Heuristic Search

CSE-345: Artificial Intelligence

Heuristic Search

The search techniques we have seen so far...

- Breadth first search
- Uniform cost search
- Depth first search
- Depth limited search
- Iterative Deepening
- Bi-directional Search

uninformed search
blind search

...are all too slow for most real world problems

8-Puzzle

Solution in Node 46

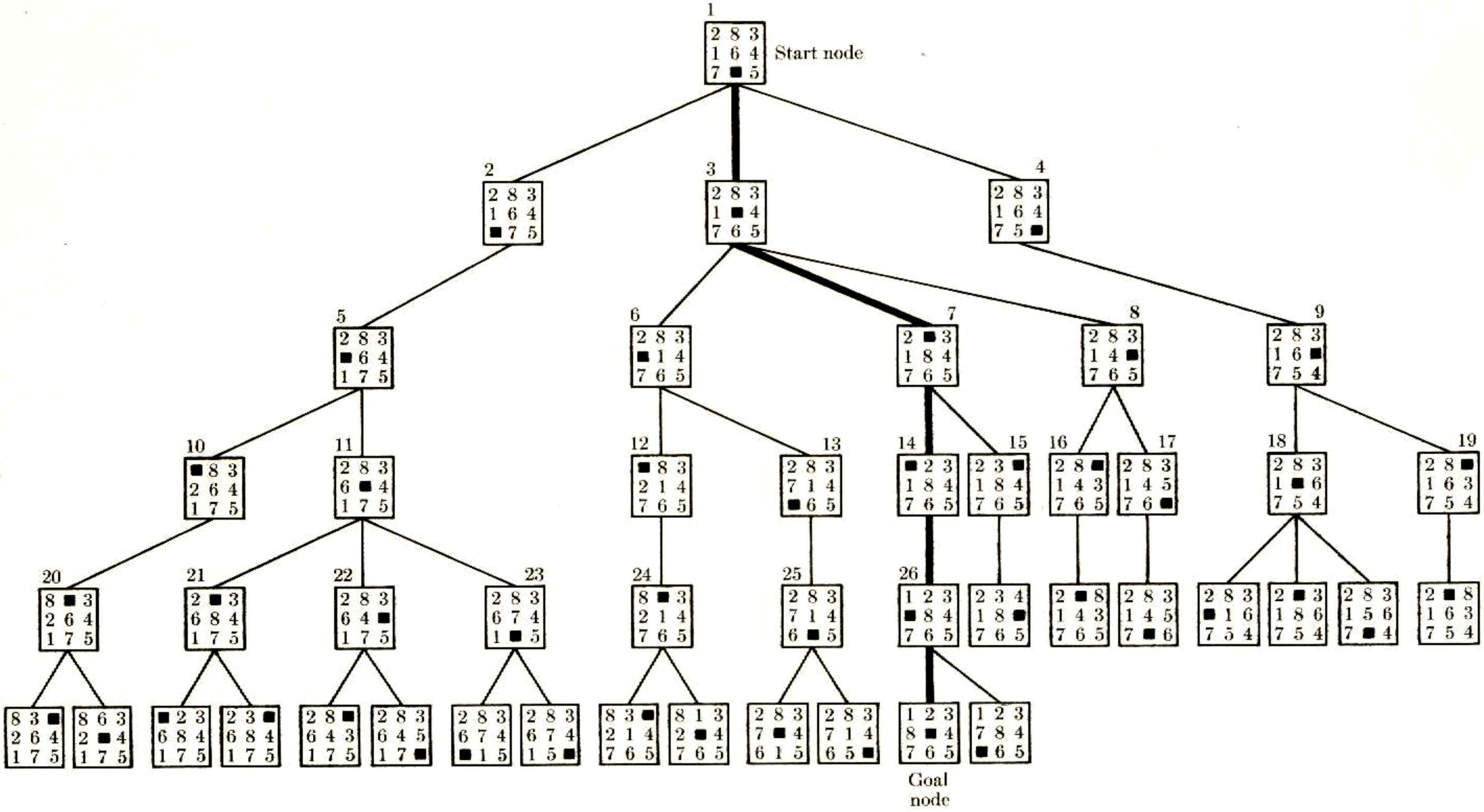


FIG. 3-2 *The tree produced by a breadth-first search.*

8-Puzzle

Solution in Node 31

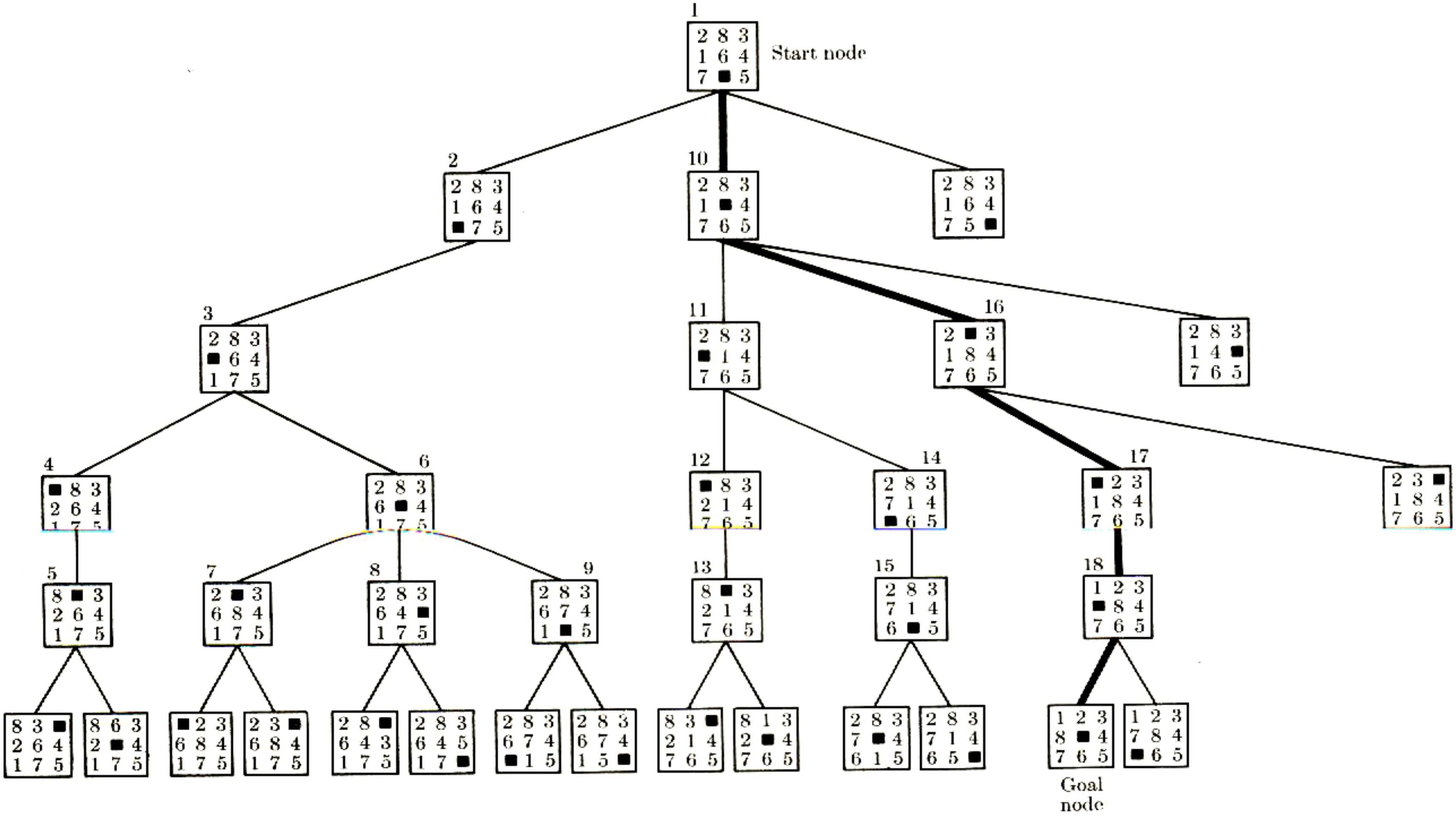
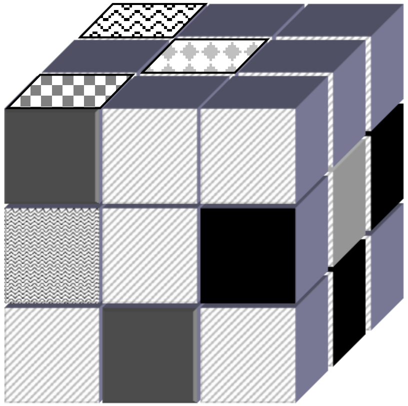


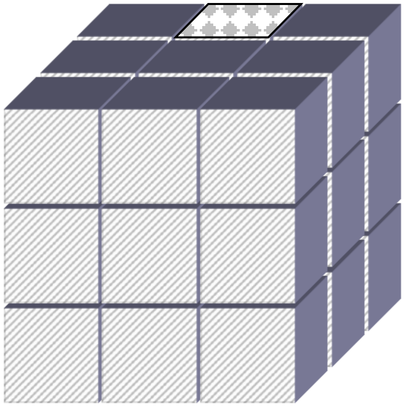
FIG. 3-5 *The tree produced by a depth-first search.*

Sometimes we can tell that some states appear better than others...

7	8	4
3	5	1
6	2	

1	2	3
4	5	6
7		8





...we can use this knowledge of the relative merit of states to guide search

Heuristic Search (informed search)

A **Heuristic** is a function that, when applied to a state, returns a number that is an estimate of the merit of the state, with respect to the goal.

In other words, the heuristic tells us approximately how far the state is from the goal state*.

Note we said “approximately”. Heuristics might underestimate or overestimate the merit of a state. But for reasons which we will see, heuristics that *only* underestimate are very desirable, and are called admissible.

*I.e Smaller numbers are better

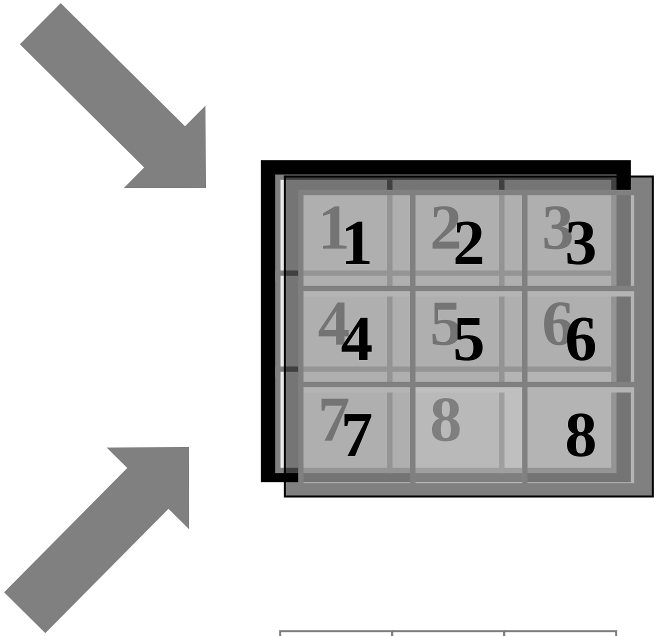
Heuristics for 8-puzzle I

Current State

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

Goal State

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



N	**N**	**N**
N	**N**	**N**
N	**Y**	

- The number of **misplaced tiles** (not including the blank)

In this case, only “**8**” is misplaced, so the heuristic function evaluates to 1.

In other words, the heuristic is *telling* us, that it *thinks* a solution might be available in just 1 more move.

Notation: *h*(n) *h*(current state) = 1

Heuristics for 8-puzzle II

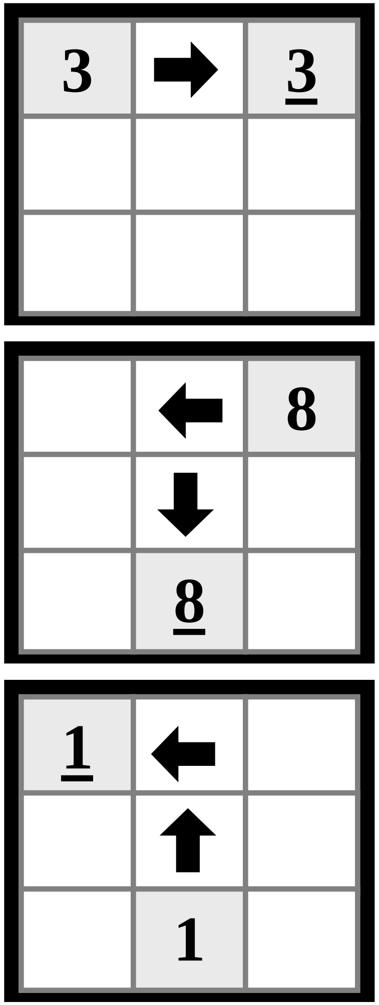
- The **Manhattan Distance** (not including the blank)

Current State

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

Goal State

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



2 spaces

3 spaces

3 spaces

Total 8

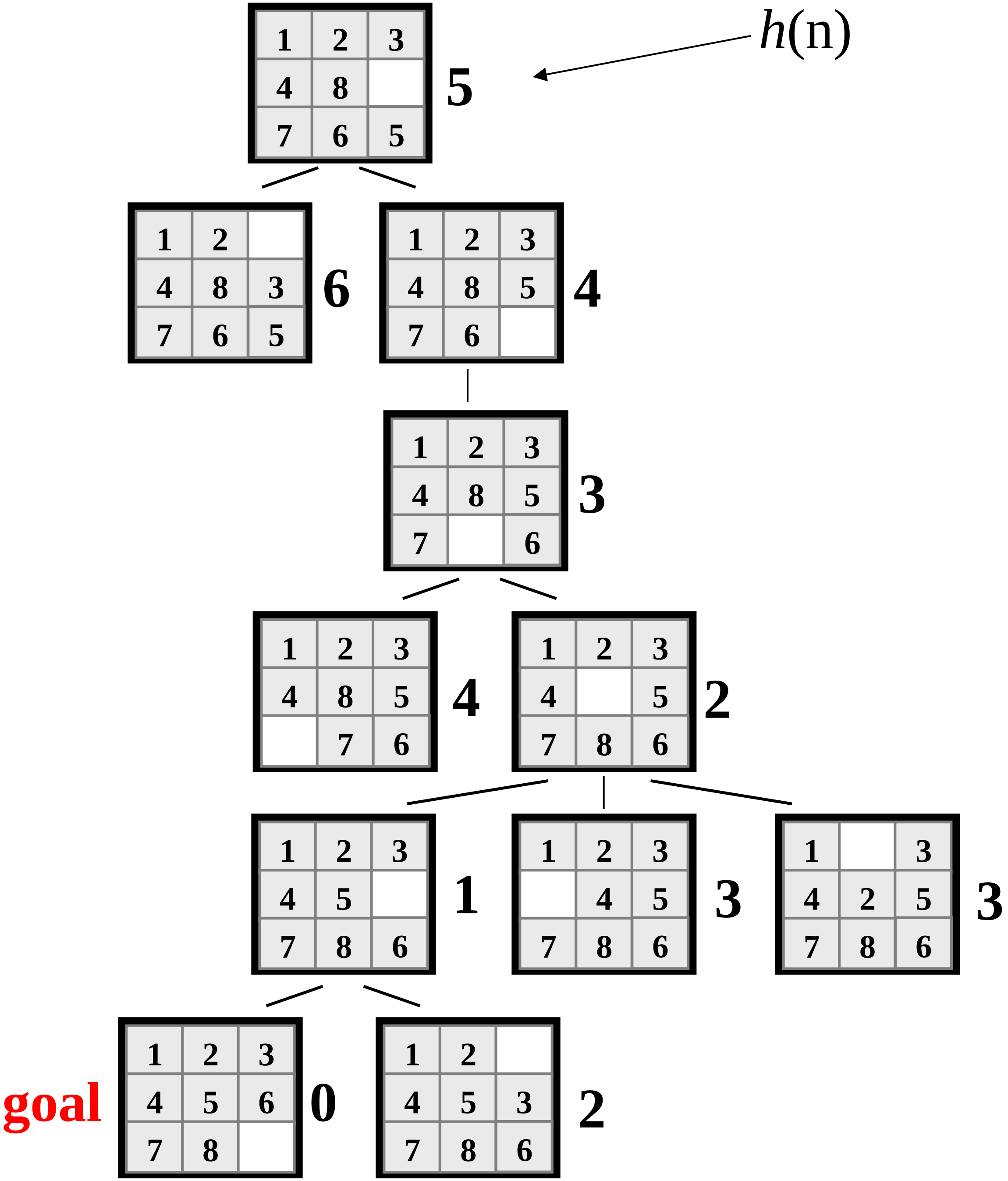
In this case, only the "**3**", "**8**" and "**1**" tiles are misplaced, by 2, 3, and 3 squares respectively, so the heuristic function evaluates to 8.

In other words, the heuristic is *telling* us, that it *thinks* a solution is available in just 8 more moves.

Notation: *h*(n) *h*(current state) = 8

We can use heuristics to guide search.

In this example, the Manhattan Distance heuristic helps us quickly find a solution to the 8-puzzle.



Heuristic Search

Heuristic

- **heuriskein (Greek word)**-to discover
- **original:** eureka

  -heurika (I have found): Archimedes uttering when determining the purity of gold

Heuristic Function

- a way to inform the search about the direction to a goal.
- provides an informed way to guess which neighbor of a node will lead to a goal.

Pros

- Like a tour guide
- Good to the extent that they point in generally interesting direction
- Improve the quality of the paths that are explored
- Using good heuristic, we can hope to get good solutions to hard problem
- It is possible to construct special-purpose heuristics that exploit domain-specific knowledge to solve particular problems

Heuristic Search

- A *heuristic* is a method that

  -might not always find the best solution

  -but is guaranteed to find a good solution in reasonable time.

✓ By sacrificing completeness it increases efficiency.

✓ Useful in solving tough problems which

  – could not be solved any other way.

  – solutions take an infinite time or very long time to compute.

- The *classic* example of heuristic search methods is the travelling salesman problem.

Heuristic Search

❑ **Traveling Salesman Problem**

1. Arbitrarily select a starting city
2. To select the next city, look at all cities not yet visited,

    -Select the one closest to the current city.

    -Go to it next

3. Repeat step 2 until all cities have been visited

Heuristic Function

- A *heuristic function* is a function that maps from problem state descriptions to measure of desirability, usually represented as numbers

  -Which aspects of the problem state are considered?

  -How those aspects are evaluated?

  -The weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution

- Well-designed heuristic functions can play an important part in efficiently guiding a search process toward a solution
- The purpose of a heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available
- The more accurately the heuristic function estimates the true merits of each node in the search tree/graph, the more direct the solution process

Heuristic Search

- Generate –and- test
- **Hill Climbing**
  - Simple Hill Climbing
  - Steepest-Ascent Hill Climbing
  - Simulated Annealing
- **Best-First Search**
  - OR Graphs
  - A* Algorithm
  - Iterative Deepening A* (IDA)
  - Greedy Best-First Search
  - Recursive Best-First Search (RBF)
- **MEMORY BOUNDED SEARCH**
  - Iterative deepening A* search (IDA*)
  - Recursive Best-First Search(RBFS)
  - Memory-bounded A* (MA*)
  - Simplified MA*
- **Problem Reduction**
  - AND-OR Graphs
  - AO* Algorithm
- **Constraint Satisfaction**
- Means-ends Analysis

Generate-and-Test

Algorithm

1. Generate a possible solution.
2. Test to see if this is actually a solution.
3. Quit if a solution has been found. Otherwise, return to step 1.

✓ Acceptable for simple problems.

✘ Inefficient for problems with large space.

Hill Climbing

- A variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space.
- Searching for a goal state = Climbing to the top of a hill
- Generate-and-test + direction to move.
- Heuristic function to estimate how close a given state is to a goal state.

Simple Hill Climbing

Algorithm

1. Evaluate the initial state.

2. Loop until a solution is found or there are no new operators left to be applied:
   - Select and apply a new operator
   - Evaluate the new state:

     goal → quit

     better than current state → new current state

Steepest-Ascent Hill Climbing (Gradient Search)

- Considers all the moves from the current state.
- Selects the best one as the next state.

Steepest-Ascent Hill Climbing (Gradient Search)

Algorithm

1. Evaluate the initial state.

2. Loop until a solution is found or a complete iteration produces no change to current state:

   – SUCC = a state such that any possible successor of the current state will be better than SUCC (the worst state).

   – For each operator that applies to the current state, evaluate the new state:

     goal → quit

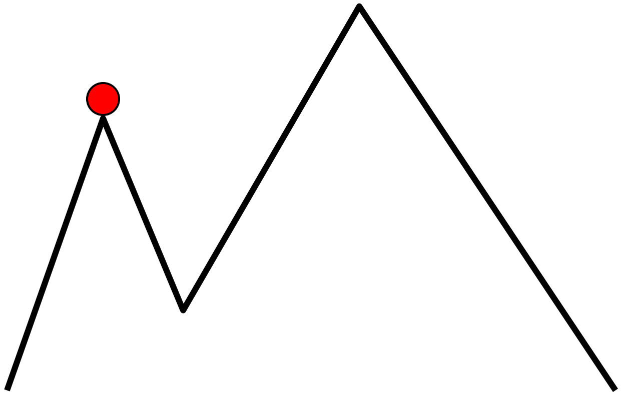
     better than SUCC → set SUCC to this state

   – SUCC is better than the current state → set the current state to SUCC.

