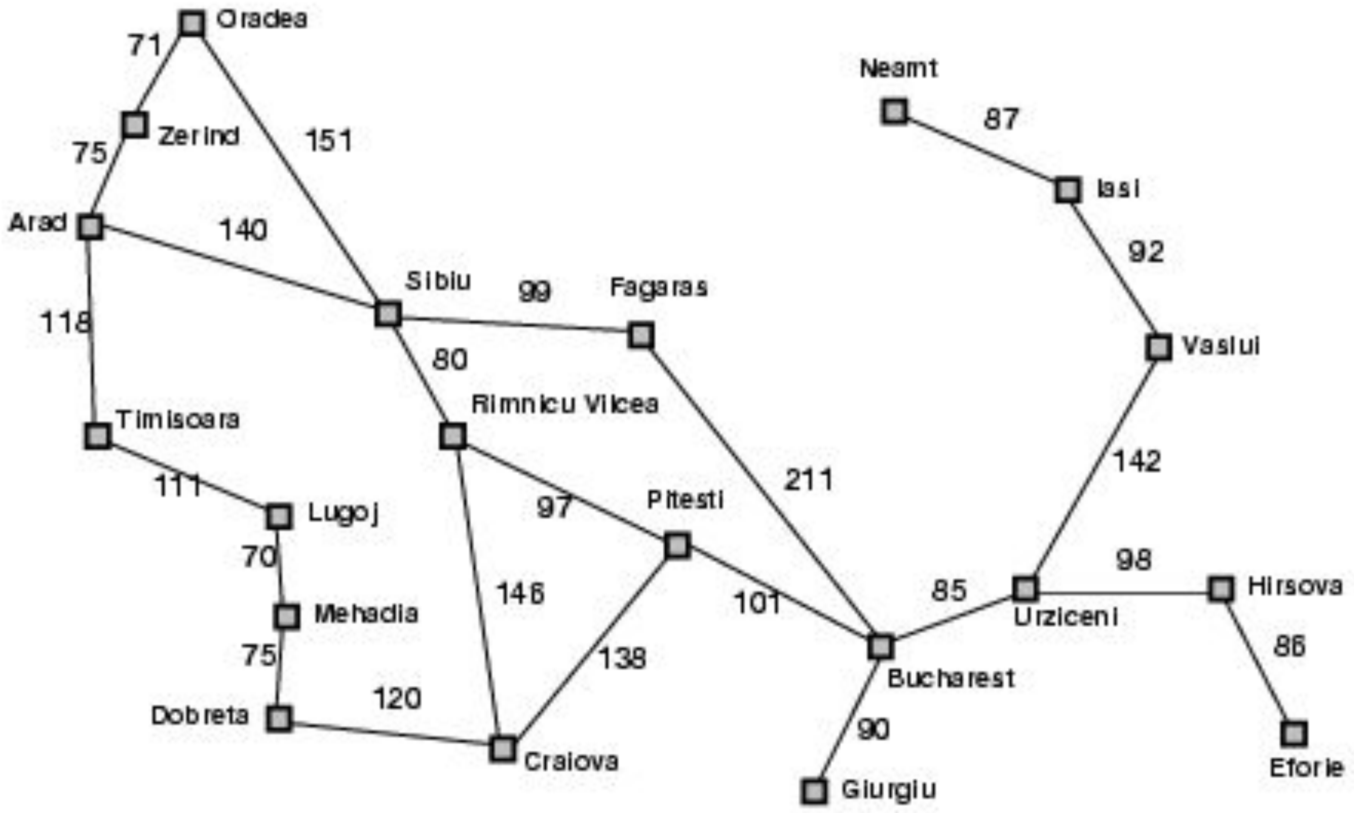
Heuristic functions for 8-puzzle



- 8-puzzle
  - Avg. solution cost is about 22 steps
  - branching factor ~ 3
  - Exhaustive search to depth 22:
    - 3.1×10^{10} states.
  - A good heuristic function can reduce the search process.

- Two commonly used heuristics
  - h_1 = the number of misplaced tiles
    - $h_1(s)=8$
  - h_2 = the sum of the distances of the tiles from their goal positions (Manhattan distance).
    - $h_2(s)=3+1+2+2+2+3+3+2=18$

Romania with straight-line dist.



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

Relationship of Search Algorithms

- *g(n)* = known cost so far to reach *n*
- *h(n)* = estimated (optimal) cost from *n* to goal
- *f(n) = g(n) + h(n)*

  = estimated (optimal) total cost of path through *n* to goal

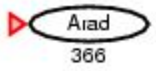
- Uniform Cost search sorts frontier by *g(n)*
- Greedy Best First search sorts frontier by *h(n)*
- A* search sorts frontier by *f(n)*
  - ***Optimal for admissible/consistent heuristics***
  - ***Generally the preferred heuristic search***
- Memory-efficient versions of A* are available
  - RBFS, SMA*

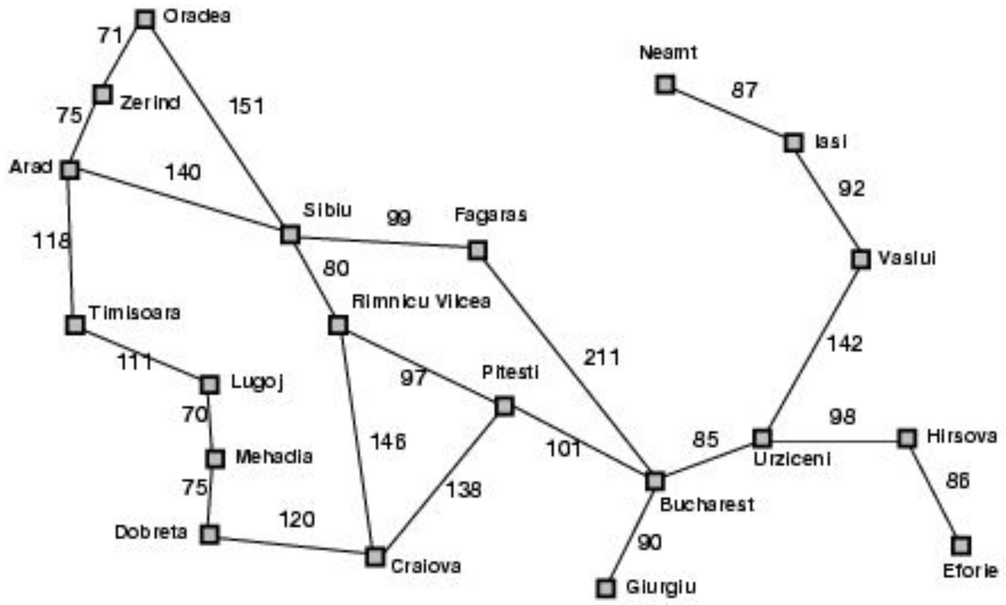
Greedy best-first search (often called just “best-first”)

- *h(n)* = estimate of cost from *n* to *goal*
  - e.g., *h(n)* = straight-line distance from *n* to Bucharest

- Greedy best-first search expands the node that appears to be closest to goal.
  - *Priority queue sort function = h(n)*

Greedy best-first search example

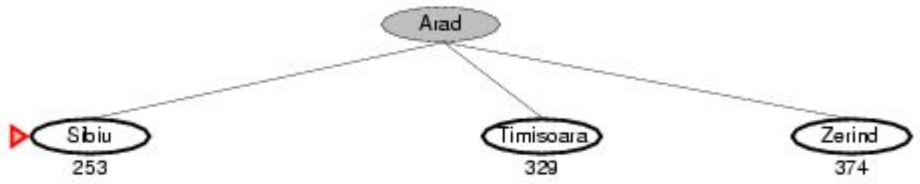


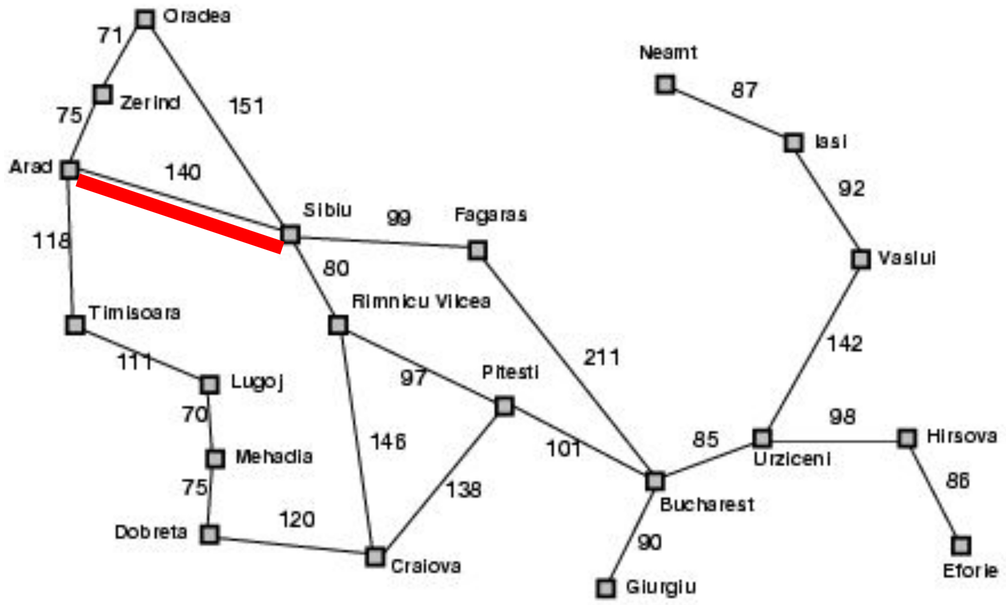


Straight–line distance to Bucharest

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 example

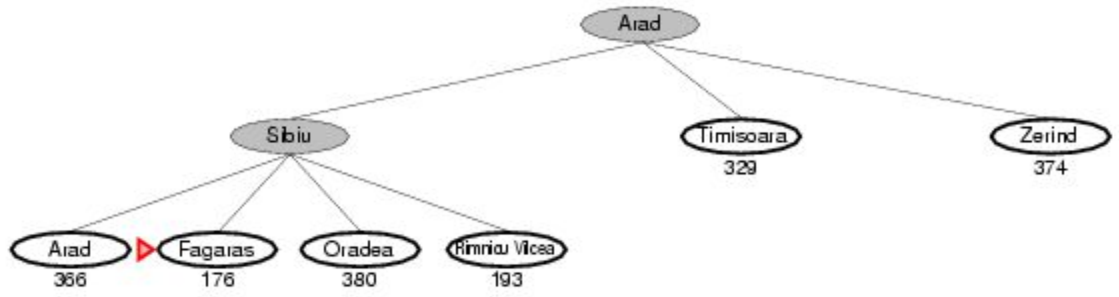


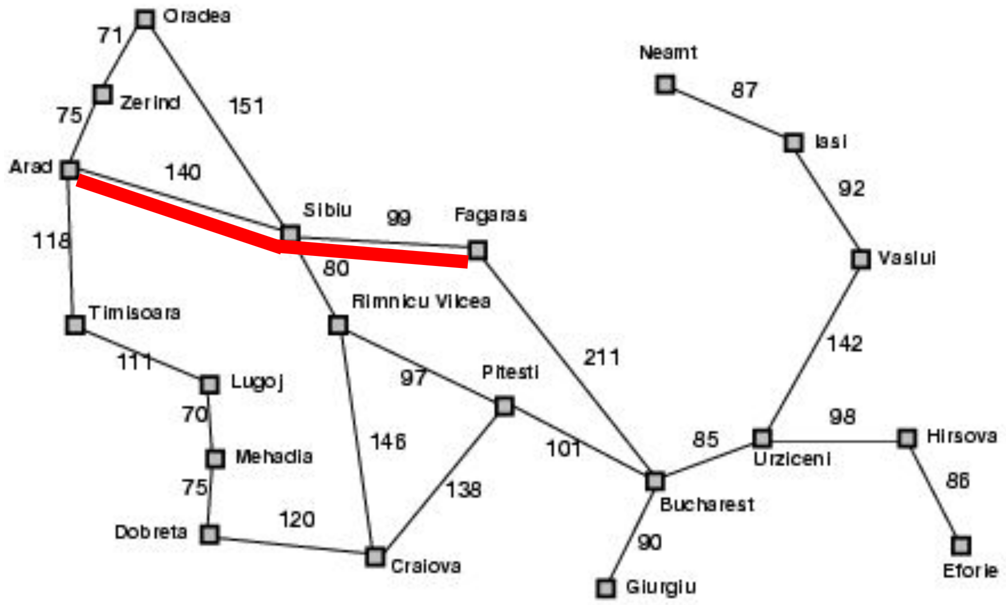


Straight–line distance to Bucharest

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 example

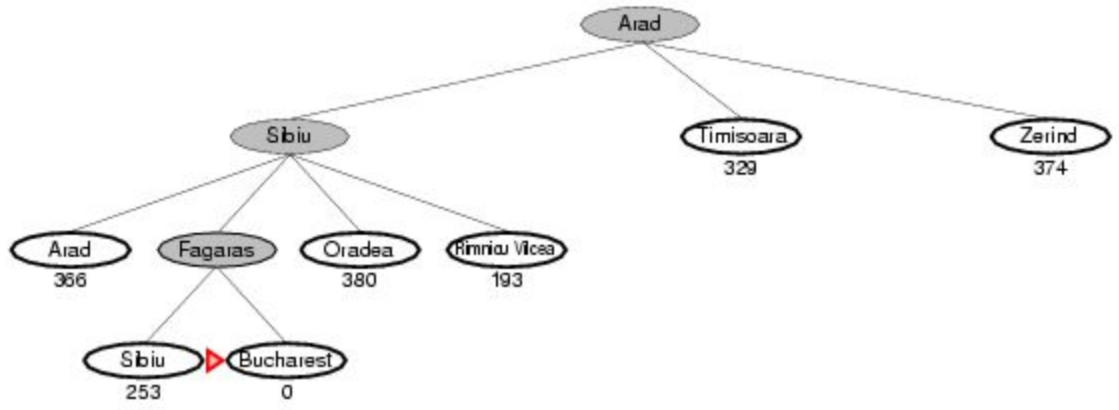


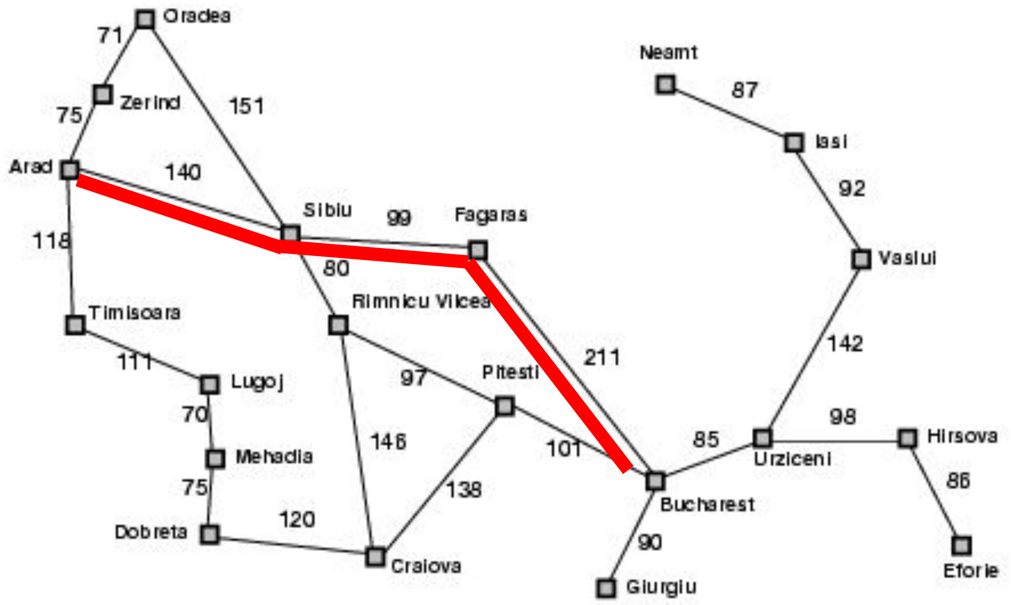


Straight–line distance to Bucharest

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 example

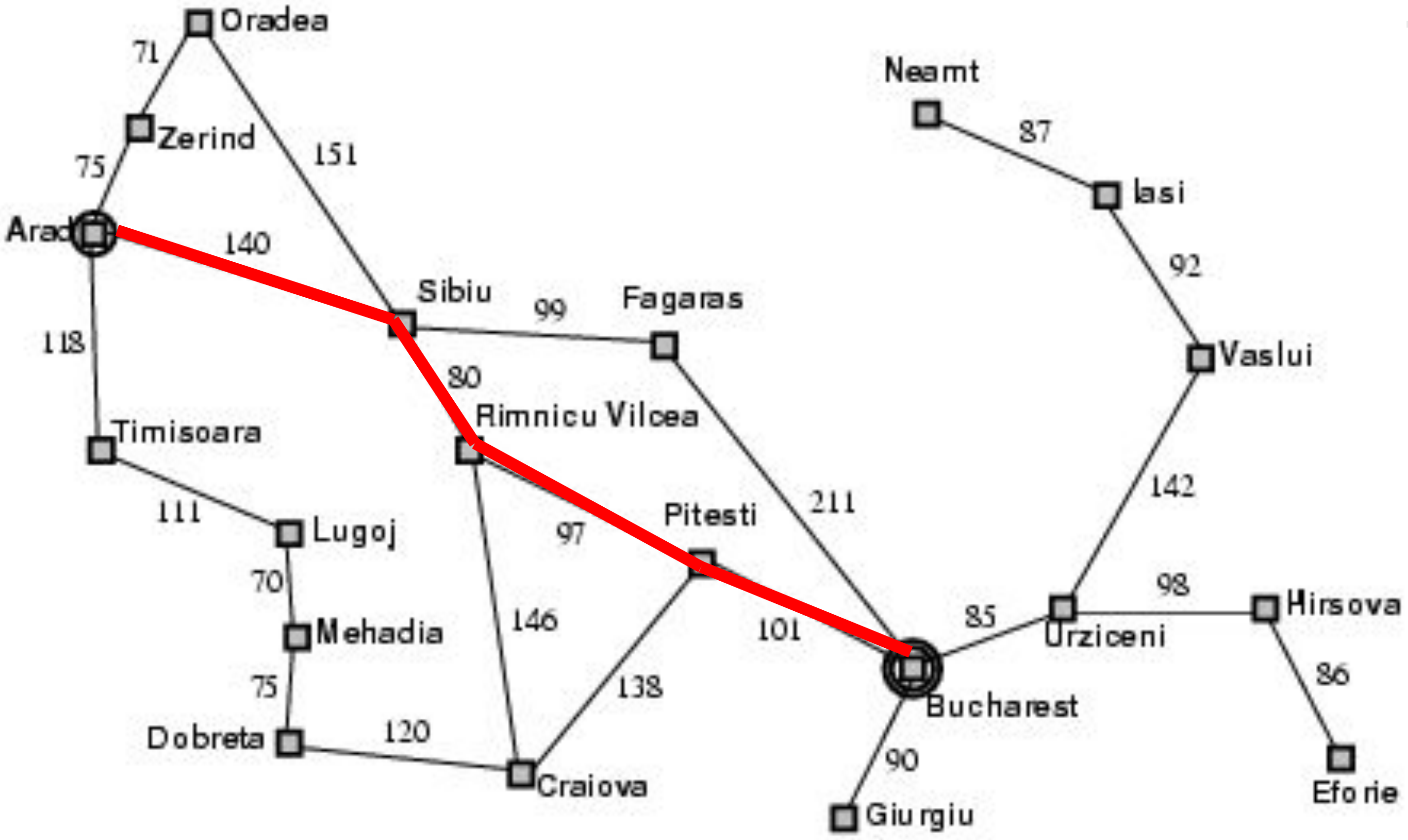




Straight–line distance to Bucharest

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

Optimal Path



Properties of greedy best-first search

- Complete?
  - Tree version can get stuck in loops.
  - Graph version is complete in finite spaces.
- Time? $O(b^m)$
  - A good heuristic can give **dramatic** improvement
- Space? $O(b^m)$
  - Keeps all nodes in memory
- Optimal? No

  e.g., Arad ☐ Sibiu ☐ Rimnicu Vilcea ☐ Pitesti ☐ Bucharest is shorter!

A^* search

- Idea: avoid paths that are already expensive
  - Generally the preferred simple heuristic search
  - Optimal if heuristic is: admissible(tree)/consistent(graph)
- Evaluation function *f(n) = g(n) + h(n)*