Hill Climbing: Disadvantages

Local maximum

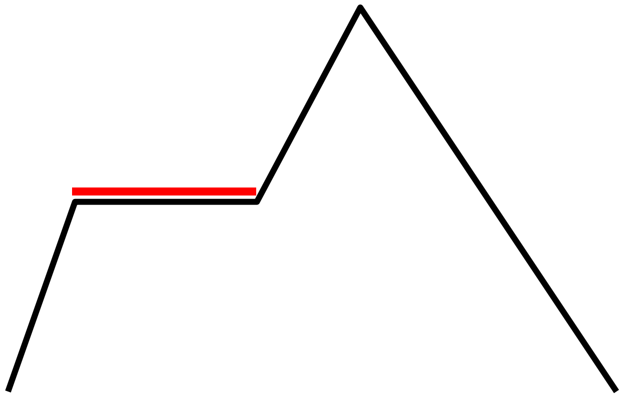
- A state that is better than all of its neighbours, but not better than some other states far away.



Hill Climbing: Disadvantages

Plateau

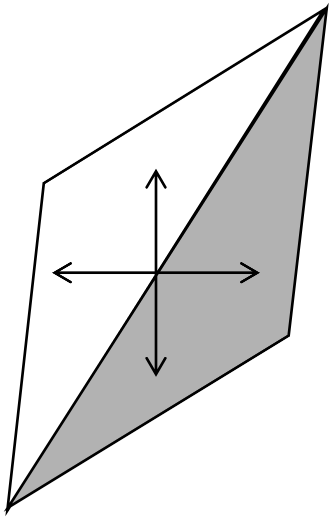
- A flat area of the search space in which all neighbouring states have the same value.



Hill Climbing: Disadvantages

Ridge

➢ The orientation of the high region, compared to the set of available moves, makes it impossible to climb up. However, two moves executed serially may increase the height.



Hill Climbing: Disadvantages

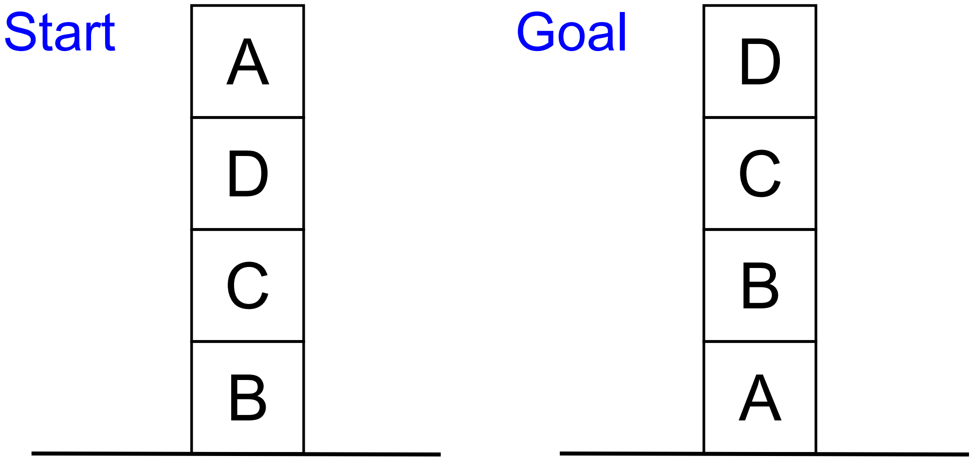
Ways Out

- Backtrack to some earlier node and try going in a different direction.
- Make a big jump to try to get in a new section.
- Moving in several directions at once.

Hill Climbing: Disadvantages

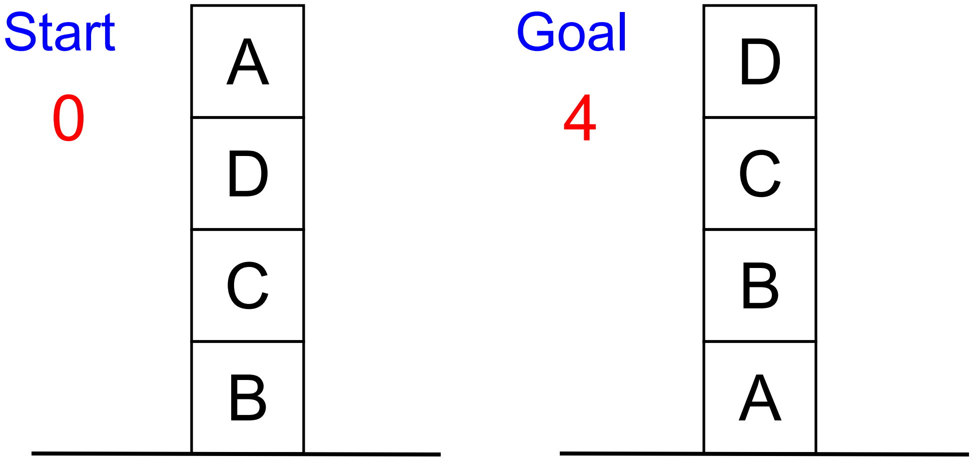
- Hill climbing is a local method:
  Decides what to do next by looking only at the "immediate" consequences of its choices.
- Global information might be encoded in heuristic functions.

Hill Climbing: Disadvantages



Blocks World

Hill Climbing: Disadvantages



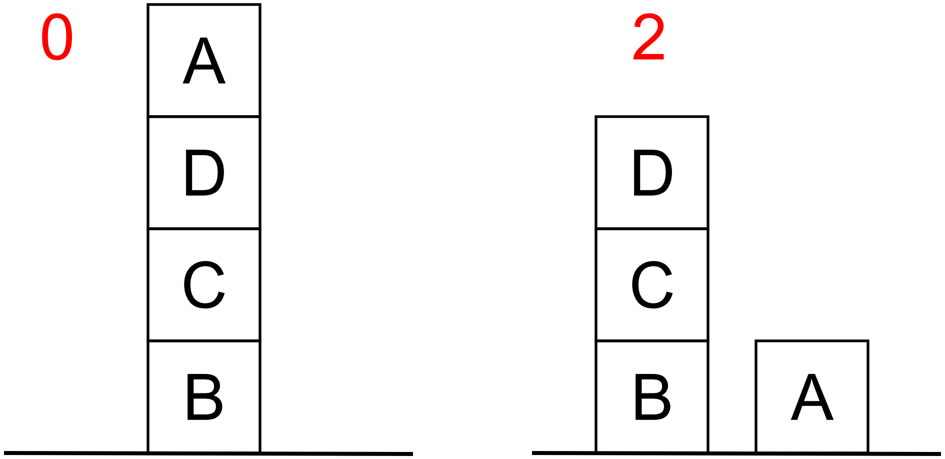
Blocks World

Local heuristic:

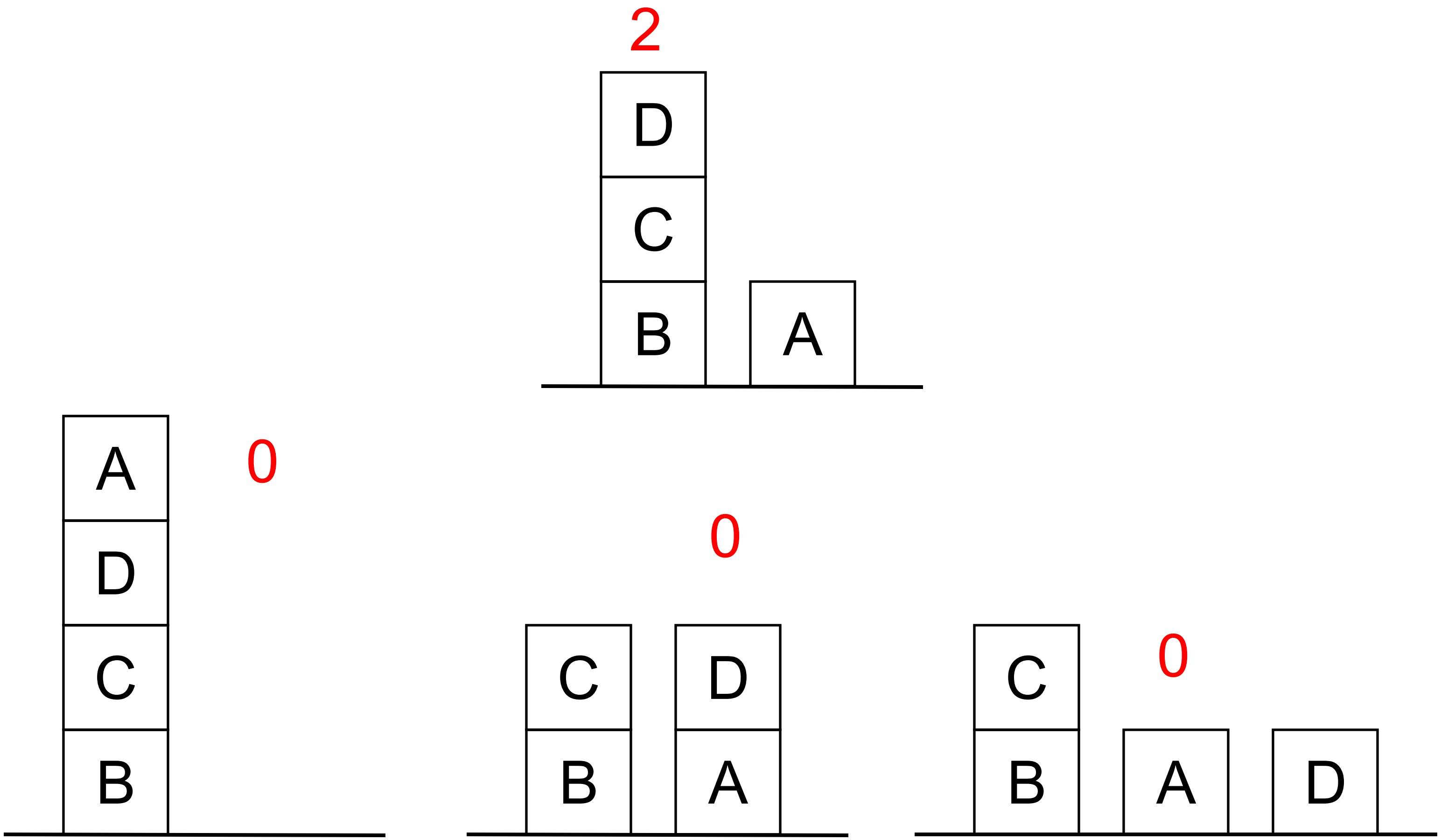
+1 for each block that is resting on the thing it is supposed to be resting on.

–1 for each block that is resting on a wrong thing.

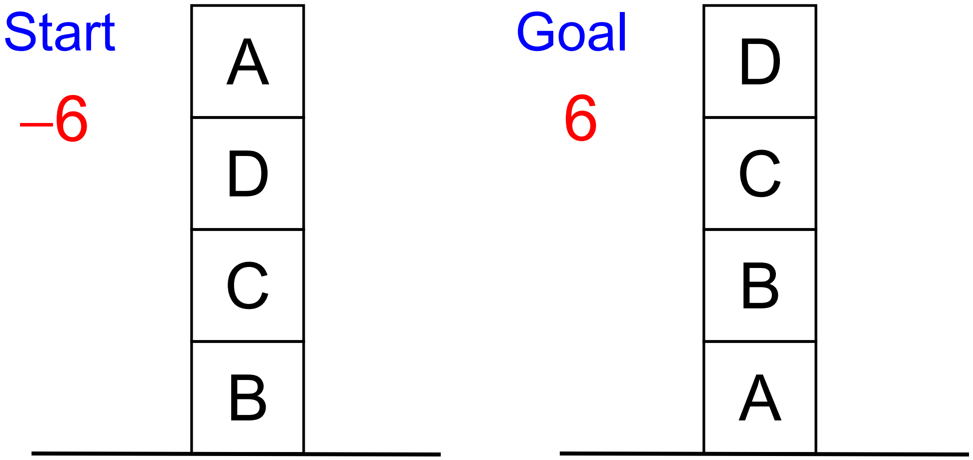
Hill Climbing: Disadvantages



Hill Climbing: Disadvantages



Hill Climbing: Disadvantages



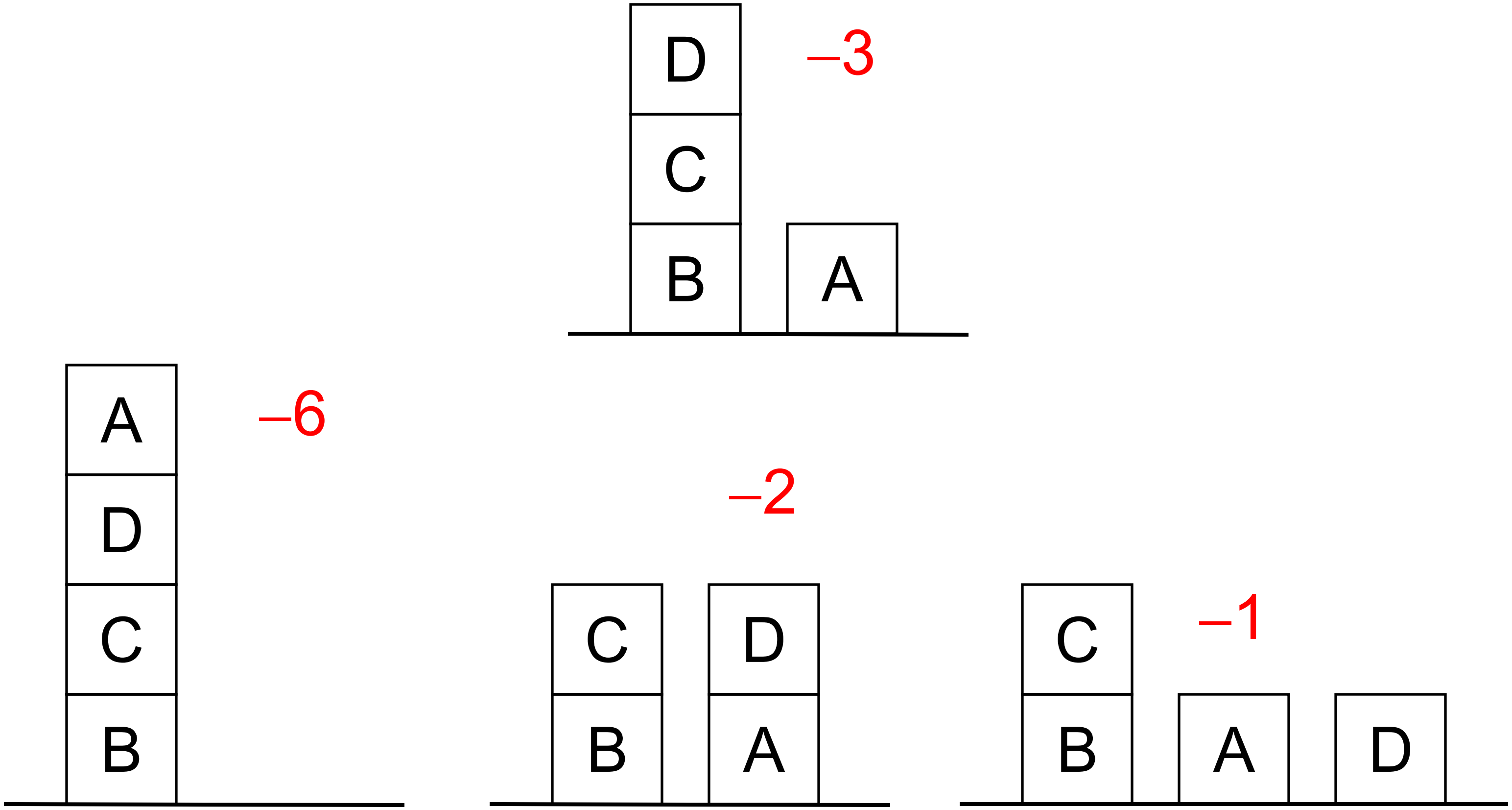
Blocks World

Global heuristic:

For each block that has the correct support structure: +1 to every block in the support structure.

For each block that has a wrong support structure: –1 to every block in the support structure.

Hill Climbing: Disadvantages



Hill Climbing: Conclusion

- Can be very inefficient in a large, rough problem space.
- Global heuristic may have to pay for computational complexity.
- Often useful when combined with other methods.

Simulated Annealing

- A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.
- To do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state.
- Lowering the chances of getting caught at a local maximum, or plateau, or a ridge.

Simulated Annealing

Physical Annealing

- Physical substances are melted and then gradually cooled until some solid state is reached.
- The goal is to produce a minimal-energy state.
- Annealing schedule: if the temperature is lowered sufficiently slowly, then the goal will be attained.
- Nevertheless, there is some probability for a transition to a higher energy state: $p=e^{-\Delta E/kT}$.

ΔE : positive change in the energy level

T : temperature

k : Boltzmann' s Constant

Simulated Annealing

Algorithm

1. Evaluate the initial state.
2. Loop until a solution is found or there are no new operators left to be applied:
   - – Set T according to an annealing schedule
   - – Selects and applies a new operator
   - – Evaluate the new state:

     goal → quit

     ΔE = Val(current state) – Val(new state)

     $\Delta E < 0$ → new current state

     else → new current state with probability $e^{-\Delta E/kT}$.

Difference from Simple Hill Climbing

1. Annealing schedule must be maintained
2. Moves to worse states may be accepted

- ➢ It is a good idea to maintain, current state, the best state found so far.
- ➢ Then, if the final state is worse than that earlier state, the earlier state is still available

Best-First Search

- A way of Combining the advantage of both DFS and BFS into a single method.

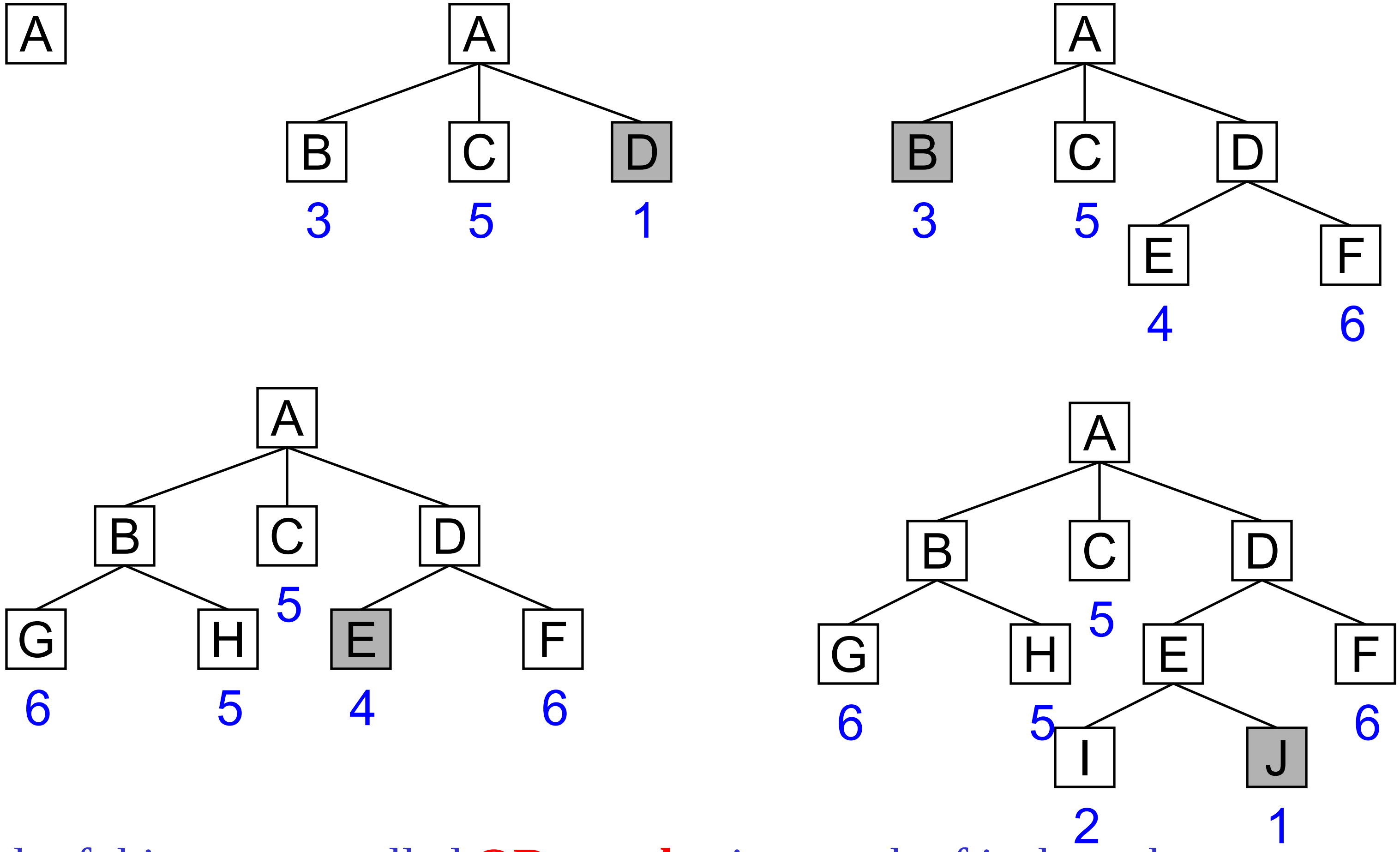
Best-First Search: OR Graphs

- Depth-first search: not all competing branches having to be expanded.

- Breadth-first search: not getting trapped on dead-end paths.

⇒ Combining the two is to follow a single path at a time, but switch paths whenever some competing path look more promising than the current one.