  - g(n) = known path cost so far to node n.
  - h(n) = estimate of (optimal) cost to goal from node n.
  - f(n) = g(n)+h(n)
    = estimate of total cost to goal through node n.
- *Priority queue sort function = f(n)*

Admissible heuristics

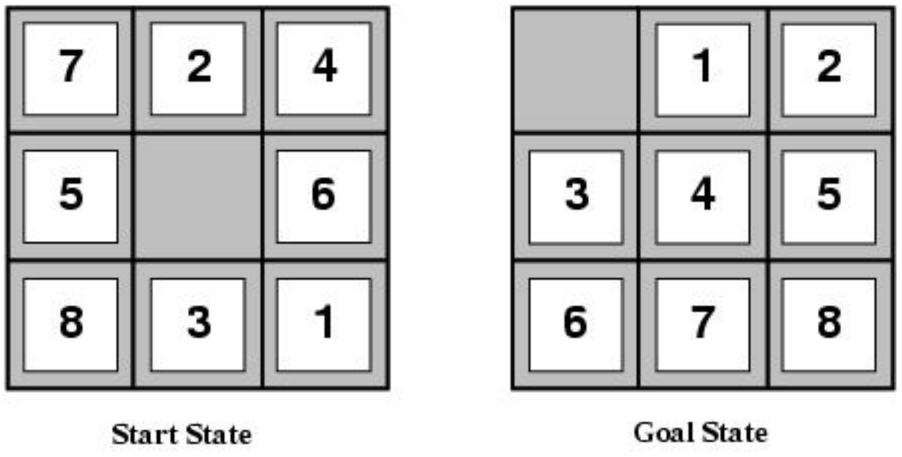
- A heuristic $h(n)$ is admissible if for every node n,
  $h(n) \leq h^*(n)$,
  where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic (or at least, never pessimistic)
  - Example: $h_{SLD}(n)$ (never overestimates actual road distance)
- Theorem:
  If $h(n)$ is admissible, A* using `TREE-SEARCH` is optimal

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)



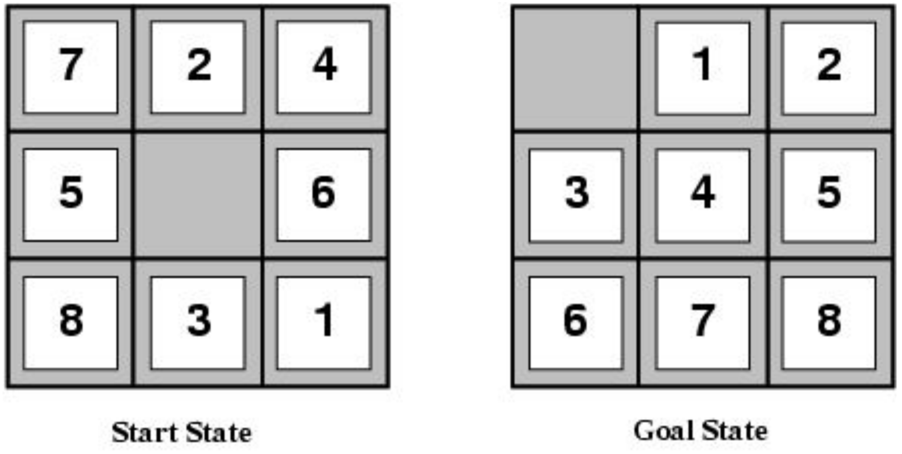
- $h_1(S) = ?$
- $h_2(S) = ?$

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)

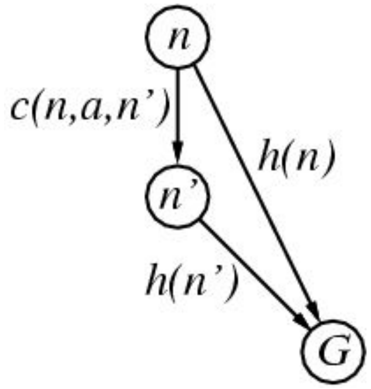


- $h_1(S) = ?$ 8
- $h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18

Consistent heuristics (consistent => admissible)

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

$$\boxed{h(n) \leq c(n,a,n') + h(n')}$$



- If *h* is consistent, we have

$$
\begin{aligned}
f(n') &= g(n') + h(n') && \text{(by def.)}\\
&= g(n) + c(n,a,n') + h(n') && (g(n')=g(n)+c(n.a.n'))\\
&\geq g(n) + h(n) = f(n) && \text{(consistency)}\\
f(n') &\geq f(n)
\end{aligned}
$$

It's the triangle inequality !