Best-First Search: OR Graphs



A graph of this sort are called **OR graph**, since each of its branches represent an alternative problem-solving path.

Best-First Search: OR Graphs

- OPEN: nodes that have been generated, but have not examined.

  This is organized as a priority queue.

- CLOSED: nodes that have already been examined.

  Whenever a new node is generated, check whether it has been generated before.

Best-First Search

Algorithm

1. OPEN = {initial state}.
2. Loop until a goal is found or there are no nodes left in OPEN:
   - Pick the best node in OPEN
   - Generate its successors
   - For each successor:

     new → evaluate it, add it to OPEN, record its parent

     generated before → change parent, update successors

Best-First Search

- Some authors have used "best-first search" to refer specifically to a search with a heuristic that attempts to predict how close the end of a path is to a solution, so that paths which are judged to be closer to a solution are extended first.
- This specific type of search is called **greedy best-first search**
- Special cases:
  - greedy best-first search
  - A^* search

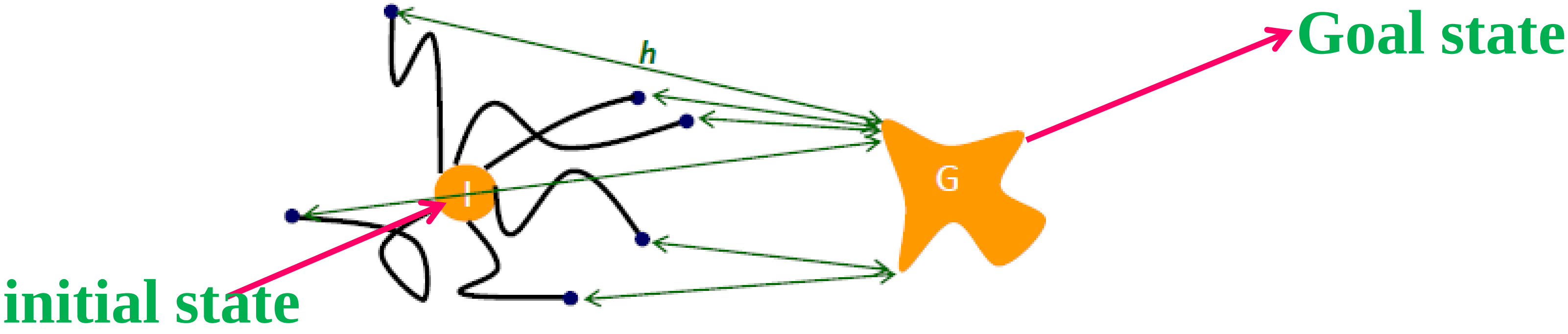
Greedy Best-First Search

- 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

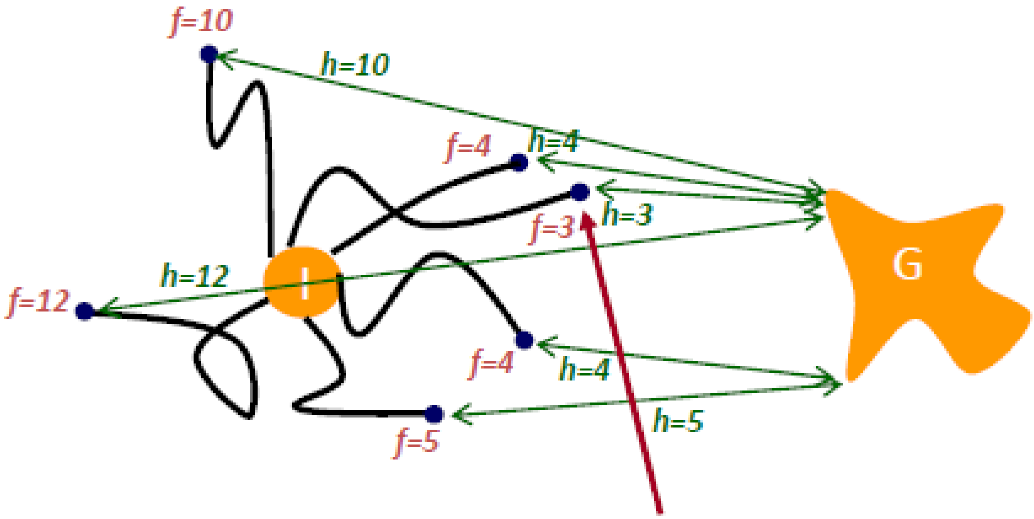


Goal state

initial state

Greedy Best-First Search

- *f(n) = h(n)* : *Expand node that appears to be* closest to the goal



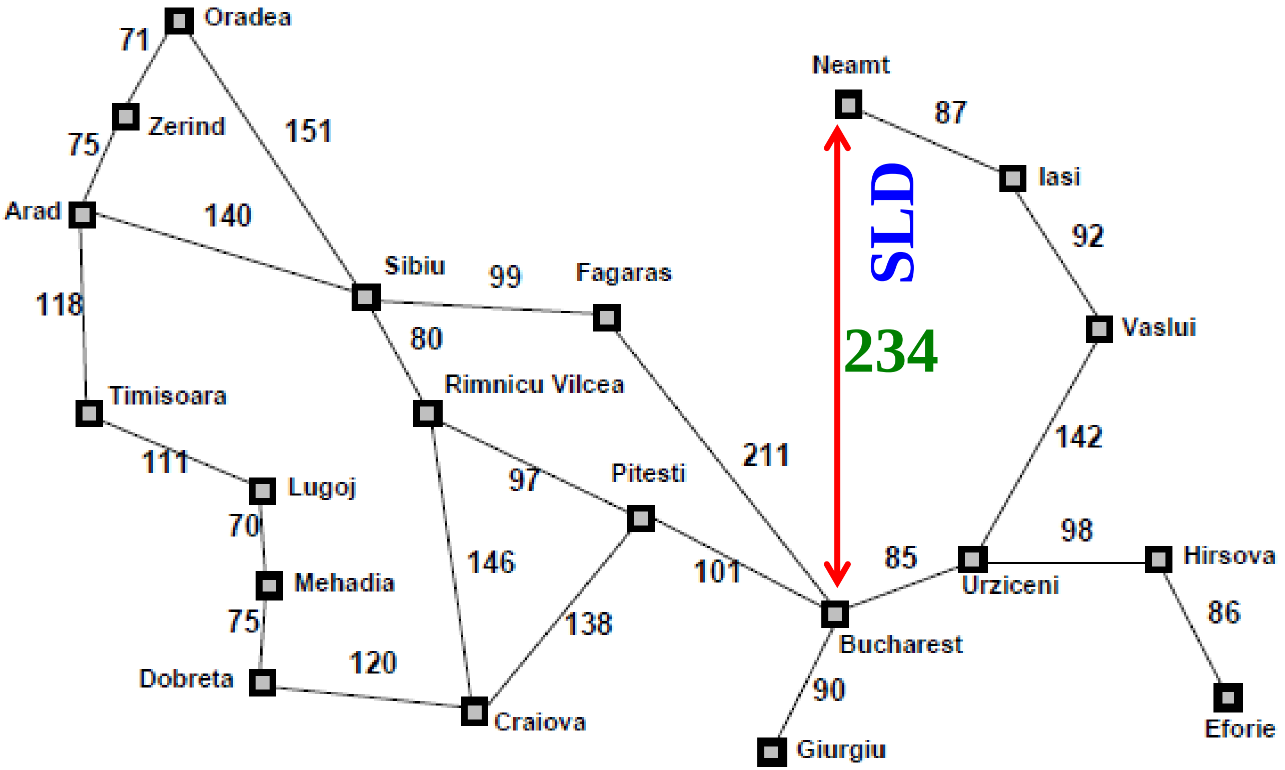
This node would be expanded next

Greedy Best-First Tree Search Example

Go from Arad to Bucharest

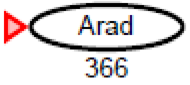
- *f(n)* = $h_{SLD}(n)$ = straight-line distance from *n STATE to Bucharest*

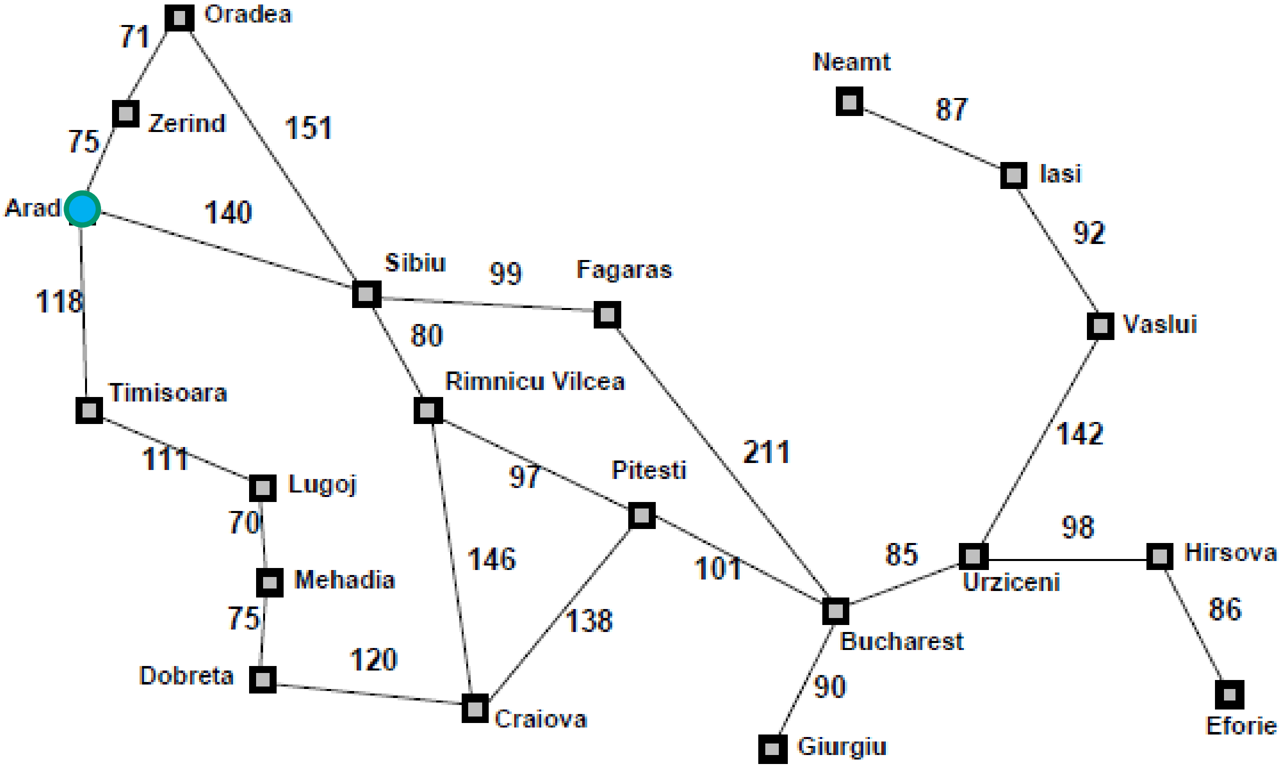
h (n)



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 Tree Search Example



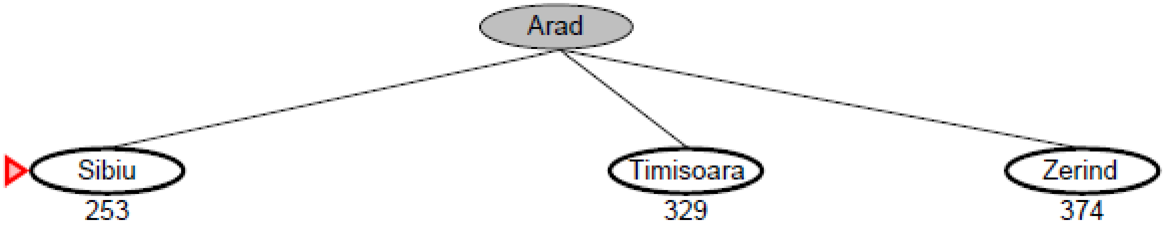


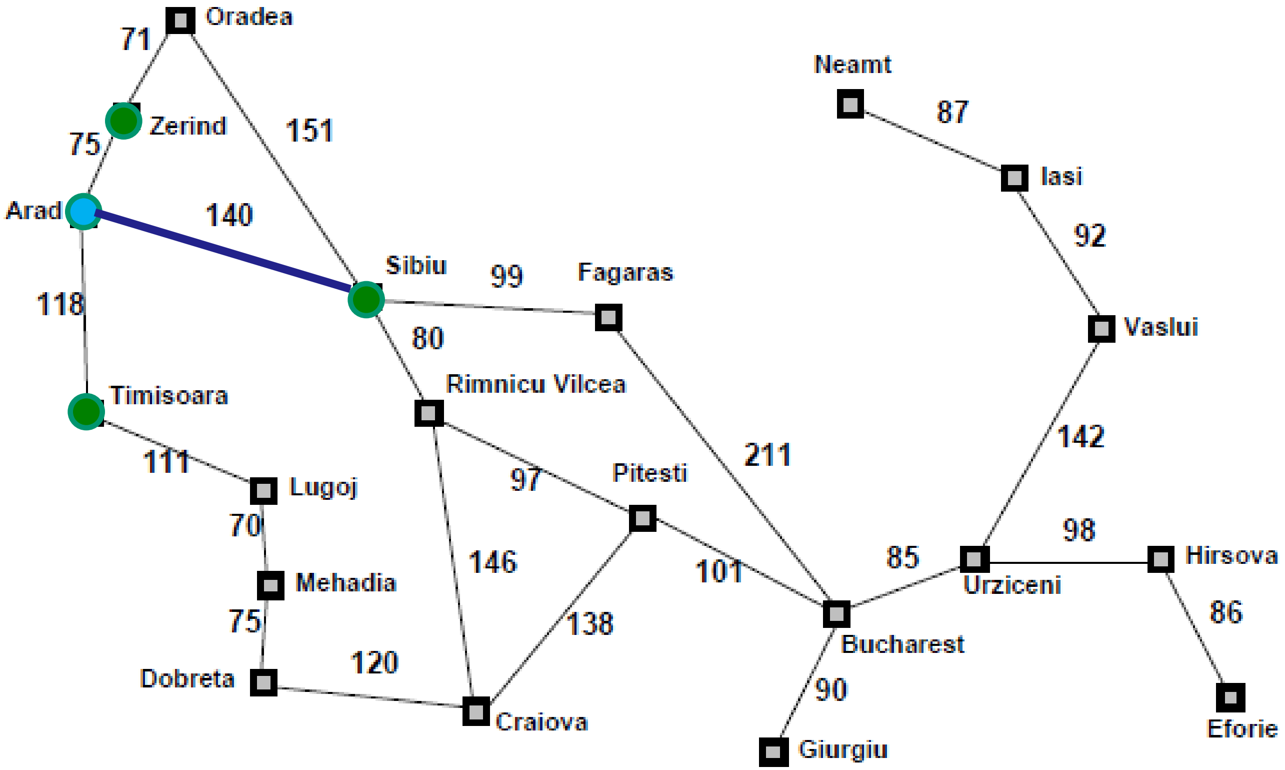
h (n)

Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Tree Search Example

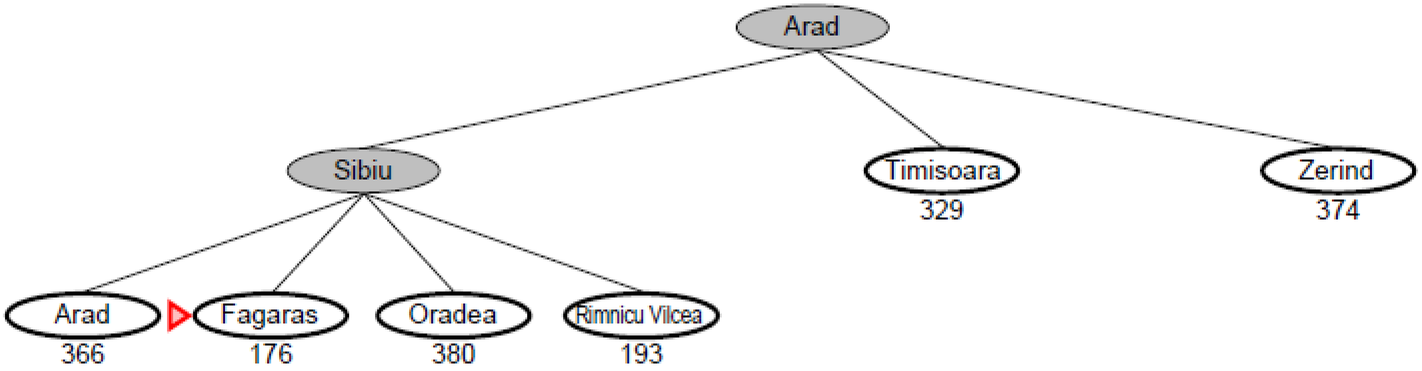


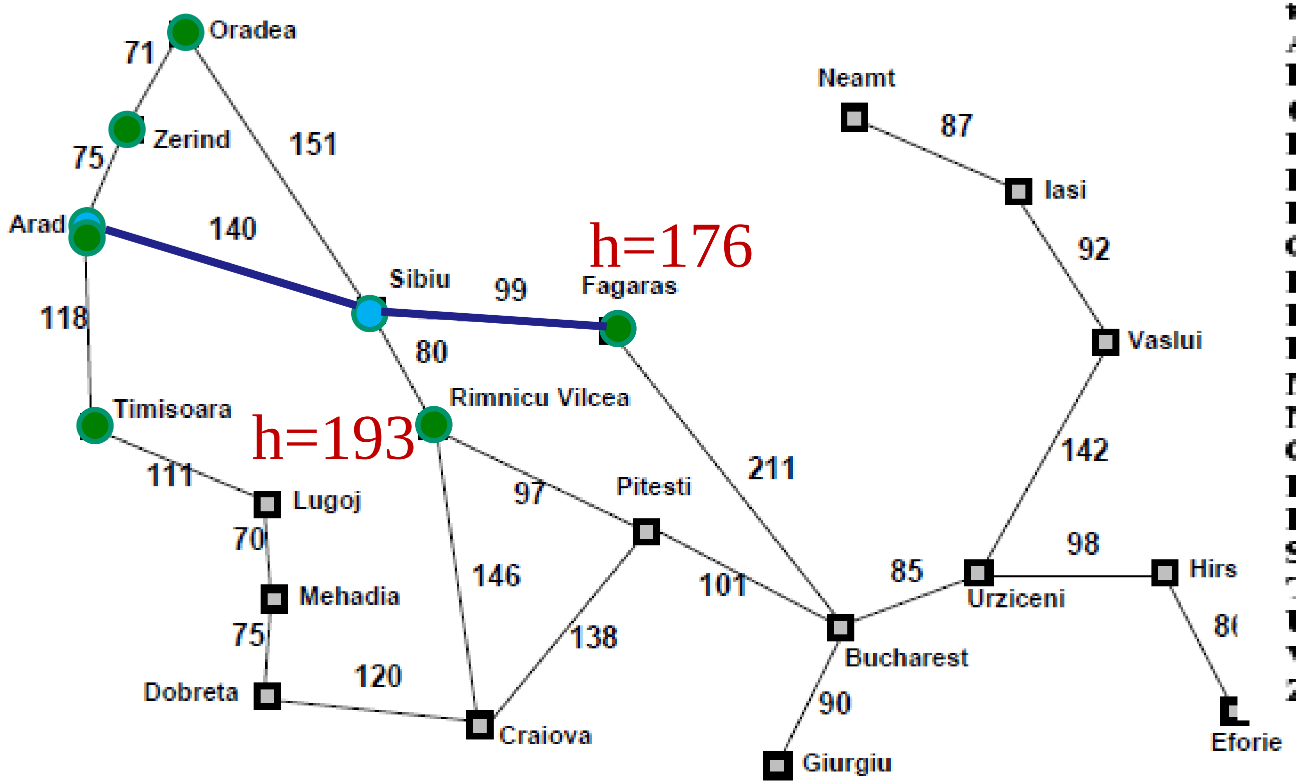


h (n)