- i.e., *f(n)* is non-decreasing along any path.

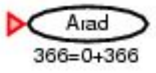
- Theorem:
  If *h(n)* is consistent, A* using GRAPH−SEARCH is optimal

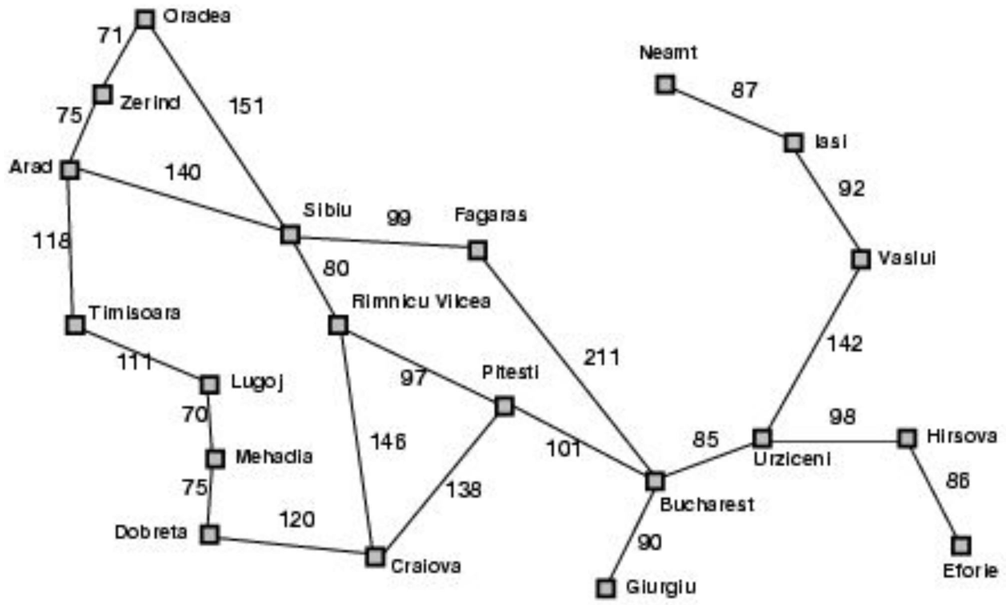
keeps all checked nodes in memory to avoid repeated states

Admissible (Tree Search) vs. Consistent (Graph Search)

- Why two different conditions?
  - In graph search you often find a long cheap path to a node after a short expensive one, so you might have to update all of its descendants to use the new cheaper path cost so far
  - A consistent heuristic avoids this problem (it can't happen)
  - Consistent is slightly stronger than admissible
  - Almost all admissible heuristics are also consistent
- Could we do optimal graph search with an admissible heuristic?
  - Yes, but you would have to do additional work to update descendants when a cheaper path to a node is found
  - A consistent heuristic avoids this problem

A* search example

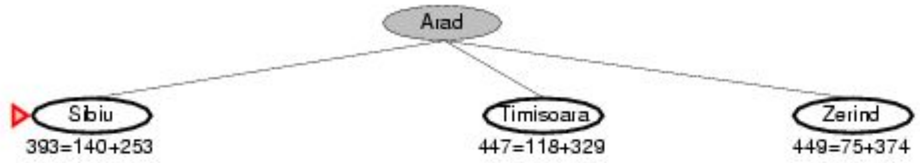


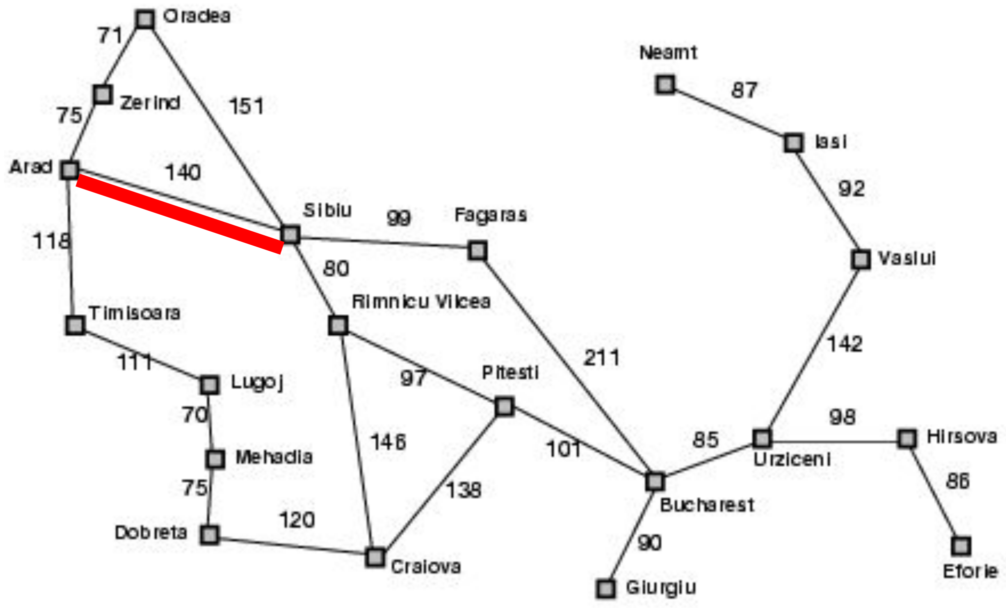


Straight–line distance to Bucharest

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

A* search example

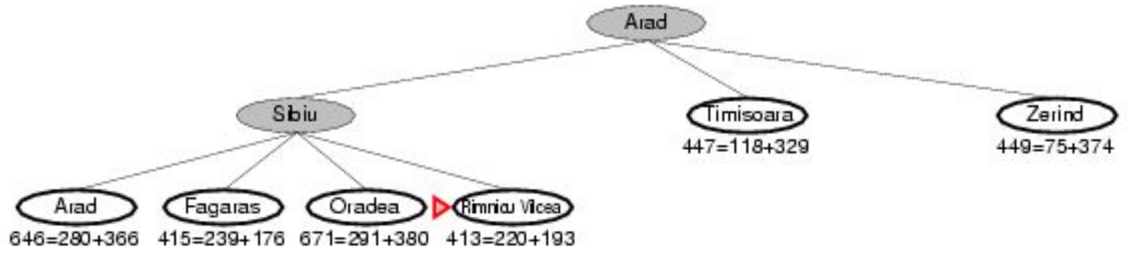


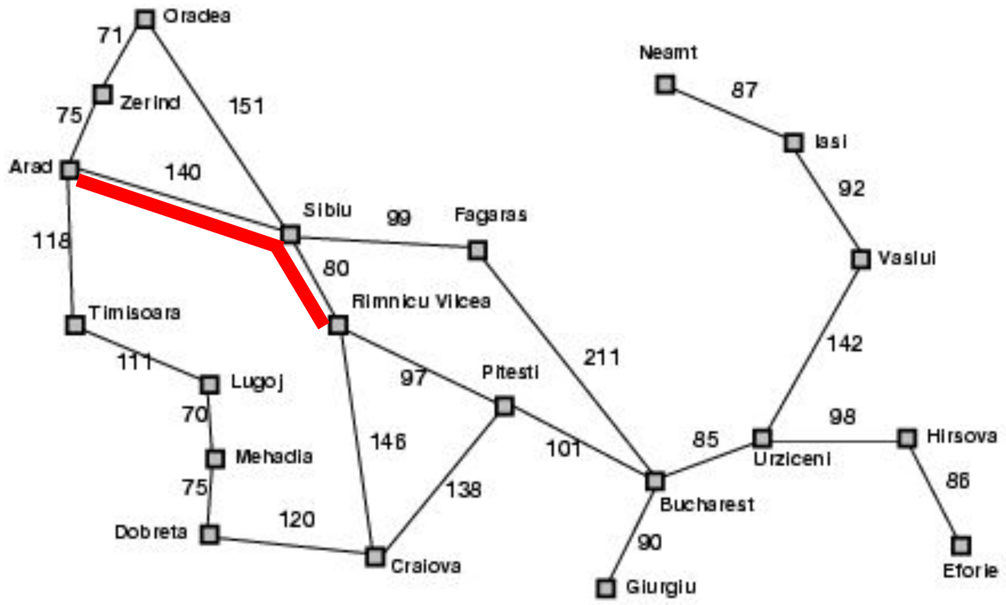


Straight–line distance
to Bucharest

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

A* search example

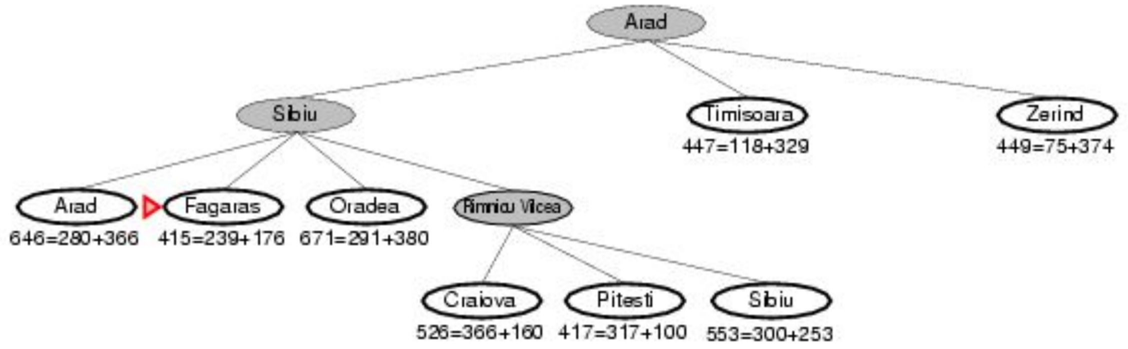


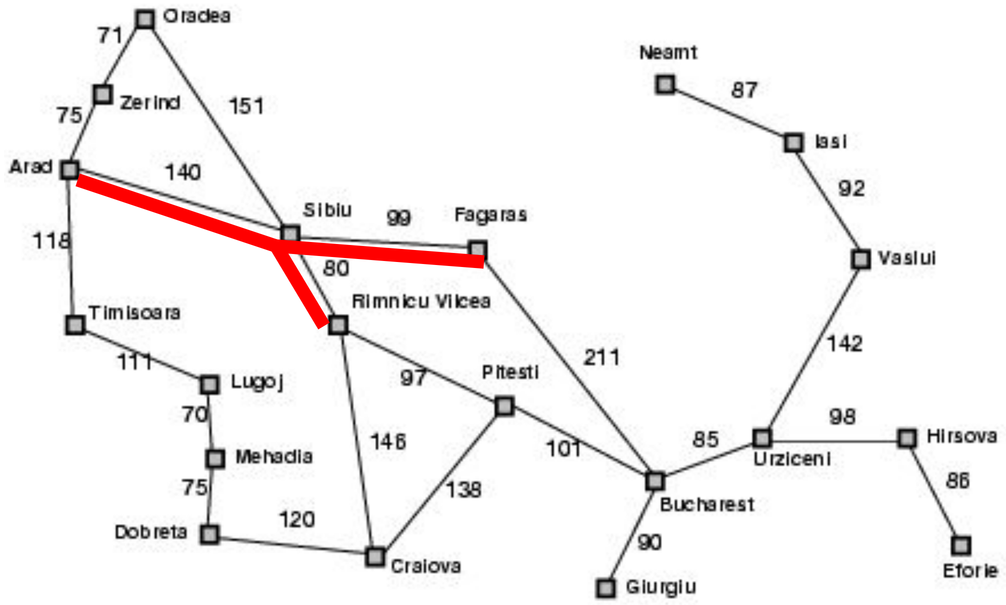


Straight−line distance to Bucharest

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

A* search example

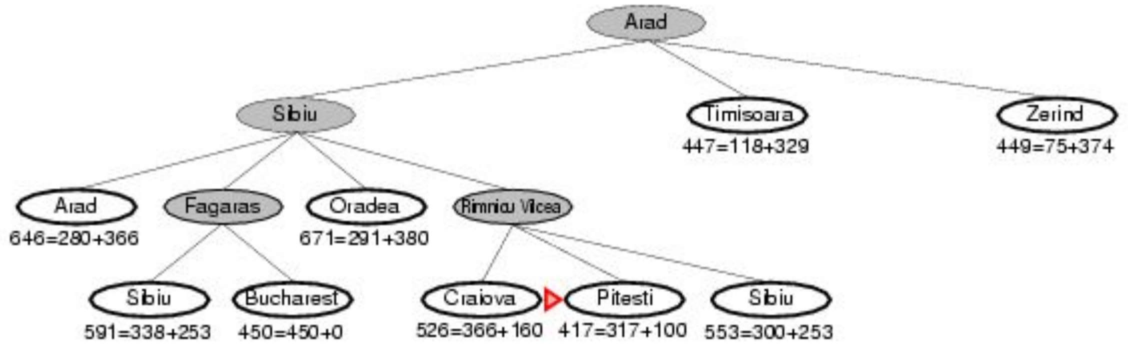


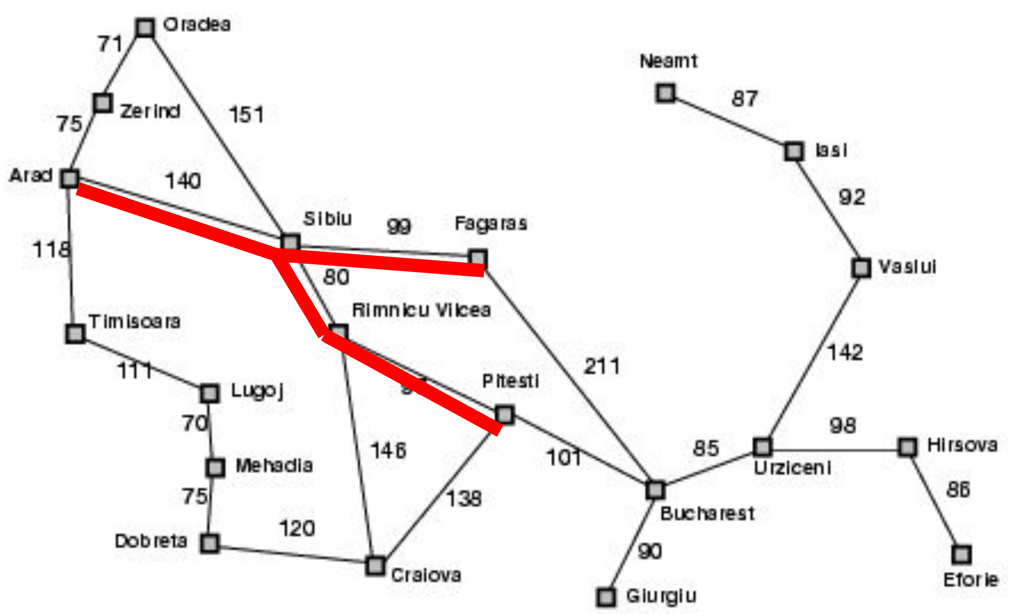


Straight–line distance to Bucharest

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

A* search example

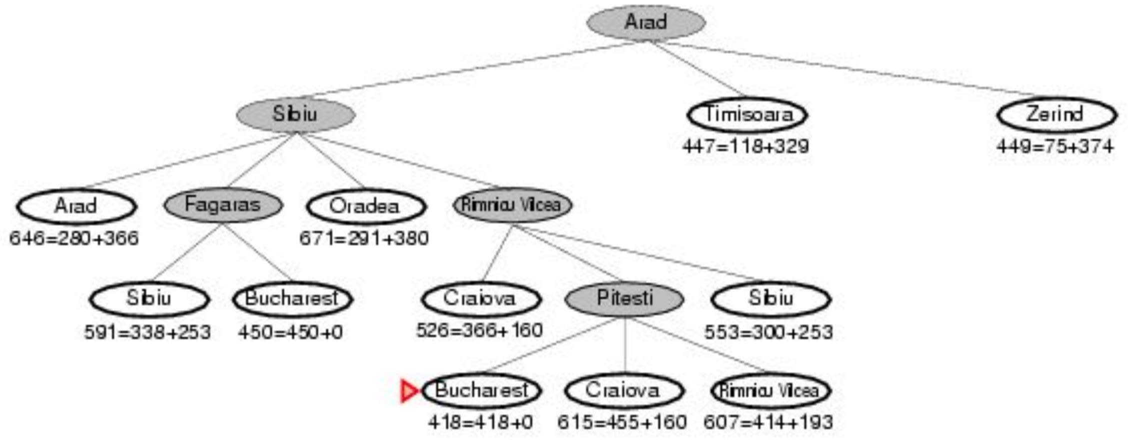


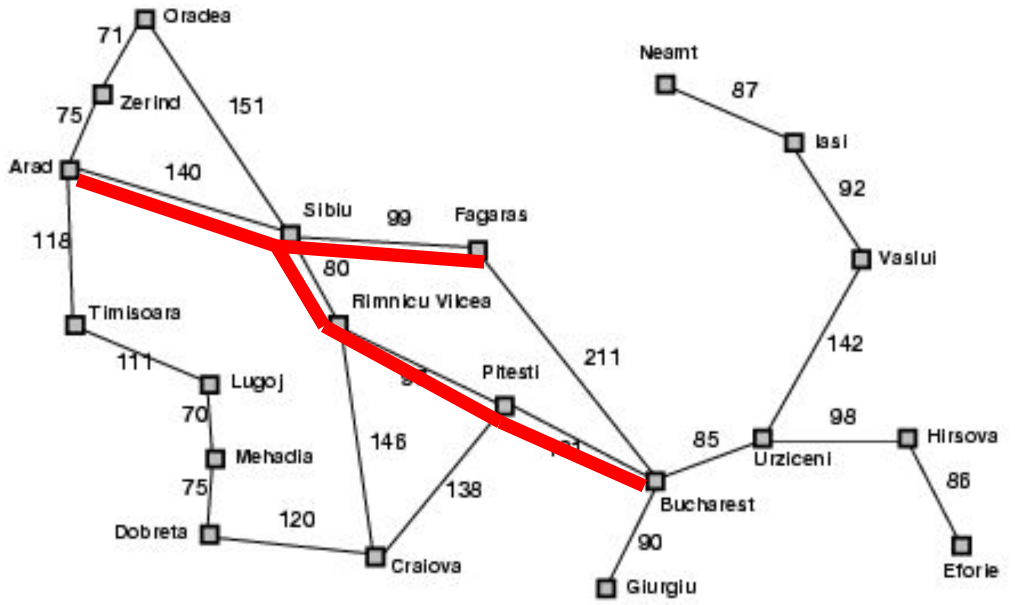


Straight–line distance to Bucharest

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

A* search example



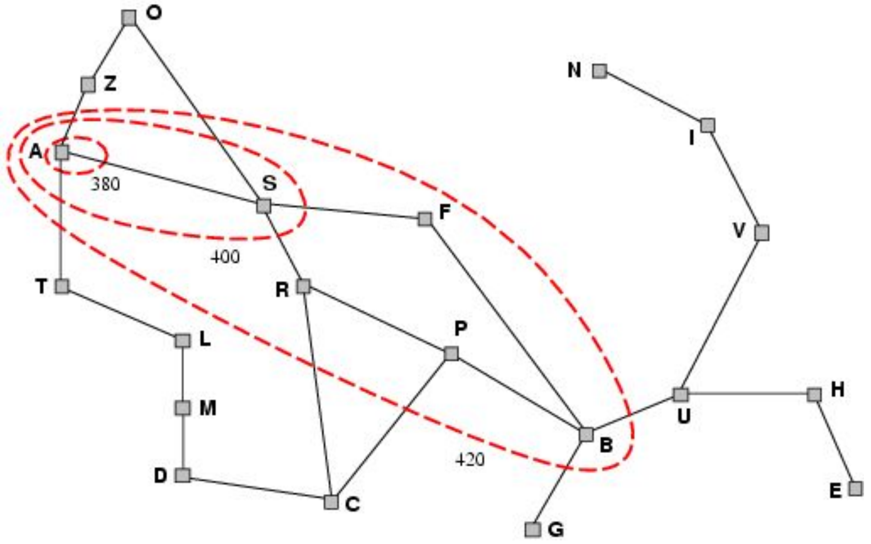


Straight–line distance
to Bucharest