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 Tree Search Example



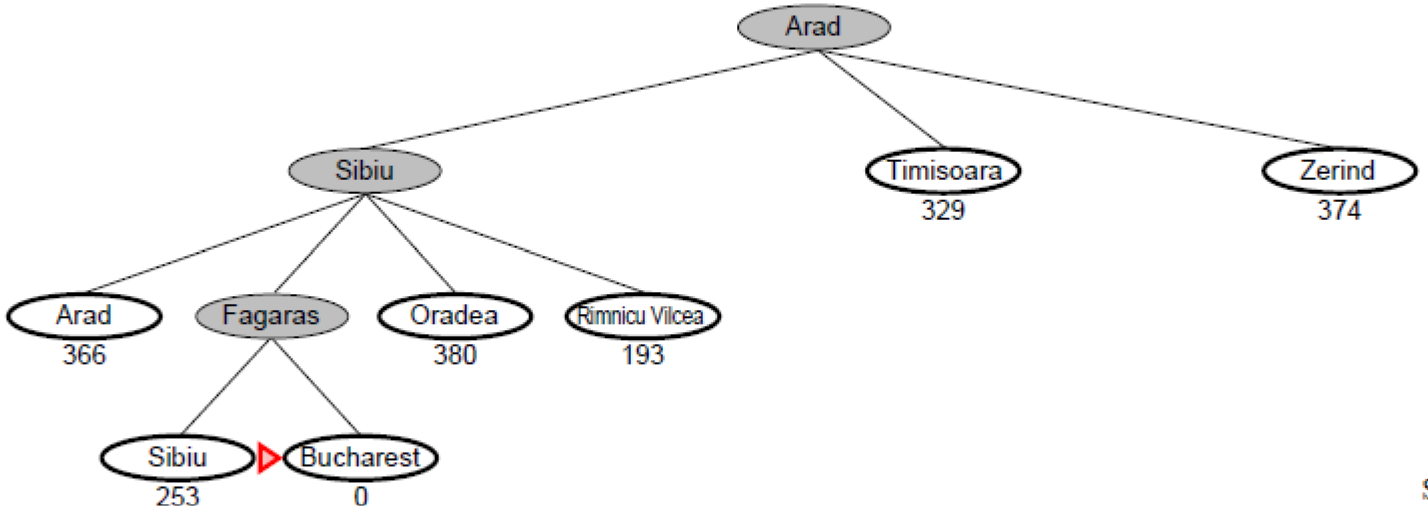


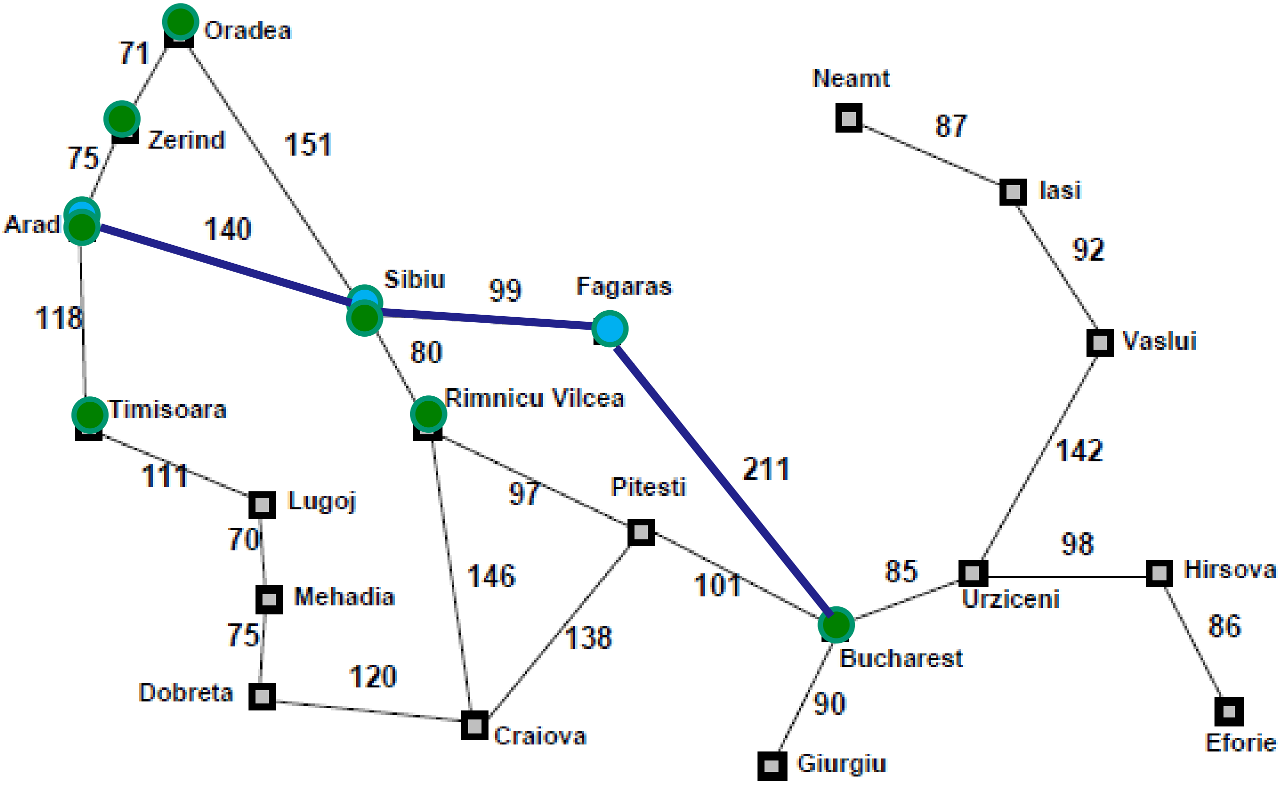
h (n)

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

Example

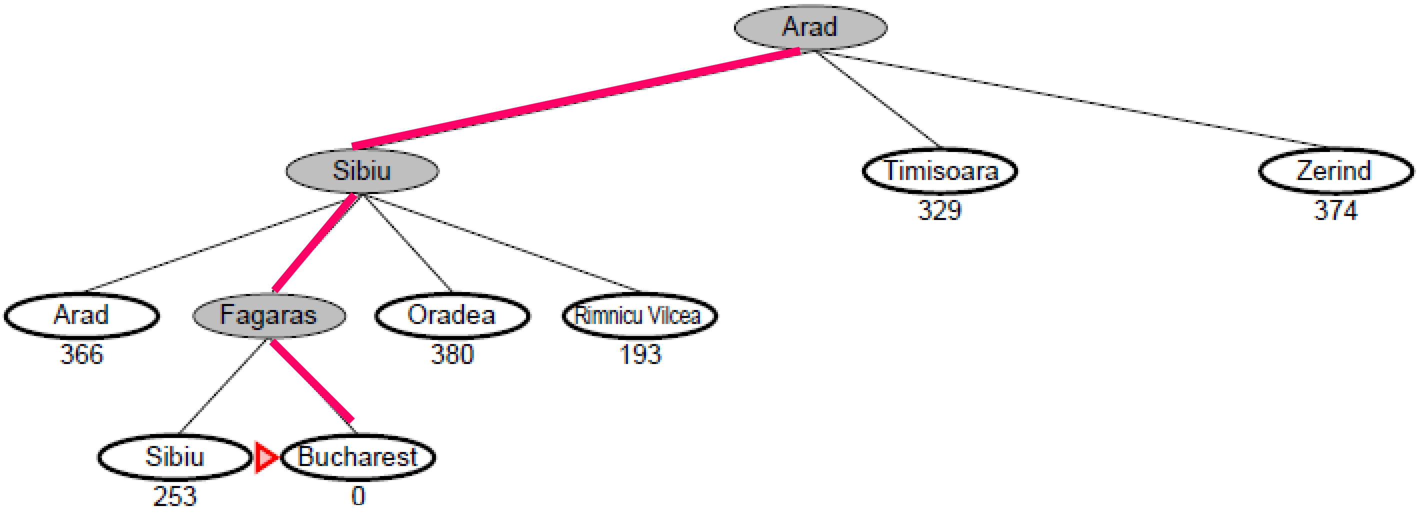


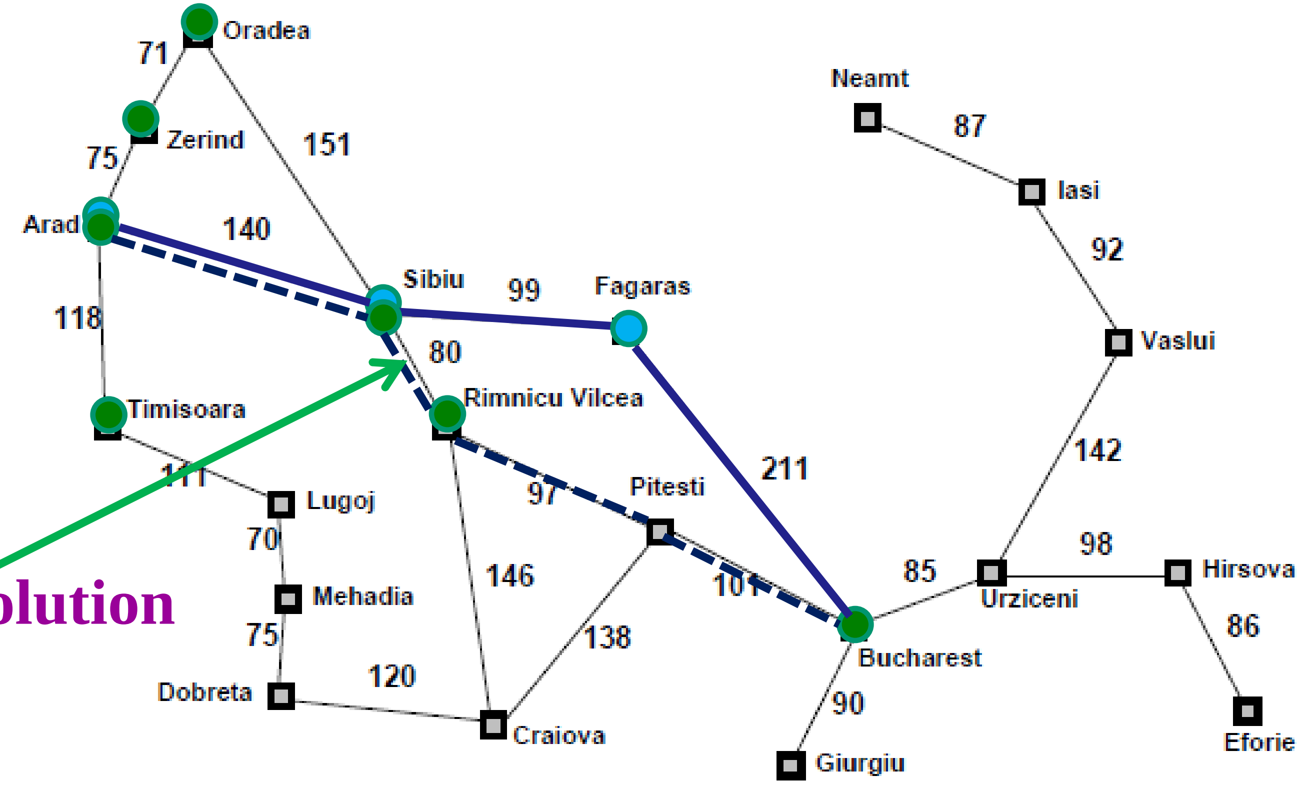


h (n)

Straight−line distance
to Bucharest

Arad	366	⇐
Bucharest	0	⬅
Craiova	160	
Dobreta	242	
Eforie	161	
Fagaras	178	⇐
Giurgiu	77	
Hirsova	151	
Iasi	226	
Lugoj	244	
Mehadia	241	
Neamt	234	
Oradea	380	⇐
Pitesti	98	
Rimnicu Vilcea	193	⇐
Sibiu	253	⬅
Timisoara	329	⬅
Urziceni	80	
Vaslui	199	
Zerind	374	⬅





Solution turns out not to be optimal

Optimal solution

Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Complete in finite space with repeated-state checking
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

Greedy Best-First Search

- Greedy search:
  h(n) = estimated cost of the cheapest path from node n to a goal state.

  Neither optimal nor complete

- Uniform-cost search:
  g(n) = cost of the cheapest path from the initial state to node n.

  Optimal and complete, but very inefficient

A* search

- Idea: avoid expanding paths that are already expensive

- include cost of reaching node

❑Evaluation function *f(n) = g(n) + h(n)*

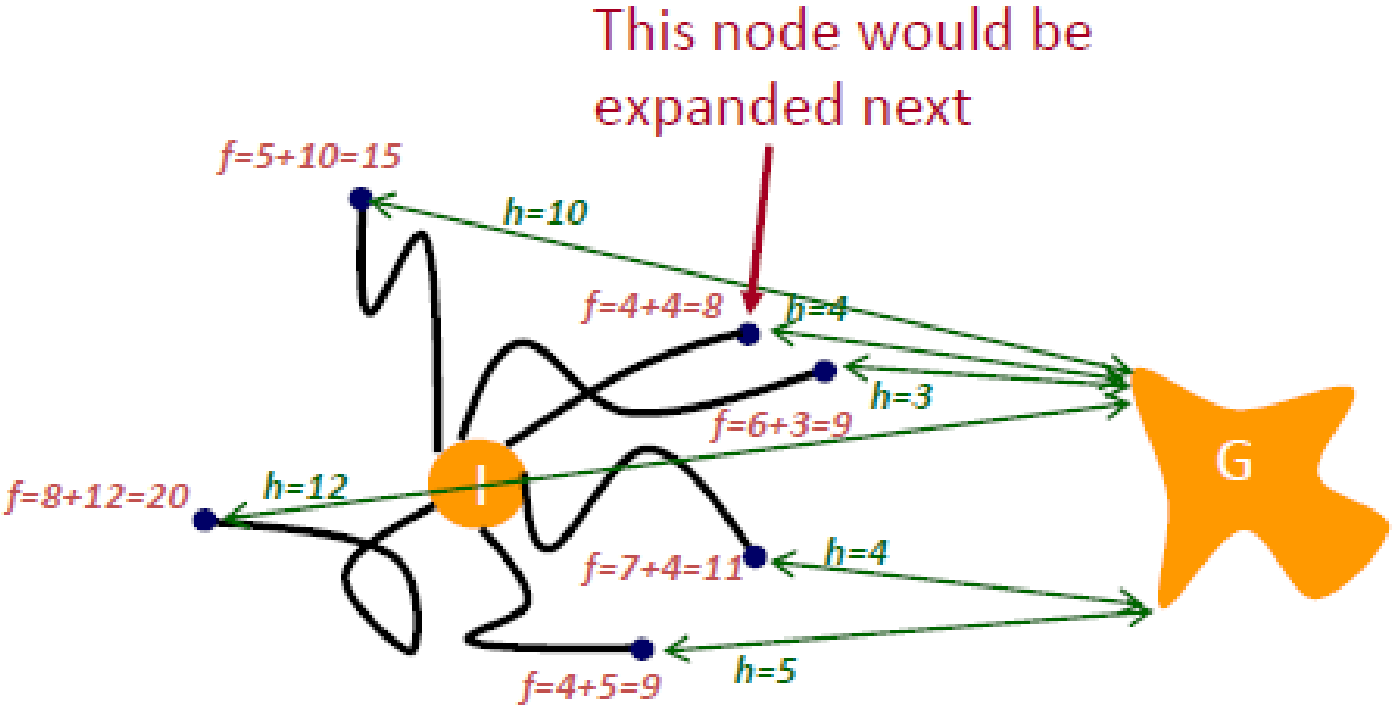
-*g(n) = cost of reaching n*

-*h(n) = estimated cost of reaching goal from state* of *n*

-*f(n) = estimated cost of the cheapest path to a* goal state that goes through path of *n*

A* Search

- *f(n) = g(n) + h(n)* : *Expand node that appears to* be on cheapest paths to the goal

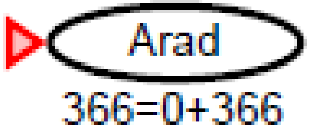


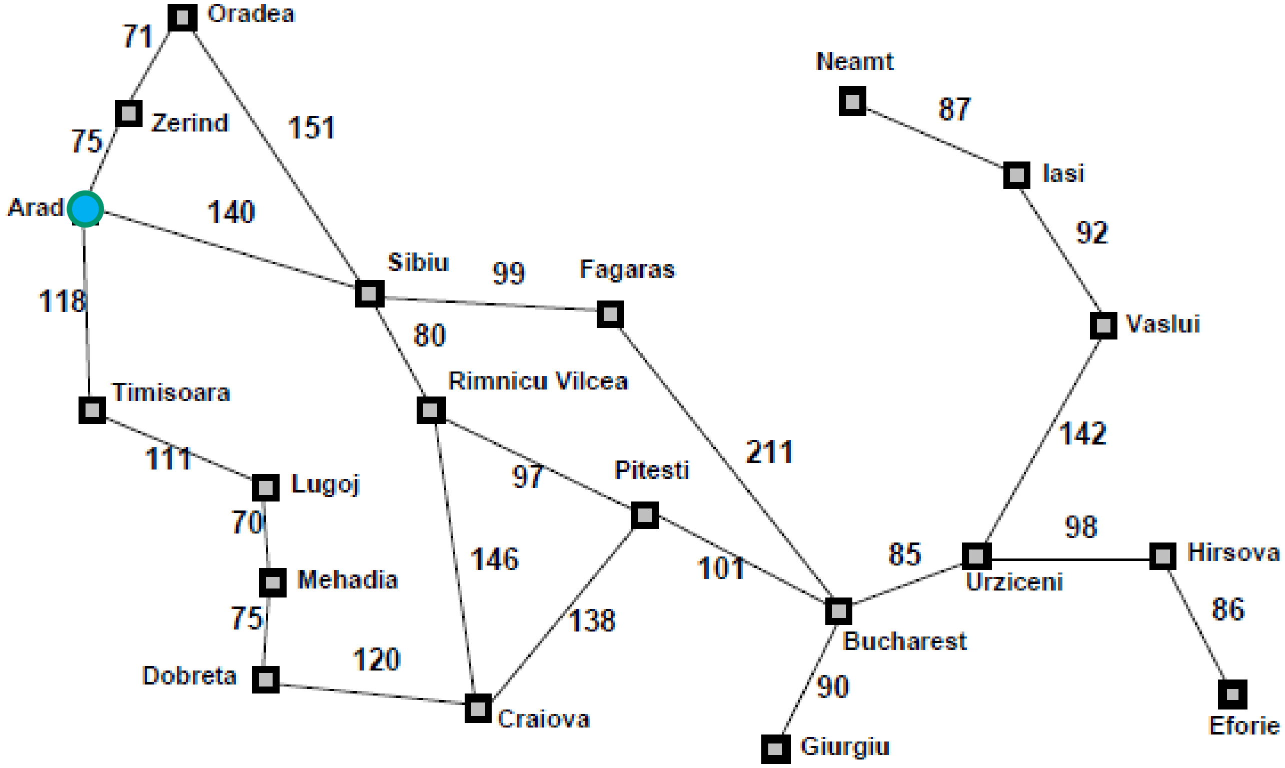
A* search example

Go from Arad to Bucharest

- $f(n) = g(n) + h_{SLD}(n)$

A* search example



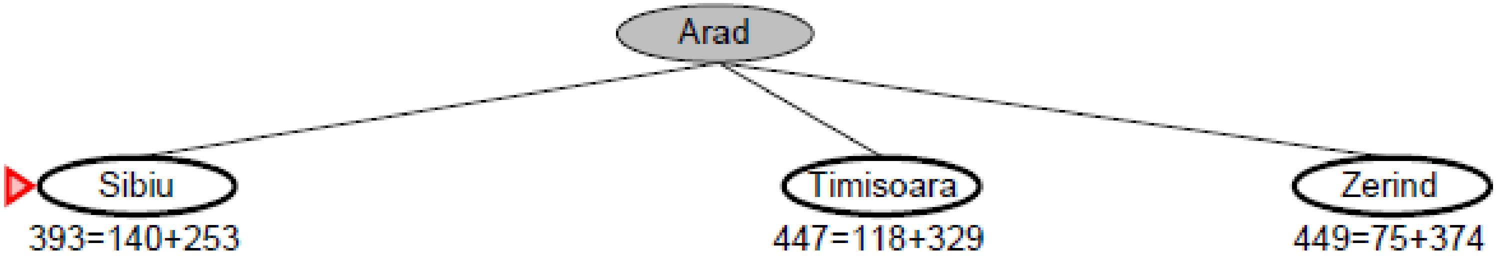


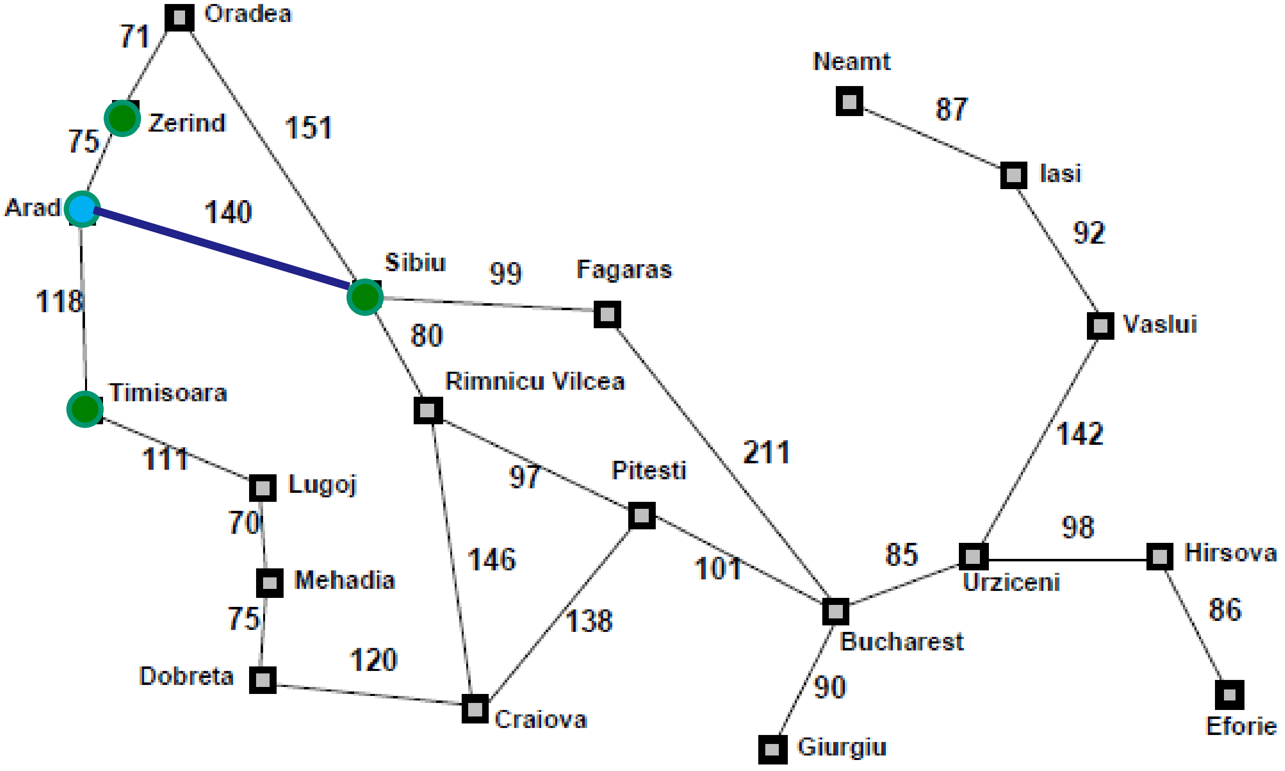
h(n)