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

Contours of A* Search

- 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*

- Complete? Yes

  (unless there are infinitely many nodes with $f \leq f(G)$;
  can't happen if step-cost $\geq \varepsilon > 0$)

- Time/Space? Exponential $O(b^d)$

  except if: $|h(n) - h^*(n)| \leq O(\log h^*(n))$

- Optimal? Yes

  (with: Tree-Search, admissible heuristic;
  Graph-Search, consistent heuristic)

- *Optimally Efficient?* Yes

  (no optimal algorithm with same heuristic is guaranteed to expand fewer nodes)

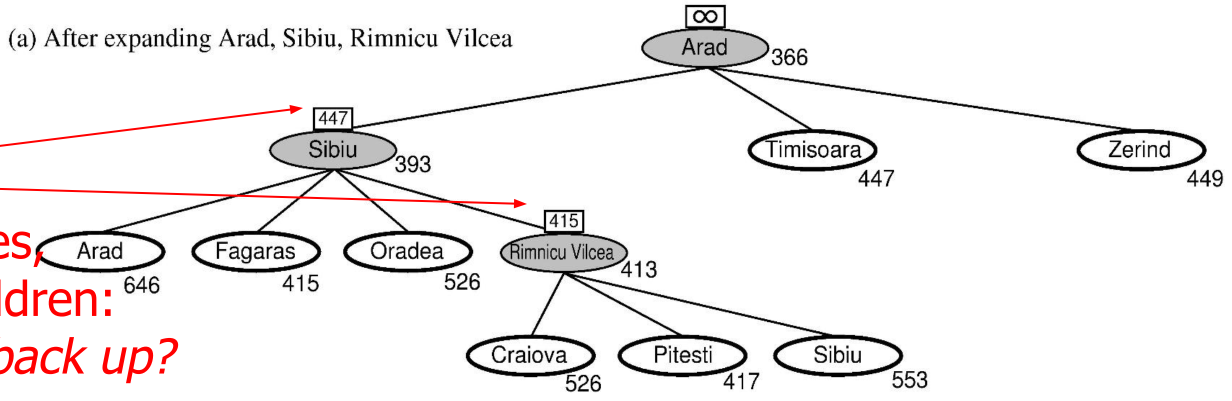
Memory Bounded Heuristic Search: Recursive Best First Search (RBFS)

- How can we solve the memory problem for A* search?

- Idea: Try something like depth first search, but let's not forget everything about the branches we have partially explored.
- *We remember the best f(n) value we have found so far in the branch we are deleting*.

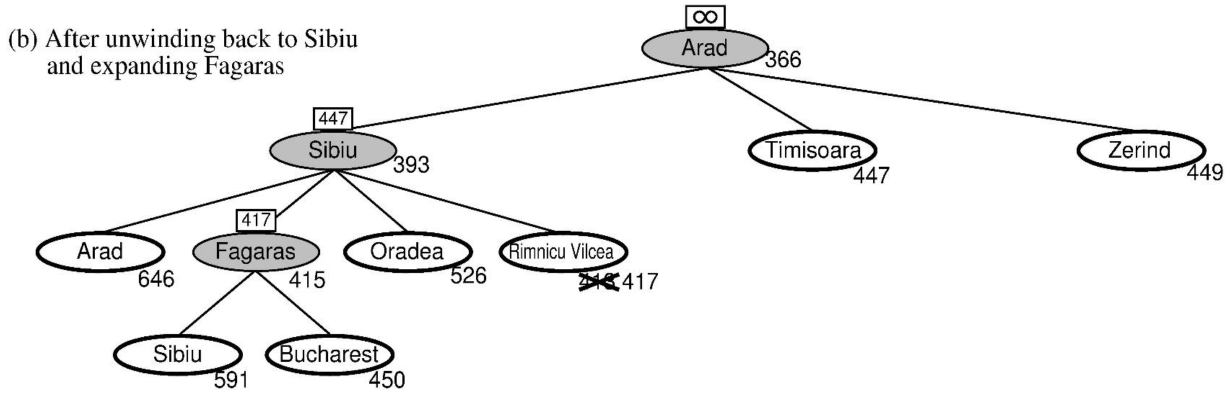
RBFS:

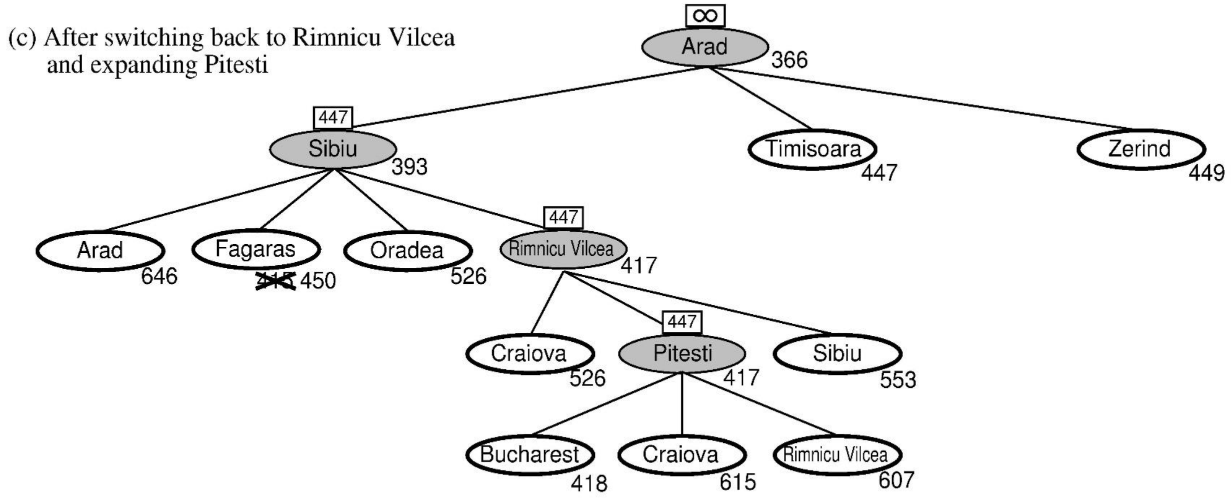
best alternative
over frontier nodes,
which are not children:
i.e. do I want to back up?



RBFS changes its mind
very often in practice.

This is because the
f=g+h become more
accurate (less optimistic)
as we approach the goal.
Hence, higher level nodes
have smaller f-values and
will be explored first.





Problem: We should keep
in memory whatever we can.

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

SMA* pseudocode (not in 2nd edition of R&N)

```
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
```

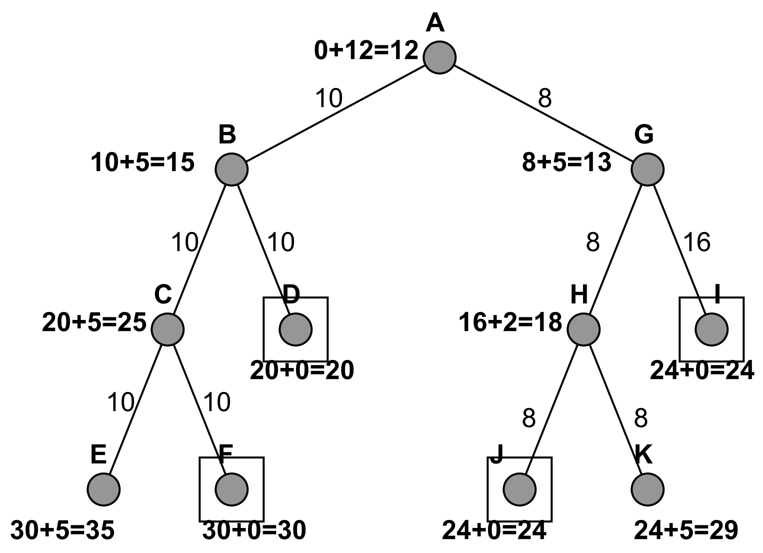
Simple Memory-bounded A* (SMA*)

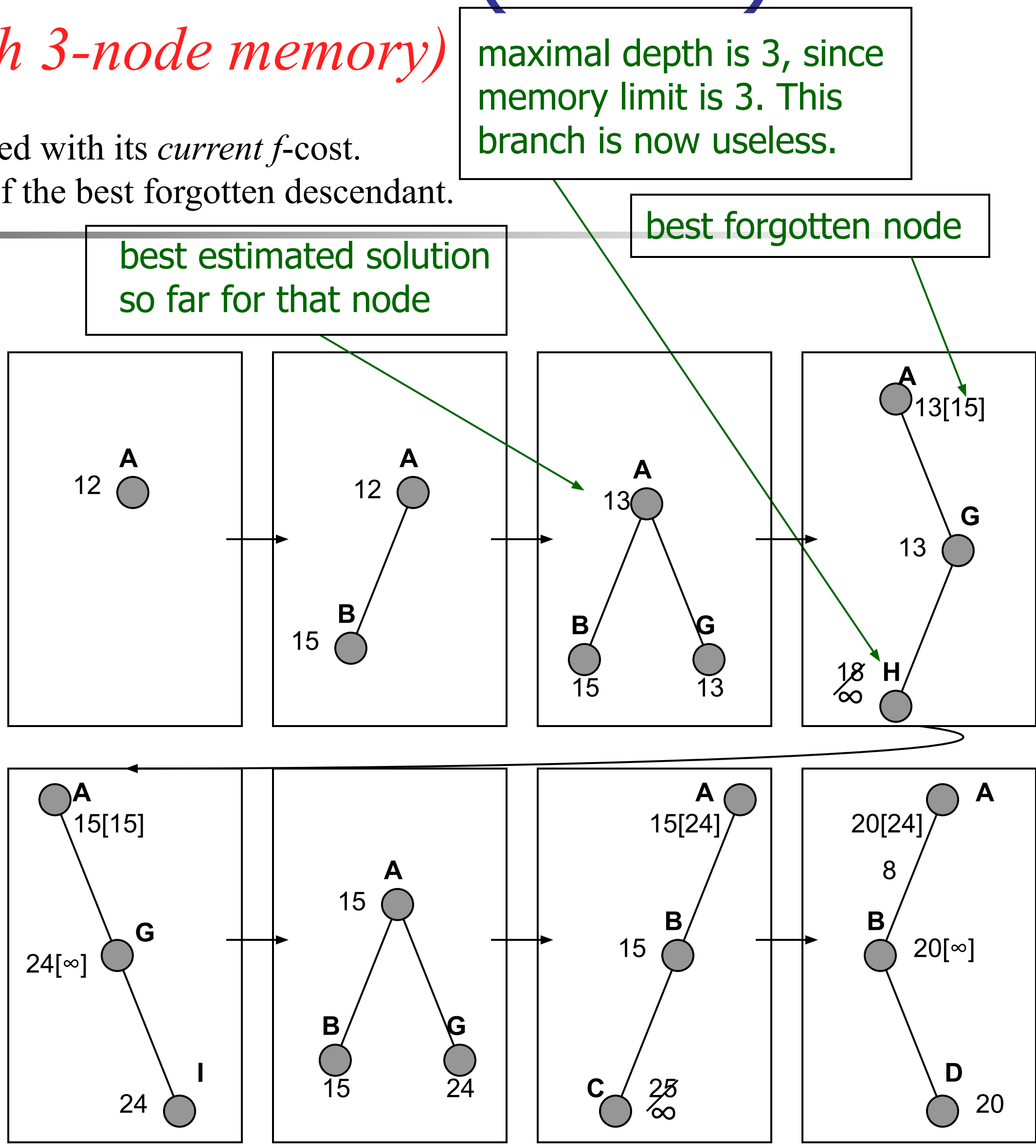
(Example with 3-node memory)

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

Search space

g+h = f   □ = goal



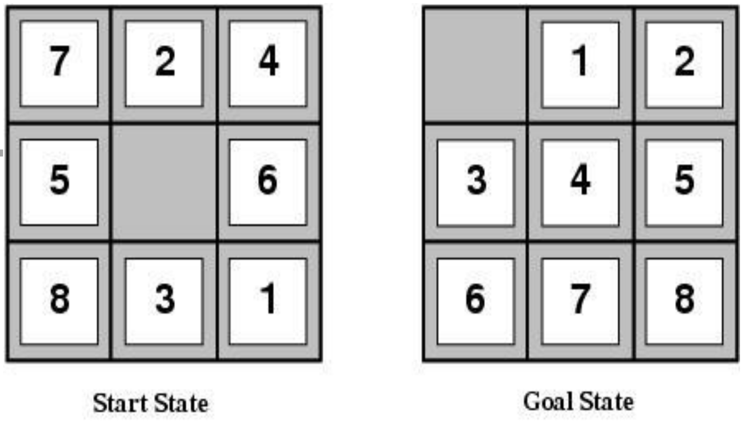


Algorithm can tell you when best solution found within memory constraint is optimal or not.

Memory Bounded A* Search

- The Memory Bounded A* Search is the best of the search algorithms we have seen so far. It uses all its memory to avoid double work and uses smart heuristics to first descend into promising branches of the search-tree.
- If memory not a problem, then plain A* search is easy to code and performs well.

Heuristic functions



- 8-puzzle
  - Avg. solution cost is about 22 steps
  - branching factor ~ 3
  - Exhaustive search to depth 22:
    - 3.1 x 10^{10} states.
  - A good heuristic function can reduce the search process.