Straight–line distance
to Bucharest

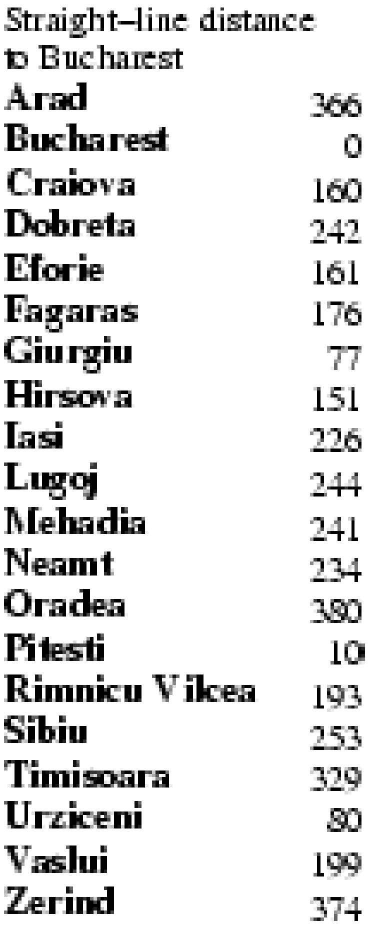
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example





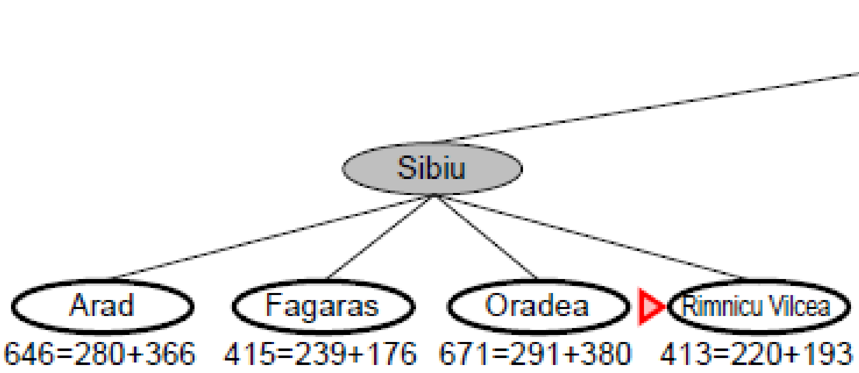
h(n)

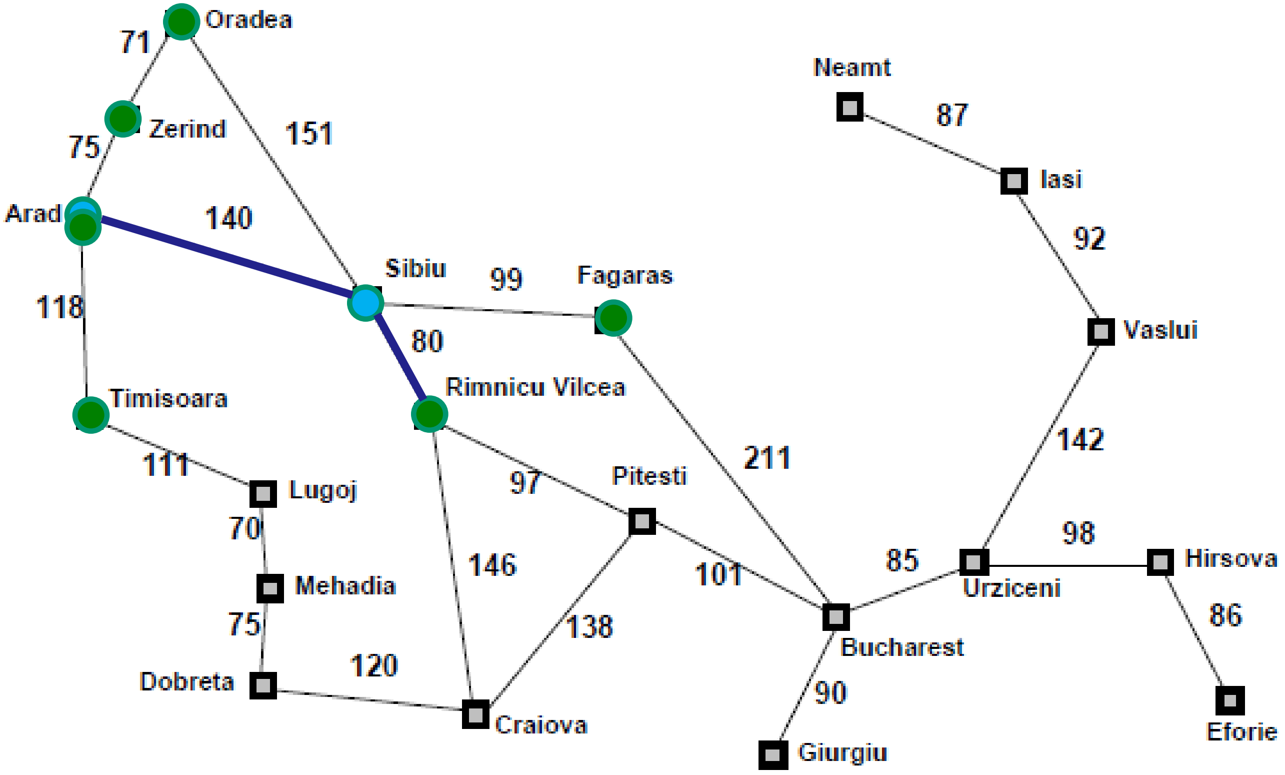


Straight−line distance
to Bucharest

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

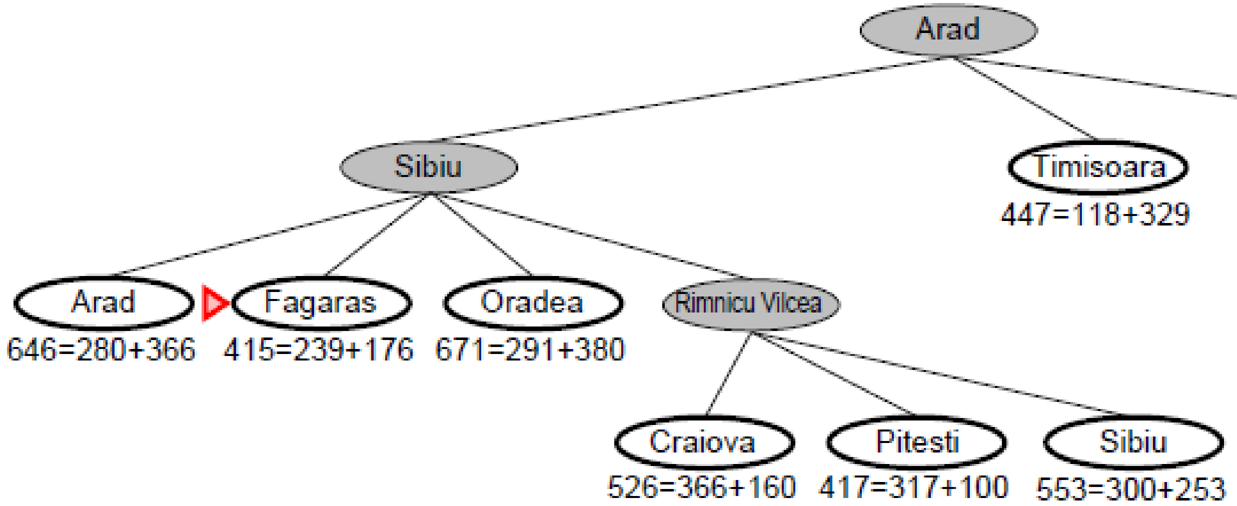
A* search example

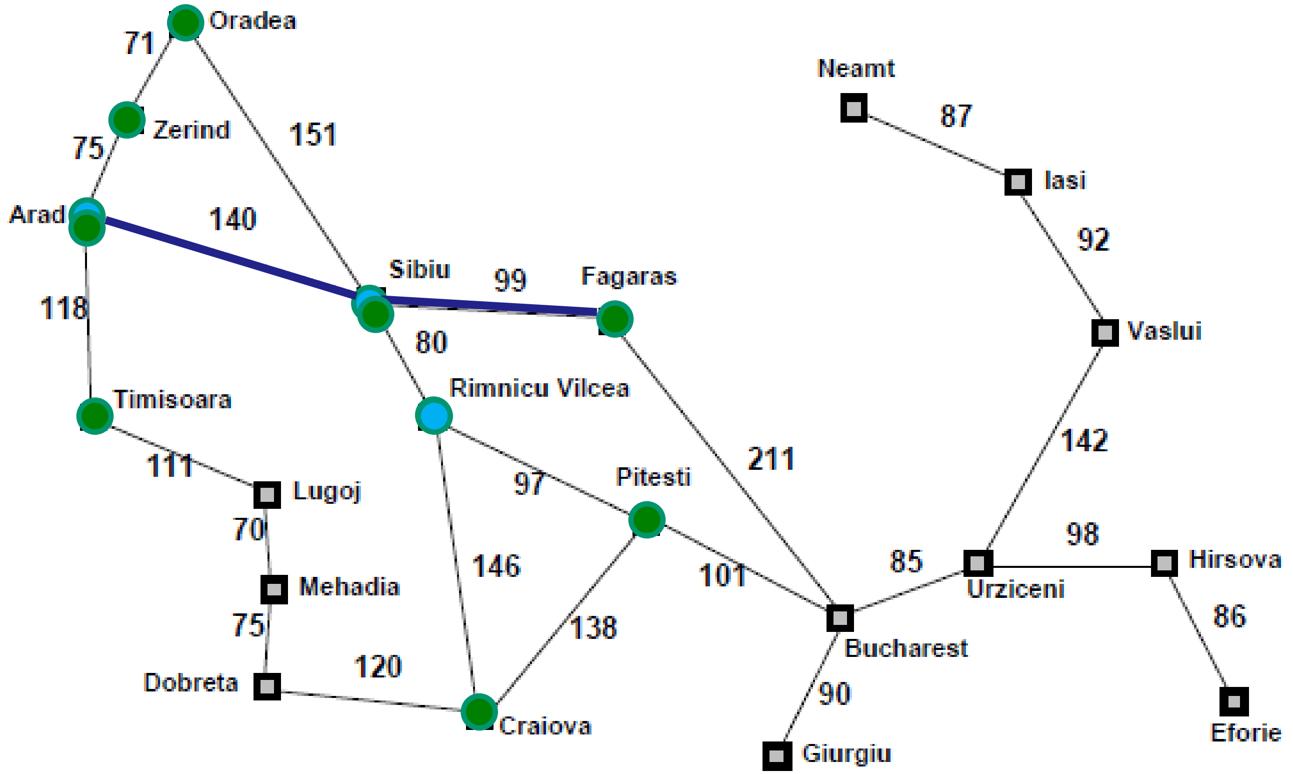




h(n)

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

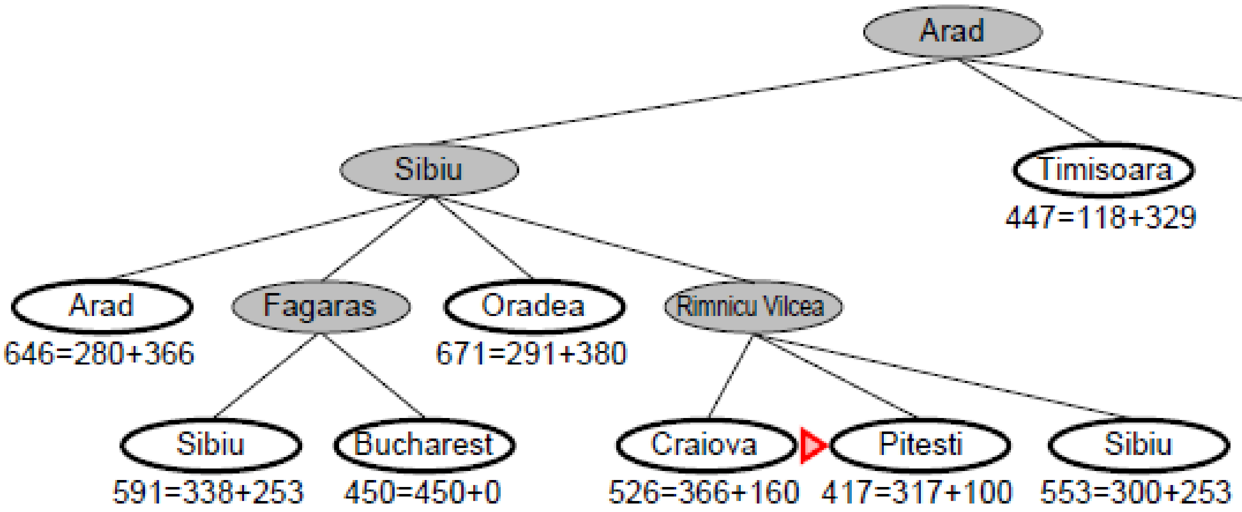


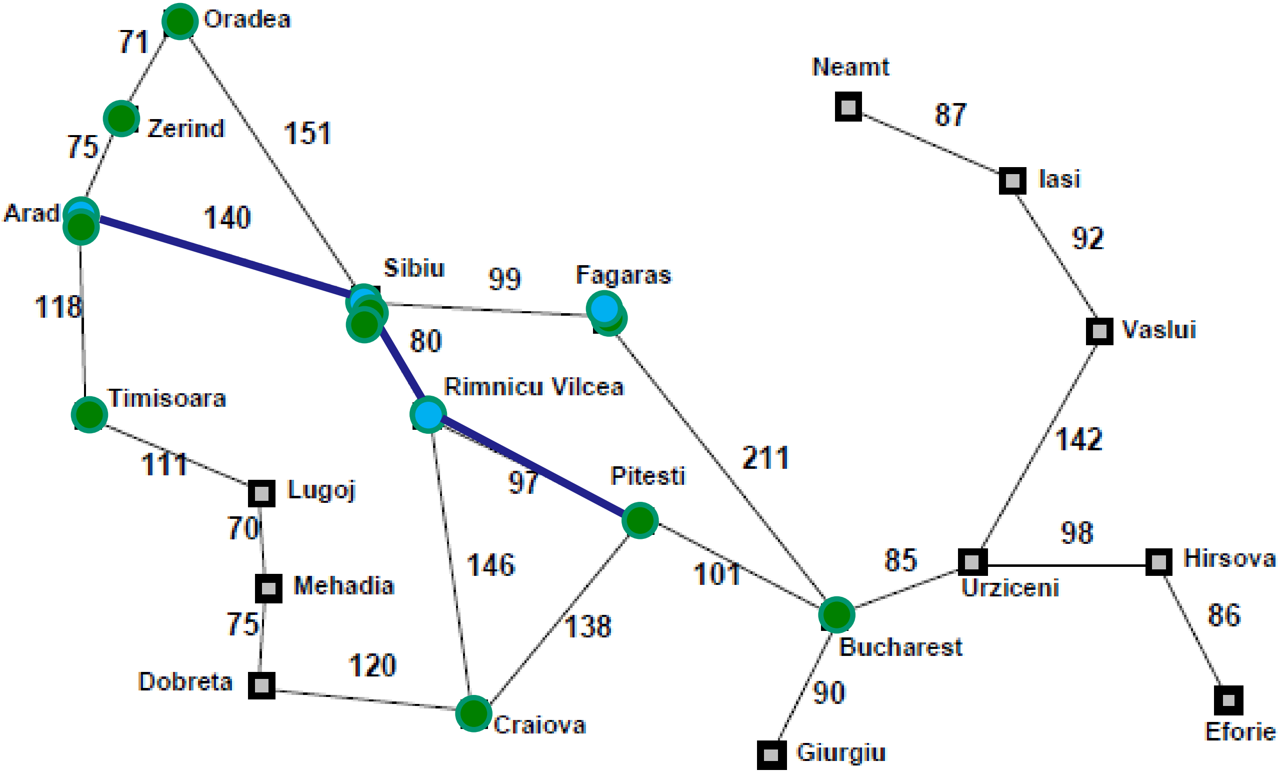


h(n)

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

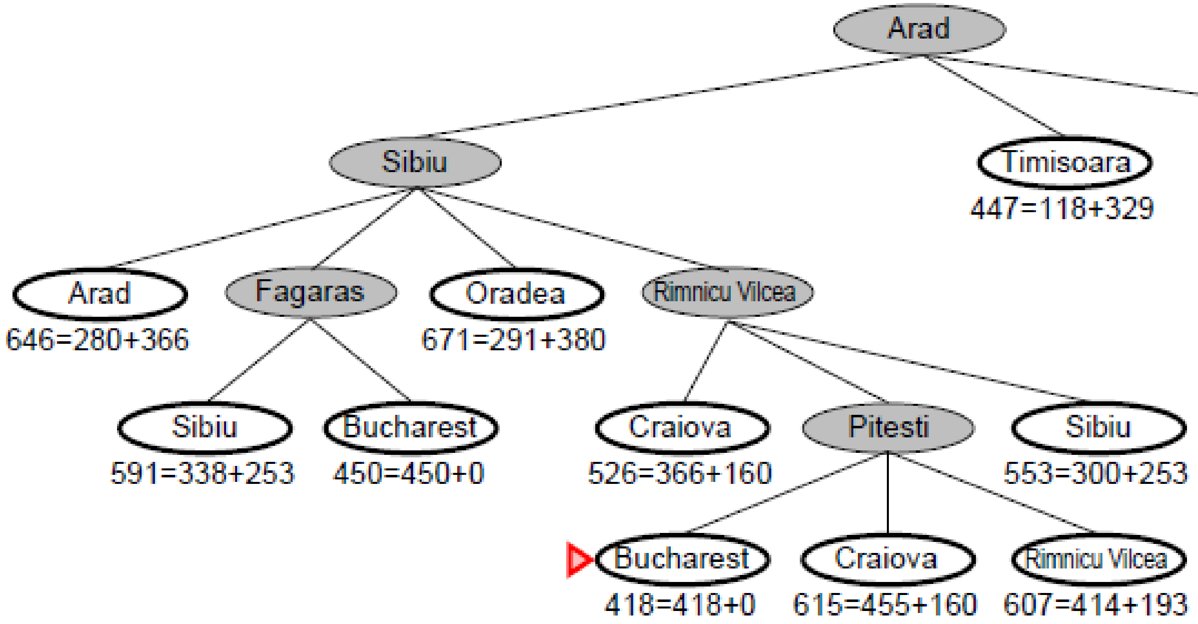


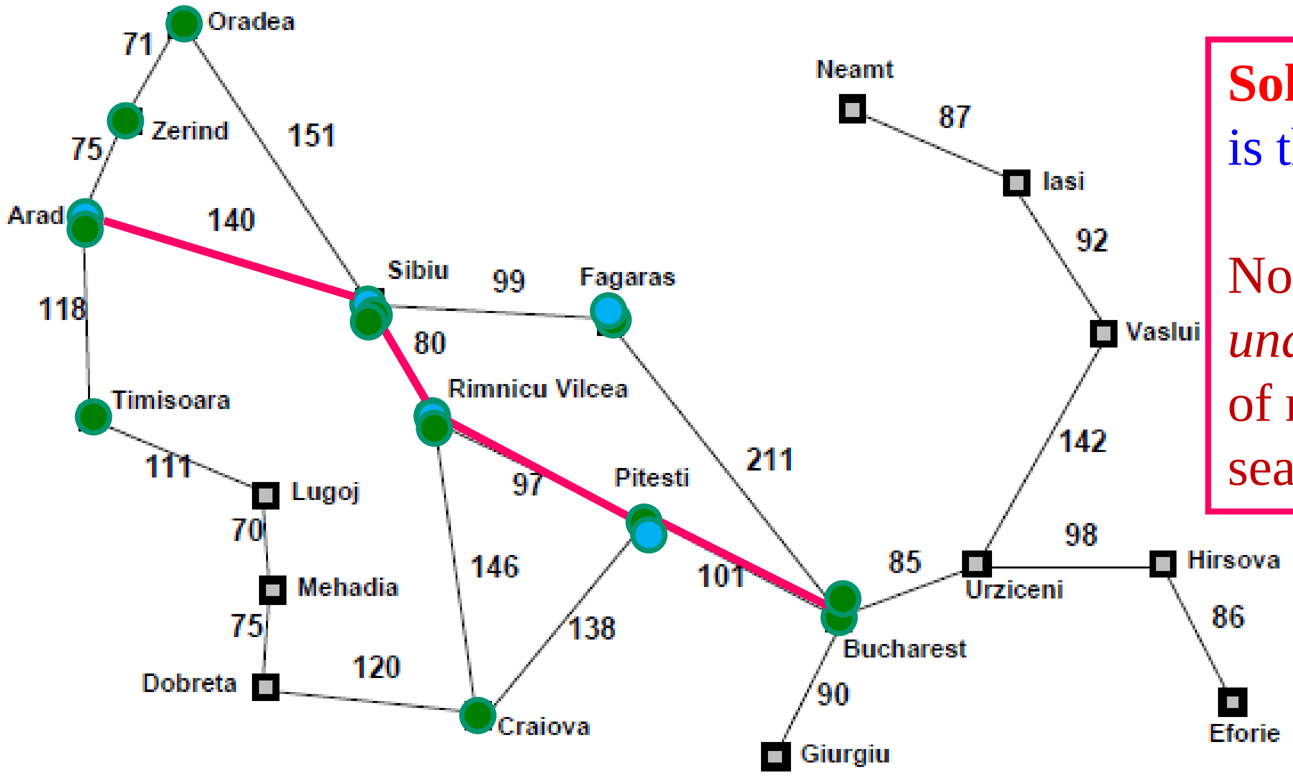


h(n)

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





Solution is optimal!

is this a coincidence?

No, when *h(n) underestimates the cost* of reaching a goal, A* (tree search) is optimal

Admissible heuristic

- Let h*(N) be the **true** cost of the optimal path from N to a goal node

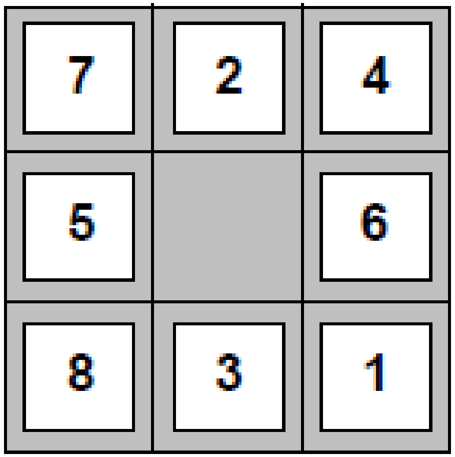
- Heuristic h(N) is admissible if:

$$0 \leq h(N) \leq h^*(N)$$

- An admissible heuristic is always optimistic

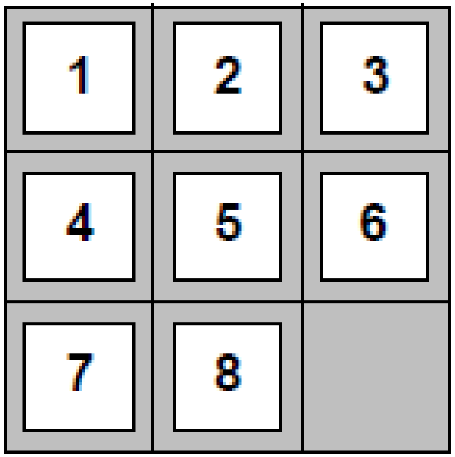
Admissible heuristics

- For the 8-puzzle: If we want to find the shortest solutions, we need a heuristic function that never overestimates the number of steps to the goal. Here are two candidates:
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (no. of squares from desired location of each tile)



7	2	4
5		6
8	3	1

Start State



1	2	3
4	5	6
7	8	

Goal State

h1(S) =??

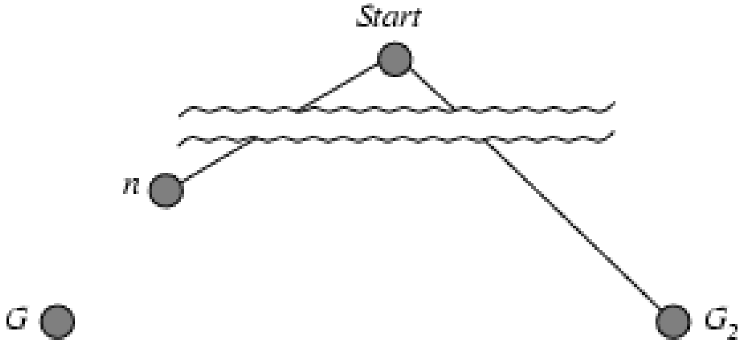
h2(S) =??

h1(S) =?? 6

h2(S) =?? 4+0+3+3+1+0+2+1 = 14

Optimality of A* (proof)

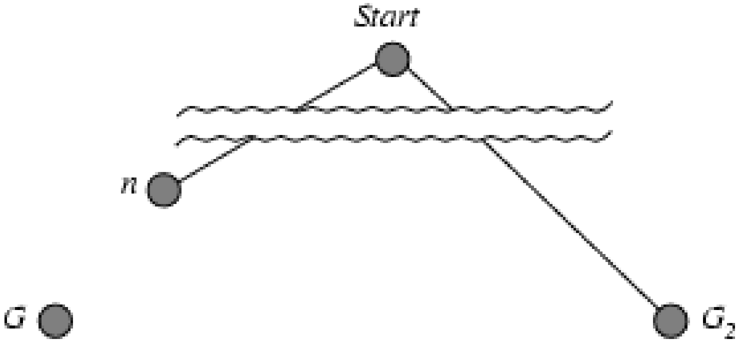
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



$f(G_2) = g(G_2)$	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

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*.



$f(G_2)$	$> f(G)$	from above
$h(n)$	$\leq h^*(n)$	since h is admissible
$g(n) + h(n)$	$\leq g(n) + h^*(n)$	
$f(n)$	$\leq f(G)$	

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

Properties of A*

- Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time?? Exponential in [relative error in h x length of soln.]
- Space?? Keeps all nodes in memory
- Optimal?? Yes-cannot expand f_{i+1} until f_i is finished

➢ A expands all nodes with $f(n) < C^*$

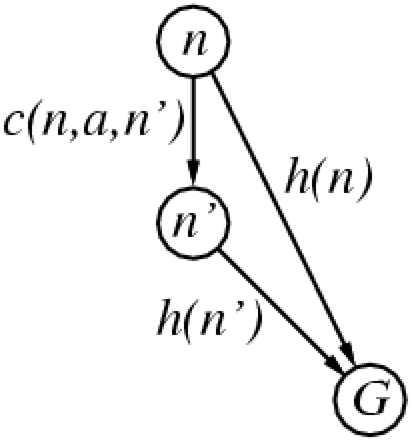
➢ A expands some nodes with $f(n) = C^*$

➢ A expands no nodes with $f(n) > C^*$

Proof of lemma: Consistency

- A heuristic is consistent if

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



- If h is consistent, we have

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

i.e., f(n) is always greater at a successor node, so once a node has been visited, it cannot be visited again at a smaller cost and thus will never be reopened

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

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)

then h_2 dominates h_1 and *is* better for search

- For 8-puzzle heuristics h1 and h2 , typical search costs (average number of nodes expanded for solution depth d):

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

d=24 IDS = 54,000,000,000 nodes
$A^*(h_1)$ = 39,135 nodes
$A^*(h_2)$ = 1,641 nodes

Major drawback of A*

- A* keeps all generated nodes in memory and usually runs out of space long before it runs out of time
- It cannot venture down a single path unless it is almost continuously having success (i.e., *h*is decreasing). Any failure to decrease *h*will almost immediately cause the search to switch to another path.

3 important factors influencing the efficiency of algorithm A*

- The cost (or length) of the path found
- The number of nodes expanded in finding the path
- The computational effort required to compute

We can overcome A* space problem without sacrificing optimality or completeness, at a small cost in execution time.

Memory Bounded Search

- Iterative deepening A* search (IDA*)
- Recursive Best-First Search(RBFS)
- Memory-bounded A* (MA*)
- Simplified MA*

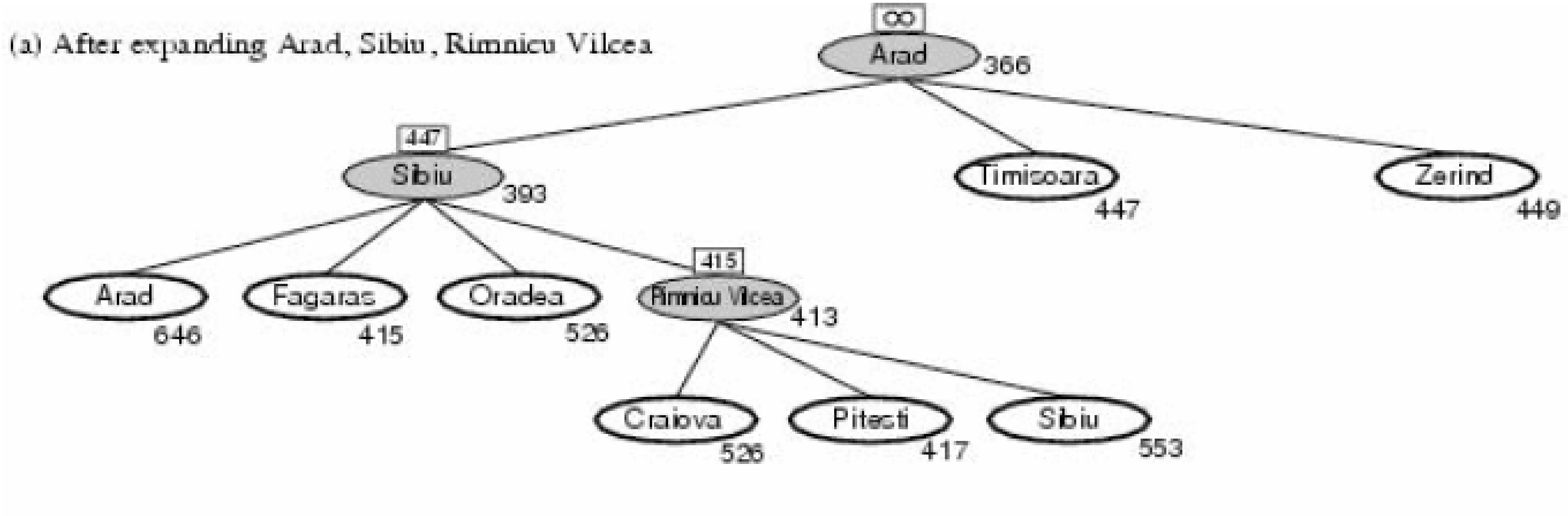
Iterative deepening A* search (IDA*)

- A useful technique for reducing memory requirements, turning A* search into iterative deepening A*, or IDA*
- In this algorithm, each iteration is a depth-first search, just as in regular iterative deepening.
- The depth-first search is modified to use an f-cost limit rather than a depth limit.
- At each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cut ff on the previous iteration.

Recursive Best-First Search(RBFS)

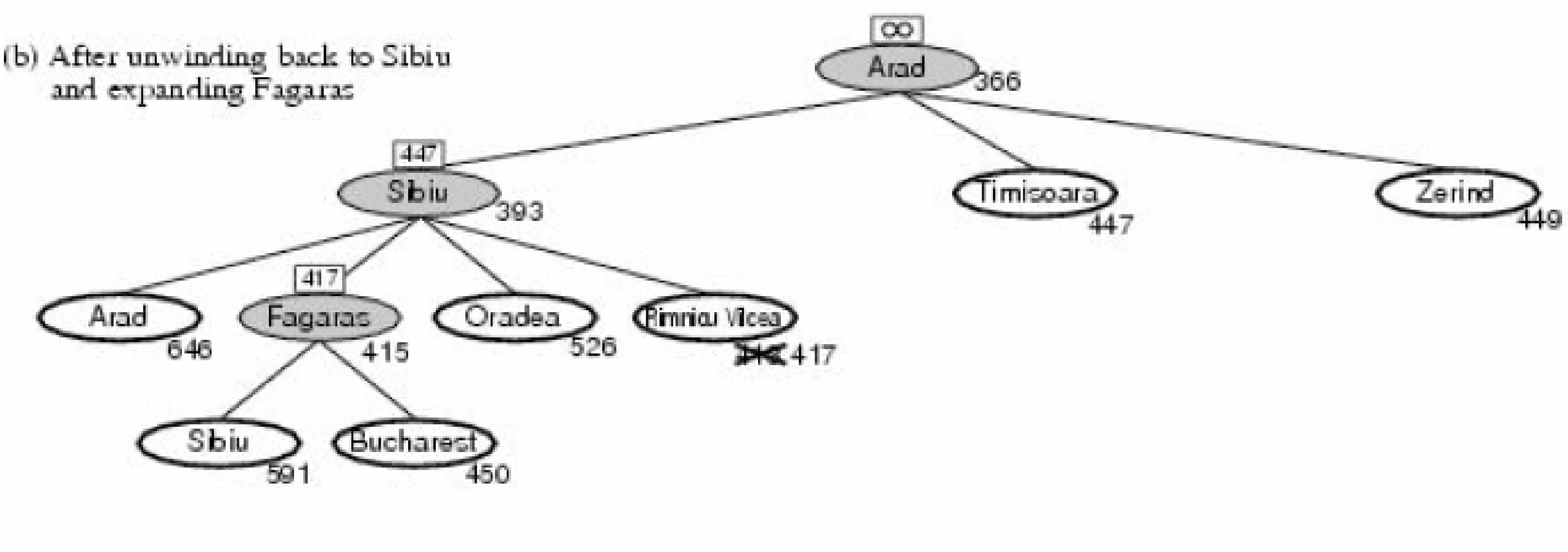
- **Keeps track of the f-value of the best-alternative path available.**
  - If current f-values exceeds this alternative f-value then backtrack to alternative path.
  - Upon backtracking change f-value to best f-value of its children.
- It takes 2 arguments:
  - a node
  - an upper bound
  - Upper bound= min (upper bound on it’s parent, current value of it’s lowest cost brother).
- It explores the sub-tree below the node as long as it contains child nodes whose costs do not exceed the upper bound.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path.
- As the recursion unwinds, RBFS replaces the *f*-value of each node along the path with the best *f*-value of its children.

RBFS: Example



- Path until Rummicu Vilcea is already expanded
- Above node; *f*-limit for every recursive call is shown on top.
- Below node: *f(n)*
- The path is followed until Pitesti which has a *f*-value worse than the *f-limit*.

RBFS: Example

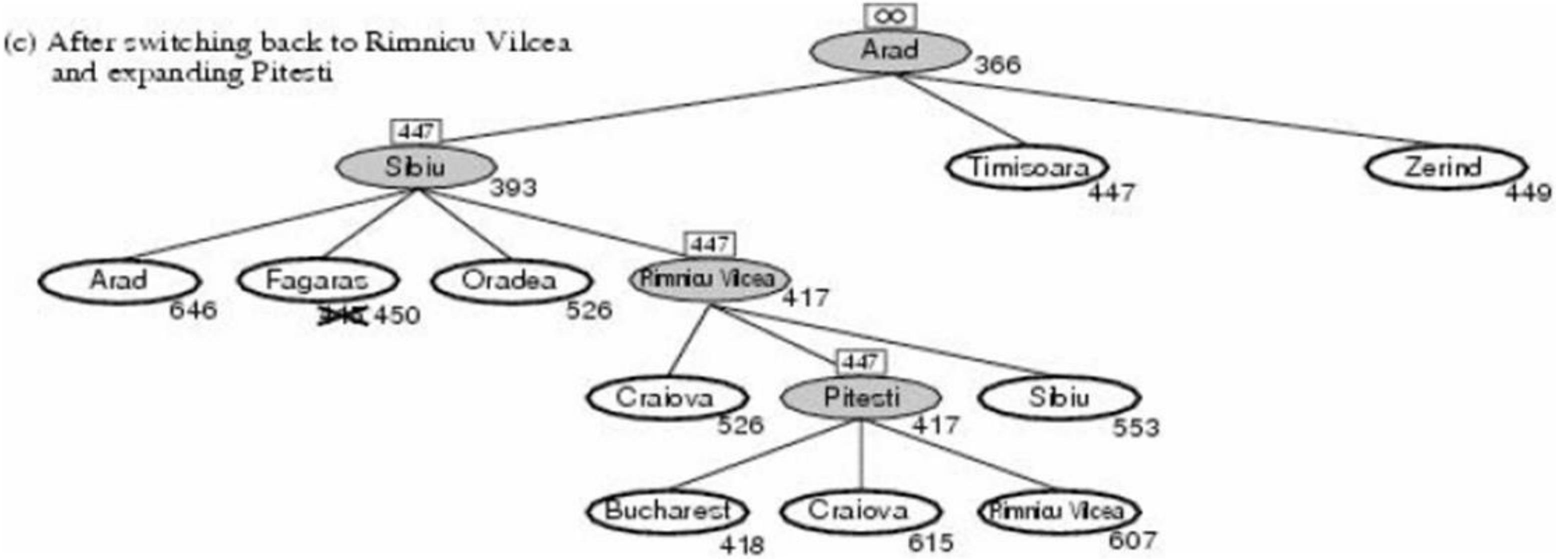


- ❑ Unwind recursion and store best *f*-value for current best leaf Pitesti

  $result, f[best] \leftarrow \text{RBFS}(problem, best, \min(f\_limit, alternative))$

- ❑ *best* is now Fagaras. Call RBFS for new *best*
  - ➲ *best* value is now 450

RBFS: Example



(c) After switching back to Rimnicu Vilcea and expanding Pitesti

- Unwind recursion and store best *f*-value for current best leaf Fagaras

  $result, f[best] \leftarrow \text{RBFS}(problem, best, \min(f\_limit, alternative))$

- *best* is now Rimnicu Viclea (again). Call RBFS for new *best*
  - Subtree is again expanded.
  - Best *alternative* subtree is now through Timisoara.
- Solution is found since because 447 > 417.

RBFS and IDA* Comparison

- RBFS is somewhat more efficient than IDA*
- But still suffers from excessive node regeneration.

- If h(n) is admissible, RBFS is optimal.
- Like A*, optimal if h(n) is admissible
- Space complexity is O(bd).
  - IDA* keeps only one single number (the current f-cost limit)
- Time complexity difficult to characterize
  - Depends on accuracy if h(n) and how often best path changes.
  - Difficult to characterize
    - Both IDA* and RBFS are subject to potential exponential increase in complexity associated with searching on Graph, because they cannot check for repeated states other than those on the current path.
    - Thus, they may explore the same state many times.

Memory-bounded A* (MA*)

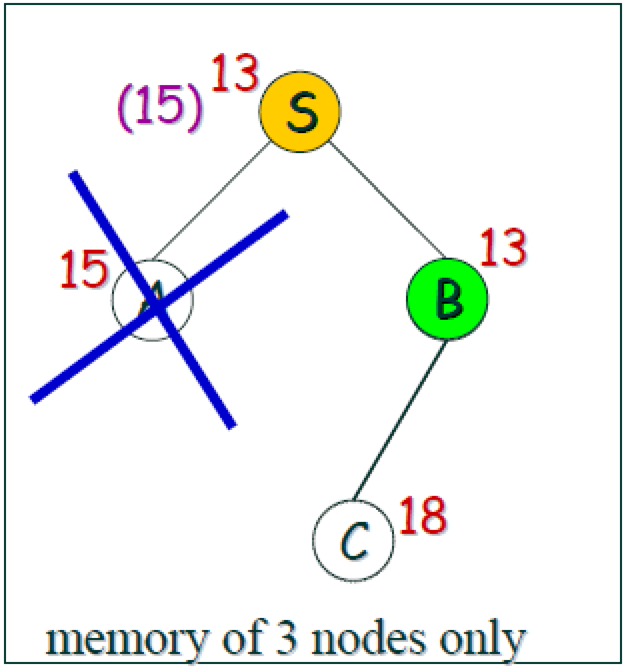
- When the bound is reached, the least promising nodes in the open list are pruned to make room for more promising nodes to be inserted.
- a simpler and more efficient version of MA*, called SMA*

Simplified MA*

- **Idea**
  - Expand the best leaf (just like A*) until memory is full
  - When memory is full, drop the worst leaf node (the one with the highest f-value) to accommodate new node.
    - If all leaf nodes have the same f-value, SMA* deletes the oldest worst leaf and expanding the newest best leaf.
  - Avoids re-computation of already explored area
    - Keeps information about the best path of a “forgotten” subtree in its ancestor.
  - Complete if there is enough memory for the shortest solution path
  - Often better than A* and IDA*
    - Trade-off between time and space requirements

SMA*

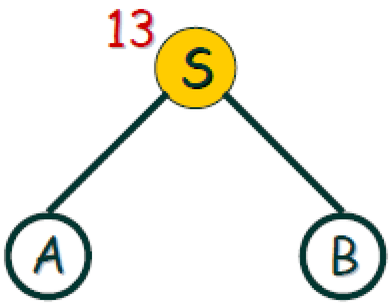
- Optimizes A* to work within reduced memory.
- Key idea:



memory of 3 nodes only

- **If memory is full and we need to generate an extra node (C):**
  - Remove the highest f-value leaf from QUEUE (A).
  - Remember the f-value of the best 'forgotten' child in each parent node (15 in S).