- Two commonly used heuristics
  - h_1 = the number of misplaced tiles
    - $h_1(s)=8$
  - h_2 = the sum of the axis-parallel distances of the tiles from their goal positions (manhattan distance).
    - $h_2(s)=3+1+2+2+2+3+3+2=18$

Dominance

- IF $h_2(n) \geq h_1(n)$ for all n (both admissible)
  THEN h_2 dominates h_1
  - h_2 is always better for search than h_1
  - h_2 guarantees to expand no more nodes than does h_1
  - h_2 almost always expands fewer nodes than does h_1

- Typical 8-puzzle 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

Effective branching factor: b*

- Let A* generate *N* nodes to find a goal at depth *d*
  - b* is the branching factor that a uniform tree of depth *d* would have in order to contain *N+1* nodes.

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

$$N+1 = ((b^*)^{d+1} - 1)/(b^* - 1)$$

$$N \approx (b^*)^d \Rightarrow b^* \approx \sqrt[d]{N}$$

  - For sufficiently hard problems, the measure b* usually is fairly constant across different problem instances.
- A good guide to the heuristic’s overall usefulness.
- A good way to compare different heuristics.

Effective Branching Factor Pseudo-code (Binary search)

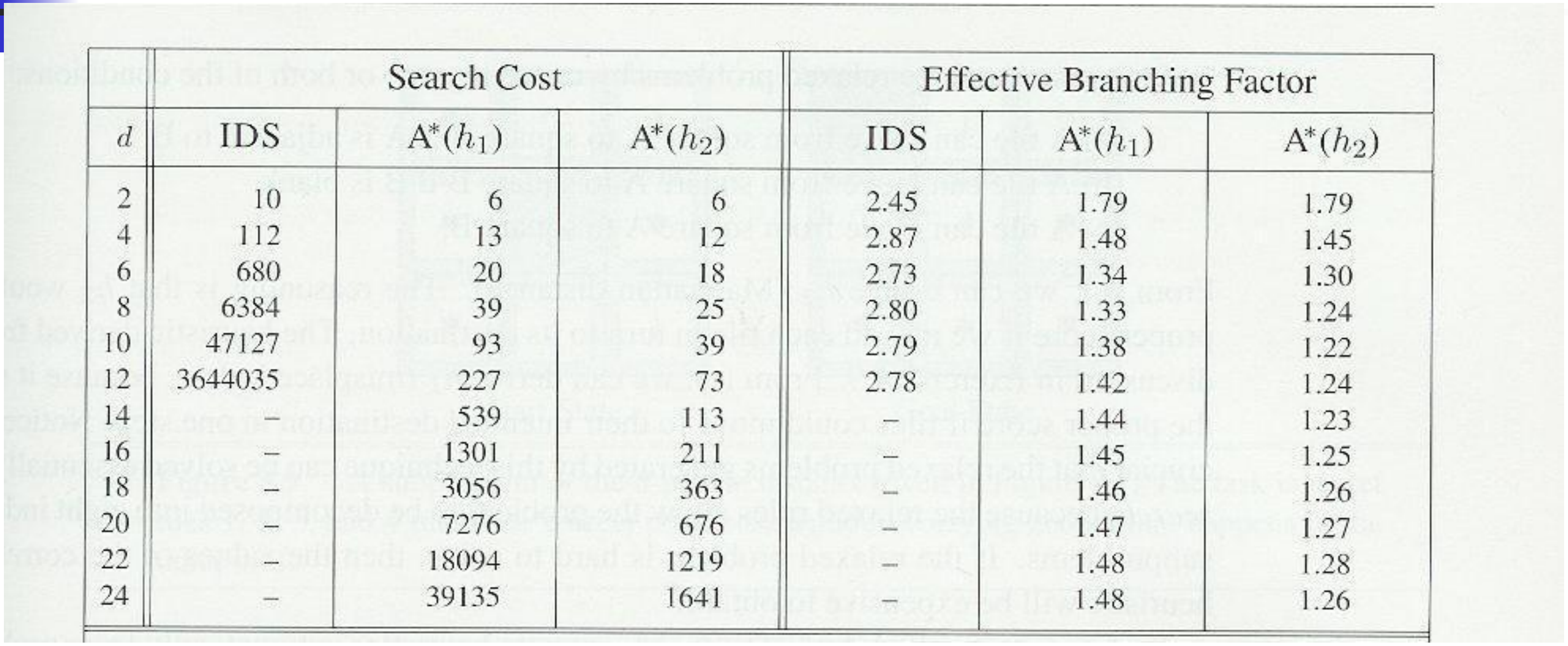
```
PROCEDURE EFFBRANCH (START, END, N, D, DELTA)
  COMMENT DELTA IS A SMALL POSITIVE NUMBER FOR ACCURACY OF RESULT.
  MID := (START + END) / 2.
  IF (END - START < DELTA)
     THEN RETURN (MID).
  TEST := EFFPOLY (MID, D).
  IF (TEST < N+1)
     THEN RETURN (EFFBRANCH (MID, END, N, D, DELTA) )
     ELSE RETURN (EFFBRANCH (START, MID, N, D, DELTA) ).
END EFFBRANCH.

PROCEDURE EFFPOLY (B, D)
  ANSWER = 1.
  TEMP = 1.
  FOR I FROM 1 TO (D-1) DO
     TEMP := TEMP * B.
     ANSWER := ANSWER + TEMP.
  ENDDO.
  RETURN (ANSWER).
END EFFPOLY.
```

- For binary search please see: http://en.wikipedia.org/wiki/Binary_search_algorithm
- An attractive alternative is to use Newton's Method (next lecture) to solve for the root (i.e., $f(b)=0$) of

  $f(b) = 1 + b + \ldots + b^d - (N+1)$

Effectiveness of different heuristics



	Search Cost			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

- Results averaged over random instances of the 8-puzzle

Inventing heuristics via "relaxed problems"

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- 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
- Can be a useful way to generate heuristics
  - E.g., ABSOLVER (Prieditis, 1993) discovered the first useful heuristic for the Rubik's cube puzzle

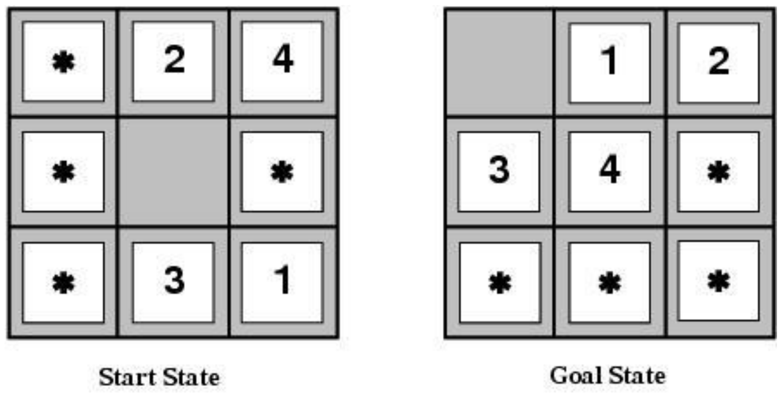
More on heuristics

- **$h(n) = \max\{ h_1(n), h_2(n), \ldots, h_k(n) \}$**
  - Assume all h functions are admissible
  - E.g., h1(n) = # of misplaced tiles
  - E.g., h2(n) = manhattan distance, etc.
  - max chooses least optimistic heuristic (most accurate) at each node

- **$h(n) = w_1 h_1(n) + w_2 h_2(n) + \ldots + w_k h_k(n)$**
  - A convex combination of features
    - Weighted sum of h(n)'s, where weights sum to 1
  - Weights learned via repeated puzzle-solving
  - Try to identify which features are predictive of path cost

Pattern databases

- Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.
- This cost is a lower bound on the cost of the real problem.
- Pattern databases store the exact solution to for every possible subproblem instance.
  - The complete heuristic is constructed using the patterns in the DB



Summary

- Uninformed search methods have uses, also severe limitations
- Heuristics are a structured way to add "smarts" to your search

- Informed (or heuristic) search uses problem-specific heuristics to improve efficiency
  - Best-first, A* (and if needed for memory limits, RBFS, SMA*)
  - Techniques for generating heuristics
  - A* is optimal with admissible (tree)/consistent (graph) heuristics

- Can provide significant speed-ups in practice
  - E.g., on 8-puzzle, speed-up is dramatic
  - Still have worst-case exponential time complexity
  - In AI, "NP-Complete" means "Formally interesting"

- Next lecture topic: local search techniques
  - Hill-climbing, genetic algorithms, simulated annealing, etc.