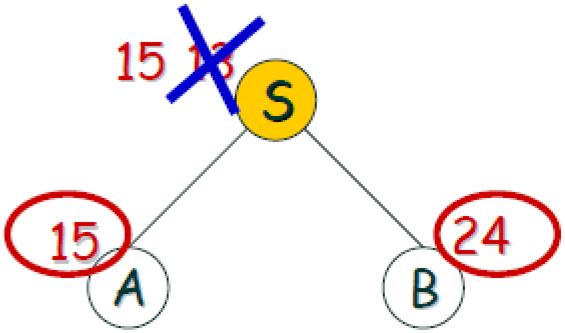
Generate Successor 1 by 1



First add A, later B

- When expanding a node (S), only add its successor 1 at a time to QUEUE.
  - we use left-to-right
- Avoids memory overflow and allows monitoring of whether we need to delete another node

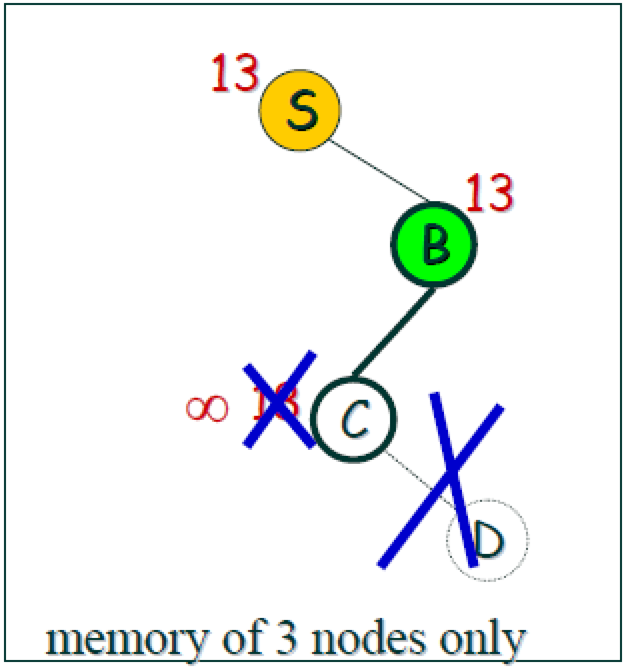
Adjust f-values



better estimate for f(S)

- ❑ **If all children M of a node N have been explored and for all M:**
  - ➲ f(S...M) > f(S...N)
- ❑ **Then reset:**
  - ➲ f(S…N) = min { f(S…M) | M child of N}
  - ➲ A path through N needs to go through 1 of its children !

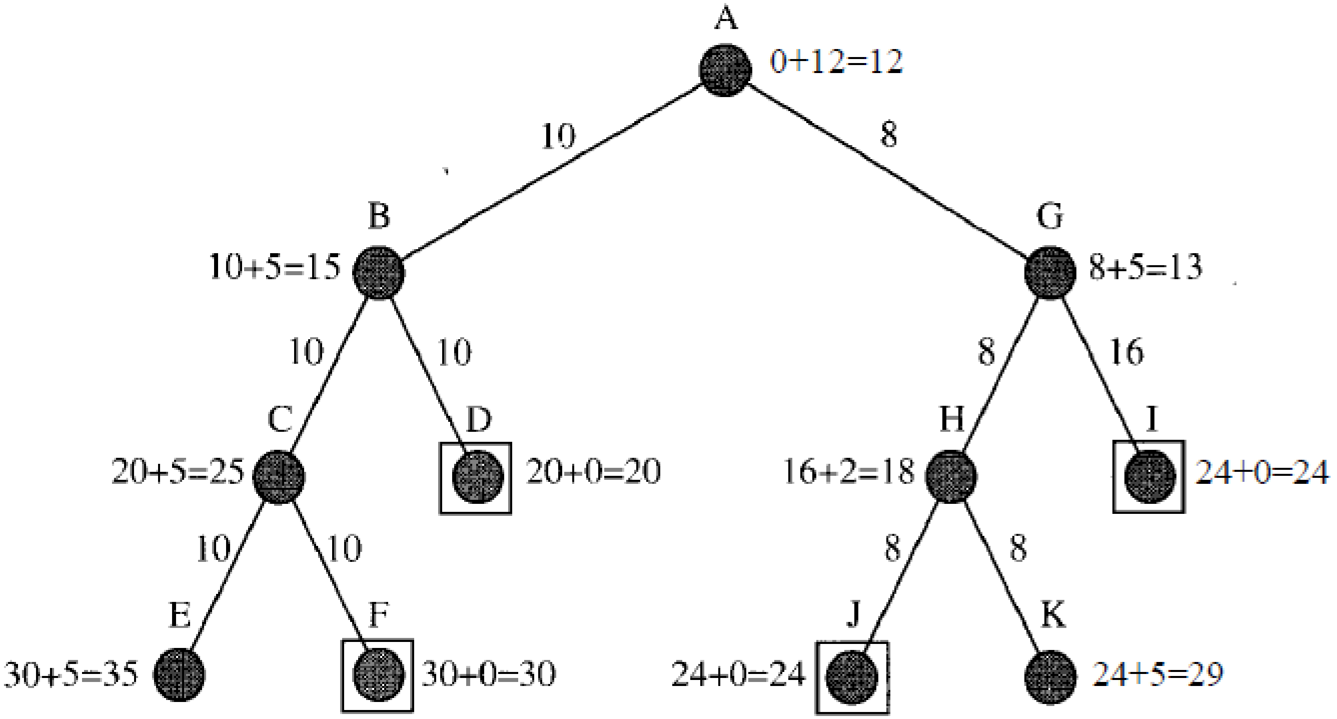
Too long path: give up



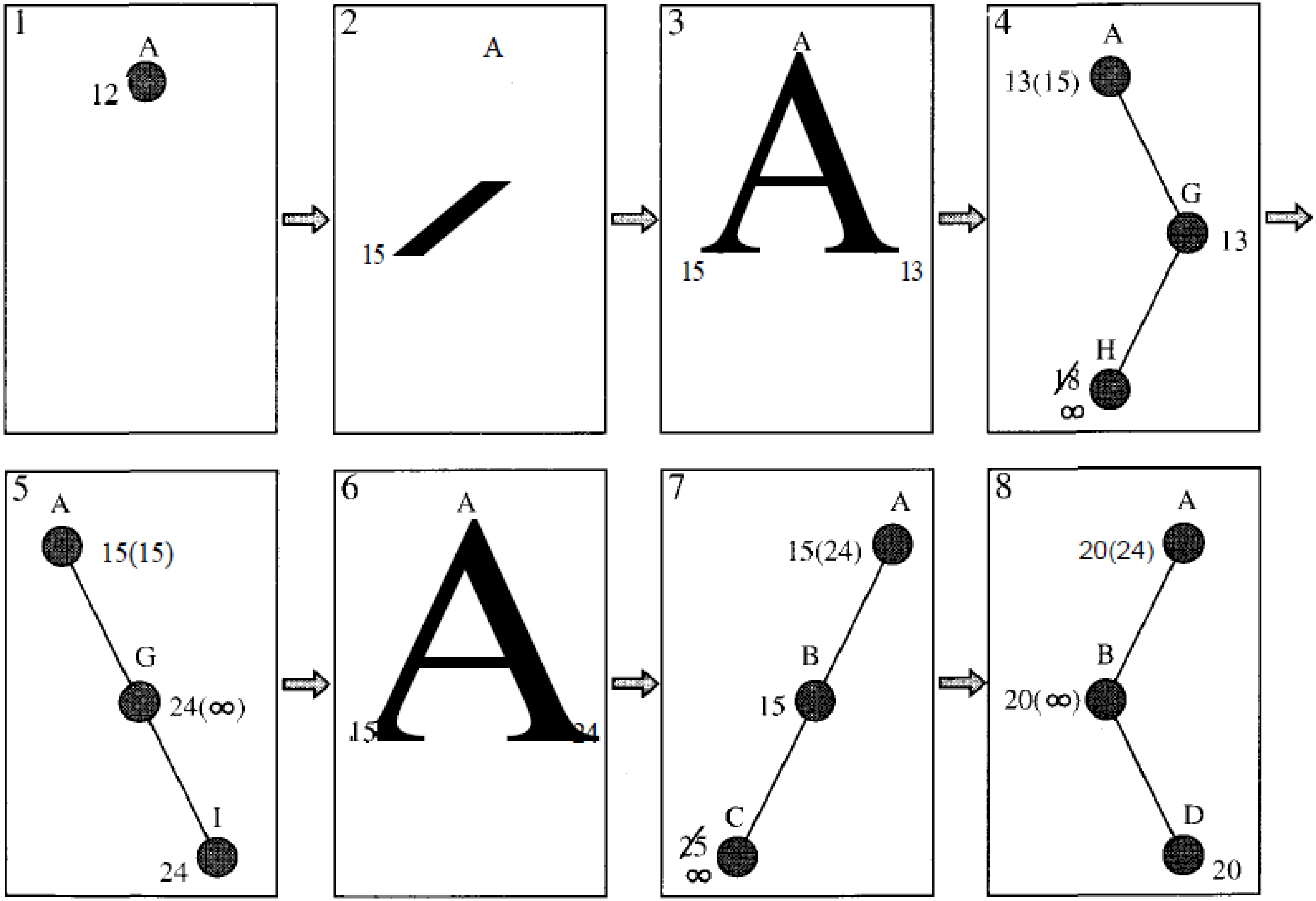
memory of 3 nodes only

- If extending a node would produce a path longer than memory: give up on this path (C).
- Set the f-value of the node (C) to ∞
  - (to remember that we can’t find a path here)

SMA* Example



Each node is labelled with *g* + *h* — *f* values, and the goal nodes (D,F, I, J) are shown in squares. The aim is to find the lowest-cost goal node with enough memory for only *three* nodes.



1
A
12
2
A
15
3
A
15
13
4
A
13(15)
G
13
H
~~18~~
∞
5
A
15(15)
G
24(∞)
I
24
6
A
15
24
7
A
15(24)
B
15
C
~~25~~
∞
8
A
20(24)
B
20(∞)
D
20

Relaxed Problems

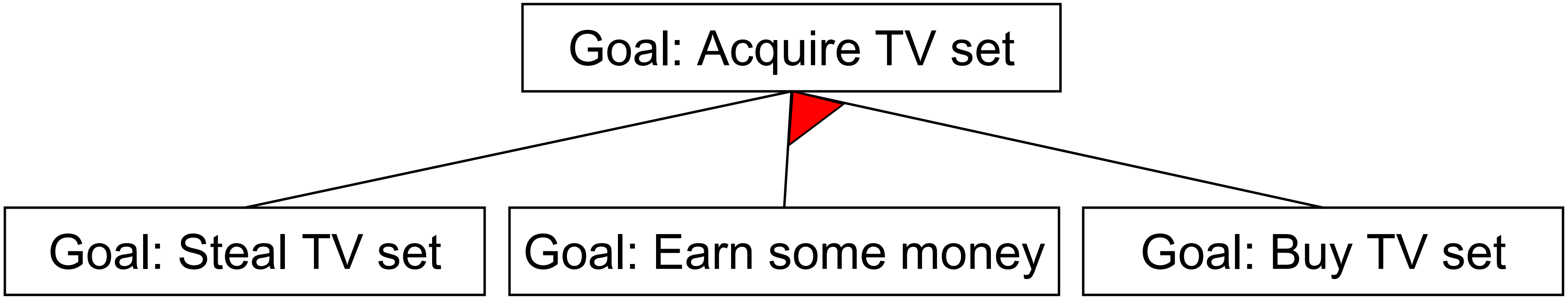
- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem
- **Example:**

➢ 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

- **Key point**: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Problem Reduction


Goal: Acquire TV set

Goal: Steal TV set

Goal: Earn some money

Goal: Buy TV set

AND-OR Graphs

Algorithm AO* (Martelli & Montanari 1973, Nilsson 1980)

Problem Reduction

1. Initialize the graph to the starting node
2. Loop until the starting node is labeled SOLVED or until its cost goes above FUTILITY

   (a) Traverse the graph, starting at the initial node & following the current best path, & accumulate the set of nodes that are on that path & have not yet been expanded/labeled as solved

   (b) Pick one of these unexpanded nodes & expand it. if there are no successors, assign FUTILITY as the value of this node. Otherwise, add its successors to the graph & for each of them compute f` (use only h` & ignore g). If f` of any node is 0, mark that node as SOLVED

(c) Change the f` estimate of the newly expanded node to reflect the new info provided by its successors

-propagate this change backward through the graph

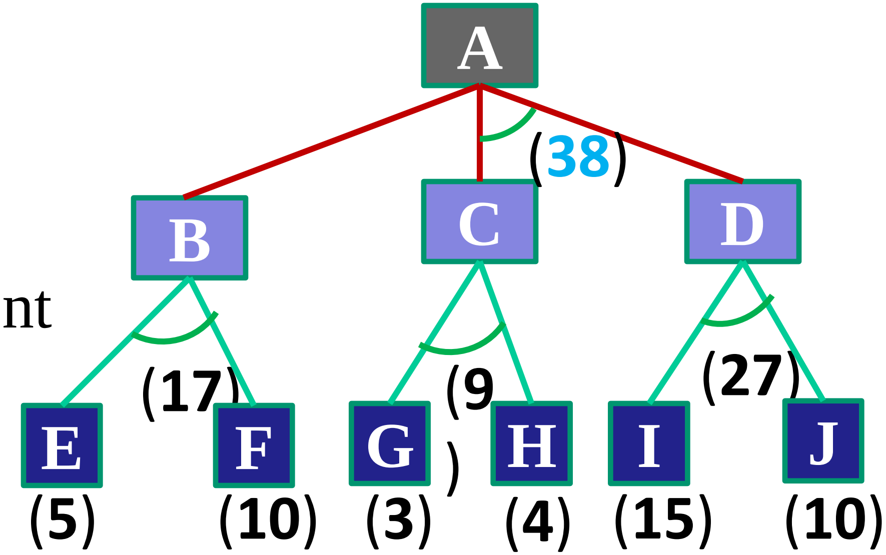
-if any node contains a successor are whose descendants are all solved, label the node itself as SOLVED

-at each node that is visited while going up the graph, decide which of its successor arcs is the most promising & mark it as part of the current best path

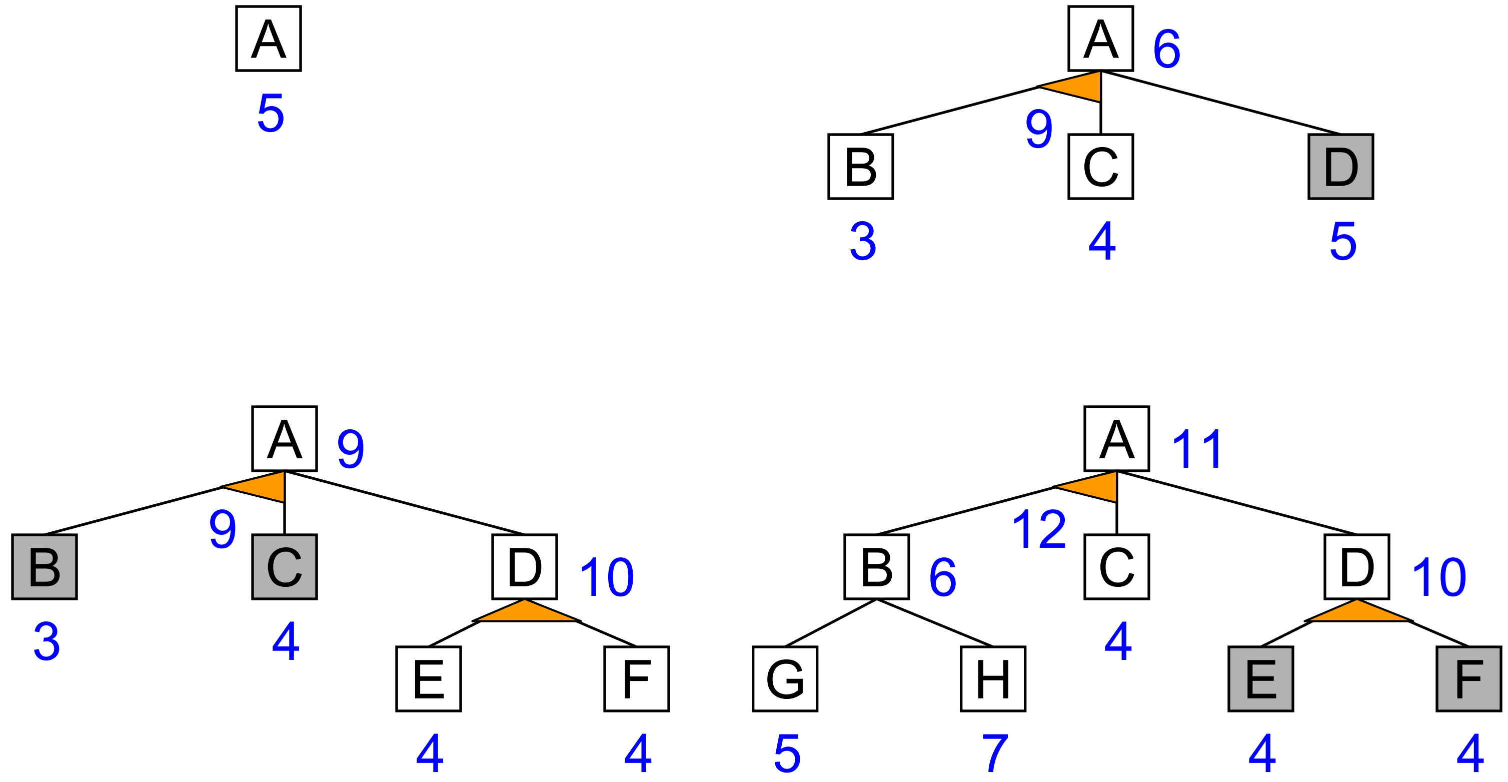
- This may cause the current best path to change
- This propagation of revised cost estimates back up the tree was not necessary in the BFS algo because only unexpanded nodes were examined
- But now expanded nodes must be reexamined so that the best current path can be selected
- Thus it is important that their f` value estimates available

AND-OR Graph

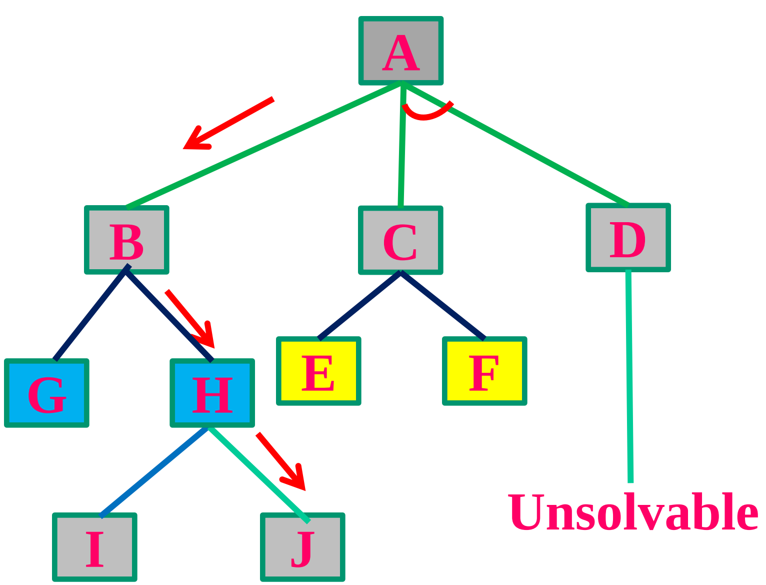
- Most promising single node: **G (f`=3)**
- Arc: **G-H (7+2=9)**
- But that arc is not part of the current best path since to use it, we must also use the arc: **I-J (27)**
- Path A-B-E-F is better (5+10+2+1=18)
- Should not expand **G** next; rather should examine either **E/F**

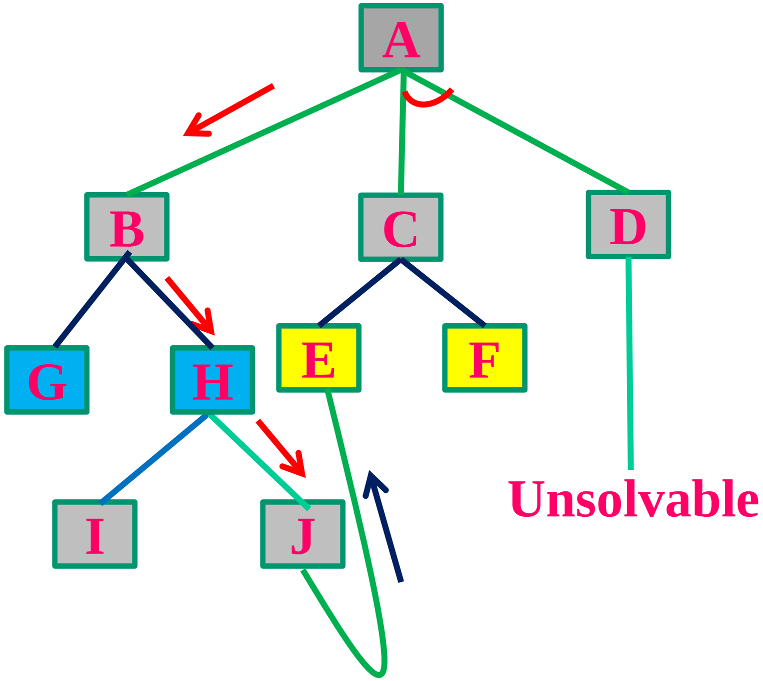


Problem Reduction(Example)



Problem Reduction: Limitation





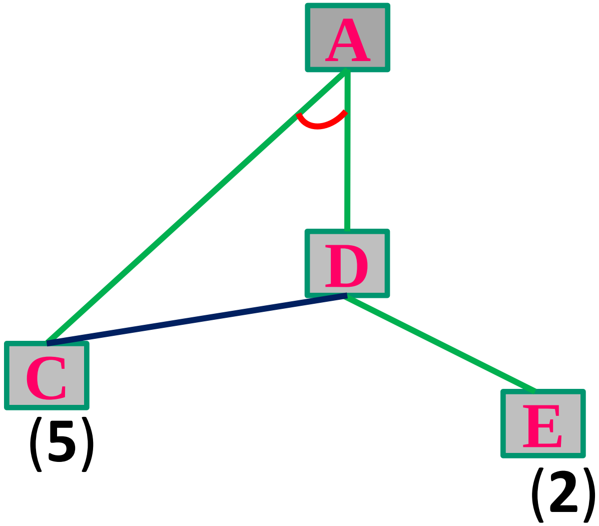
-suppose J is expanded at the next step
-that one of its successors in node E

❑**Long path may be better**

-This new path to E is longer than the previous path to E going through C
-but since the path through **C** will only lead to a solution if there is also a solution to **D**, which we Know there is not, the path through **J** is better

Problem Reduction: Limitation

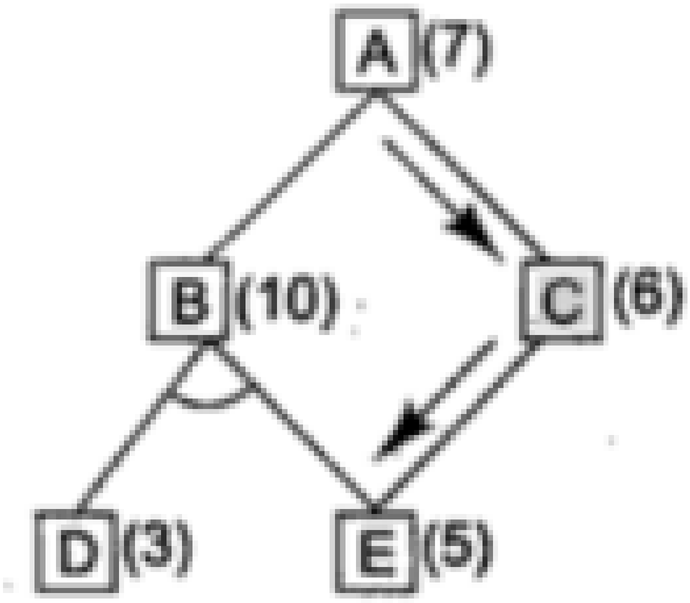
- Fails to take into account any interaction between sub goals
- Both node C/E ultimately lead to a solution, algorithm will report a complete solution that includes both of them
- AND-OR graph states that for A to be solved, both C/D must be solved
- The algo considers the solution of D as a completely separate process from the solution of C
- Looking just at the alternatives from D, E is the best path
- But it turns out that C is necessary anyway, so it would be better also to use it to satisfy D
- But the algo does not consider such interactions, it will find a non-optimal path.



AO* Algorithm

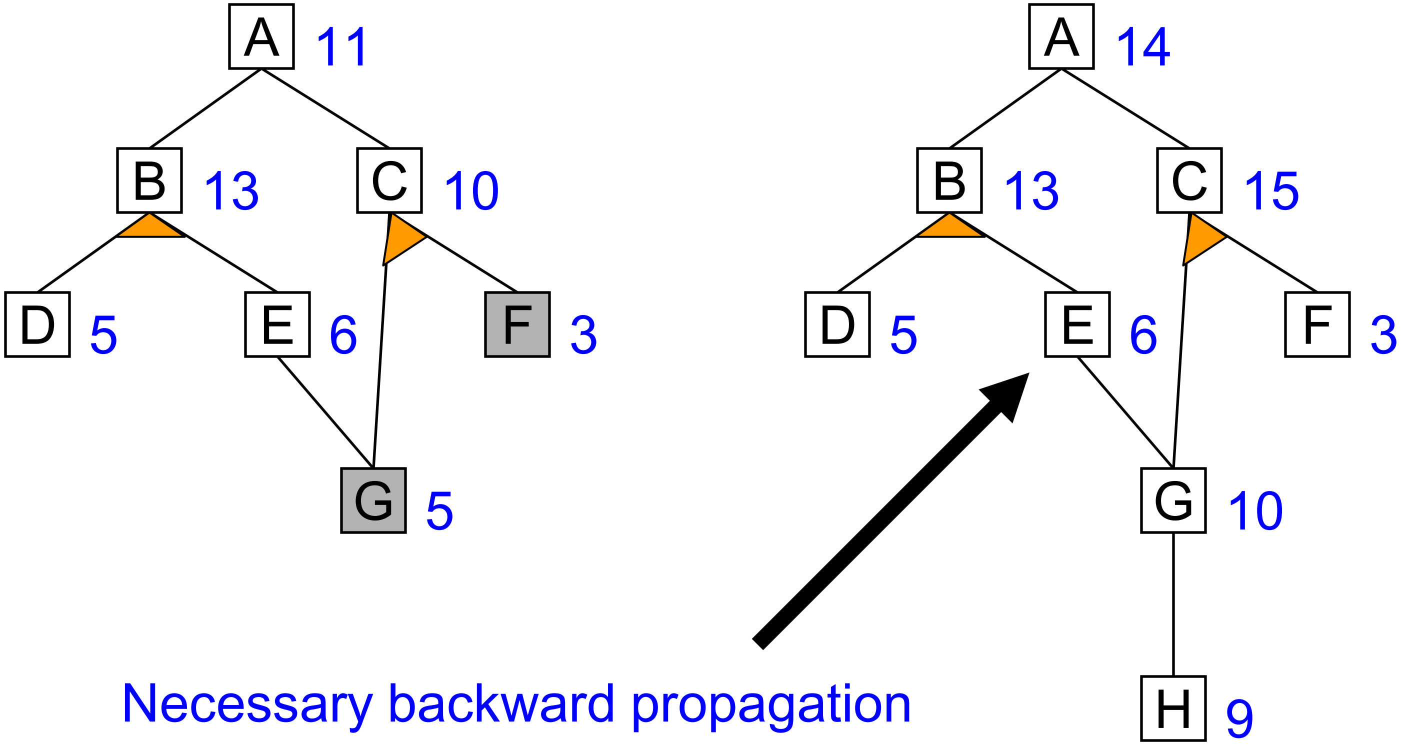
- Algorithm for searching AND-OR graph is called AO*
- Here single structure G, representing the part of the search graph that has been explicitly generated so far is used rather than two lists, OPEN and CLOSED in previous algorithms.
- Each node in the graph will point both down to its immediate successors and up to its immediate predecessor.
- Each node in a graph will also have associated with it an h value (an estimate of the cost of a path from current node to a set of solution nodes).
- We will not store g (cost from start to current node) as it is not possible to compute a single such value since there may be many paths to the same state.
- Also such value is not necessary because of the topdown traversing of the best-known path which guarantees that only nodes that are on the best path will ever be considered for expansion. So h will be good estimate for AND/OR graph search.

AO* Algorithm (Example)



Unnecessary backward propagation

AO* Algorithm (Example)

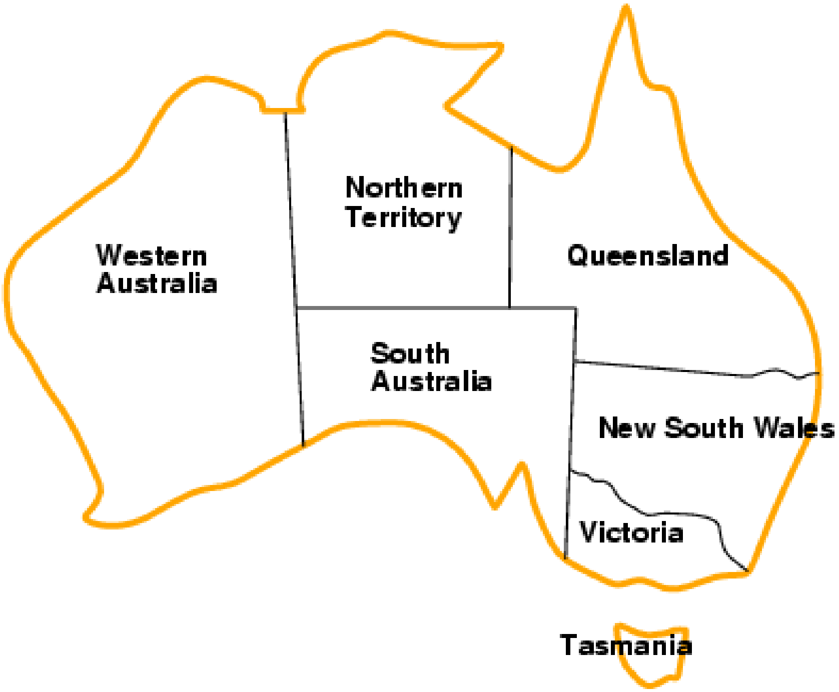


Necessary backward propagation

Constraints Satisfaction Problem (CSP)

- Standard search problem:
  - state is a "black box“ – any data structure that supports successor function, heuristic function, and goal test
- **CSP:**
  - state is defined by variables X_i with values from domain D_i
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms

Example: Map-Coloring



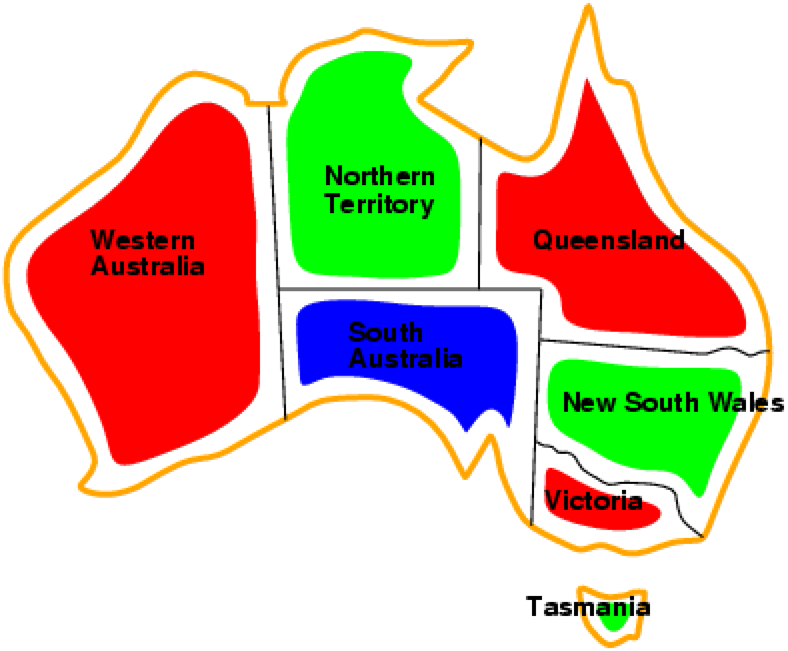
Variables *WA, NT, Q, NSW, V, SA, T*

Domains D_i = {red,green,blue}

Constraints: adjacent regions must have different colors

e.g., **WA ≠ NT,** or (WA,NT) in {(red,green),(red,blue),(green,red), (green,blue),(blue,red),(blue,green)}

Example: Map-Coloring



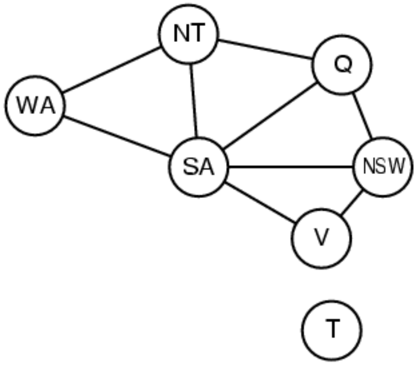
- Solutions are complete and consistent assignments satisfying all constraints,

e.g.,{WA = red, NT = green, Q = red, NSW = green, V = red,

SA = blue, T = green}

Constraint graph

- **Binary CSP**: each constraint relates at most two variables
- **Constraint graph:** nodes are variables, arcs show constraints



Varieties of CSPs

❑ Unary constraints involve a single variable,

SA ≠ green

❑ Binary constraints involve pairs of variables,

SA ≠ WA

❑ Higher-order constraints involve 3 or more variables,

cryptarithmetic column constraints

❑ Preferences (soft constraints), e.g., red is better than green

-often representable by a cost for each variable assignment→ constrained optimization problems

Constraint Satisfaction

- As compared with a straightforward search procedure, viewing a problem as one of constraint satisfaction can reduce substantially the amount of search.

Constraint Satisfaction

- Operates in a space of constraint sets.
- Initial state contains the original constraints given in the problem.
- A goal state is any state that has been constrained “enough”.

Constraint Satisfaction

Two-step process:

1. Constraints are discovered and propagated as far as possible.
2. If there is still not a solution, then search begins, adding new constraints.

Constraint Satisfaction

Two kinds of rules:

1. Rules that define valid constraint propagation.
2. Rules that suggest guesses when necessary.

Constraint Satisfaction

- Many AI problems can be viewed as problems of constraint satisfaction.

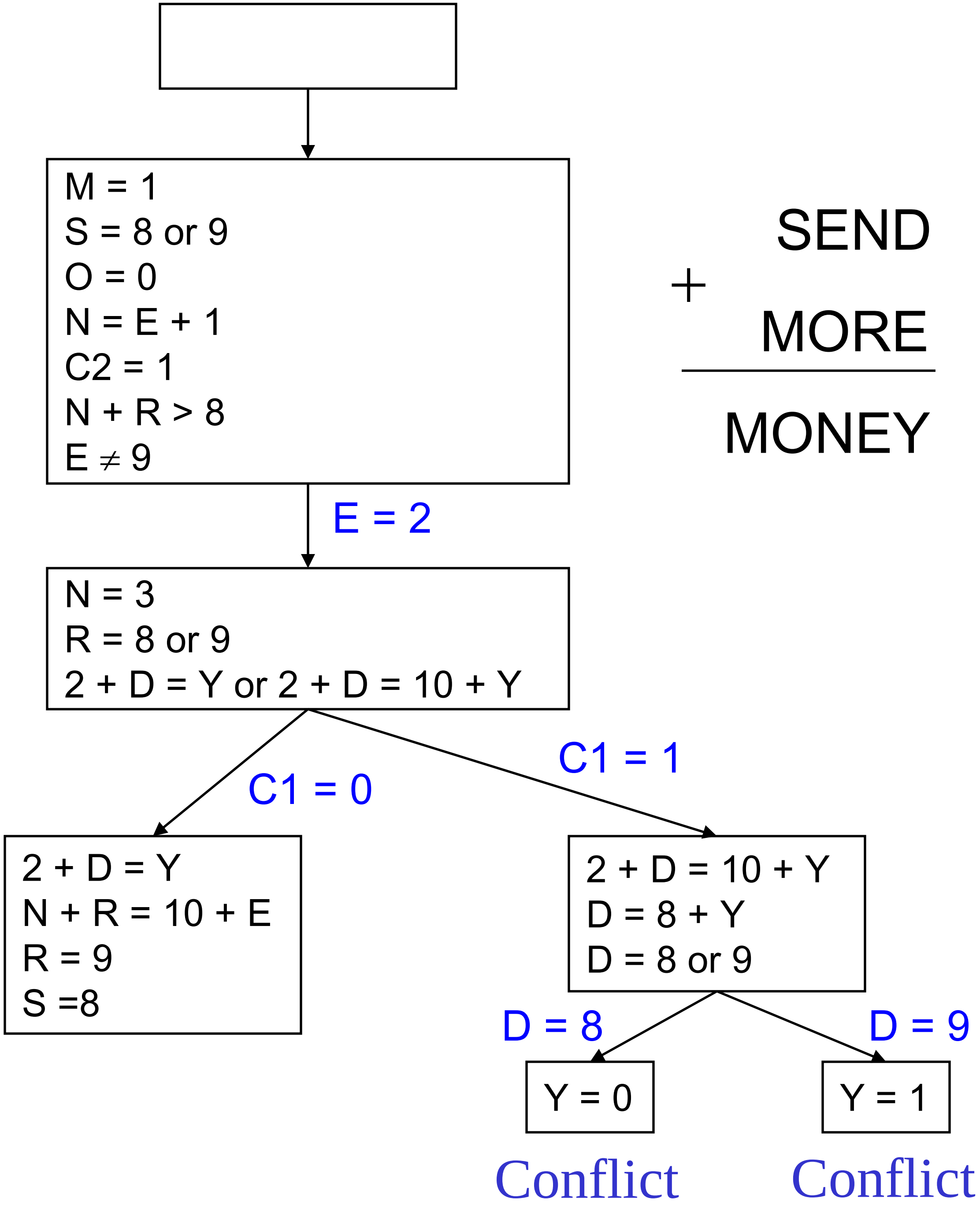
Cryptarithmetic puzzle:

$$\begin{array}{rr} & \text{SEND} \\ + & \text{MORE} \\ \hline & \text{MONEY} \end{array}$$

Initial state:

- No two letters have the same value.
- The sum of the digits must be as shown.

Initial state of problem.
D=?
E=?
Y=?
N=?
R=?
O=?
S=?
M=?
C1=?
C2=?
C1 ,C 2, C3 stands for the carry variables respectively.
Goal State: the digits to the letters must be assigned in such a manner so that the sum is satisfied.



M = 1
S = 8 or 9
O = 0
N = E + 1
C2 = 1
N + R > 8
E ≠ 9

E = 2

N = 3
R = 8 or 9
2 + D = Y or 2 + D = 10 + Y

C1 = 0

2 + D = Y
N + R = 10 + E
R = 9
S =8

C1 = 1

2 + D = 10 + Y
D = 8 + Y
D = 8 or 9

D = 8

Y = 0

Conflict

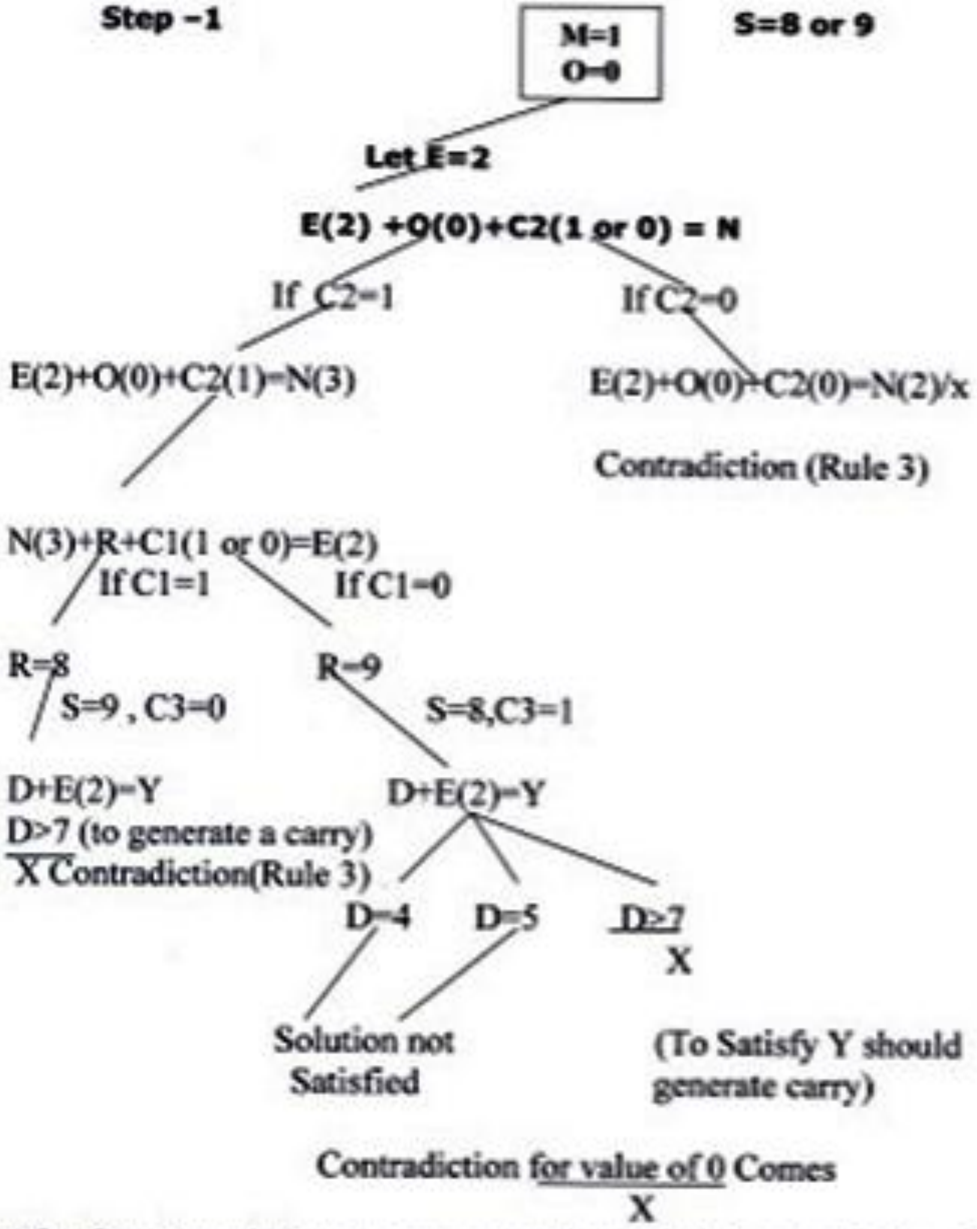
D = 9

Y = 1

Conflict

$$
\begin{array}{r}
\text{SEND} \\
+\ \text{MORE} \\
\hline
\text{MONEY}
\end{array}
$$

Step -1



M=1
O=0

S=8 or 9

Let E=2

E(2) +O(0)+C2(1 or 0) = N

If C2=1

If C2=0

E(2)+O(0)+C2(1)=N(3)

E(2)+O(0)+C2(0)=N(2)/x

Contradiction (Rule 3)

N(3)+R+C1(1 or 0)=E(2)

If C1=1

If C1=0

R=8

S=9 , C3=0

R=9

S=8,C3=1

D+E(2)=Y

D>7 (to generate a carry)

X Contradiction(Rule 3)

D+E(2)=Y

D=4

D=5

D>7

X

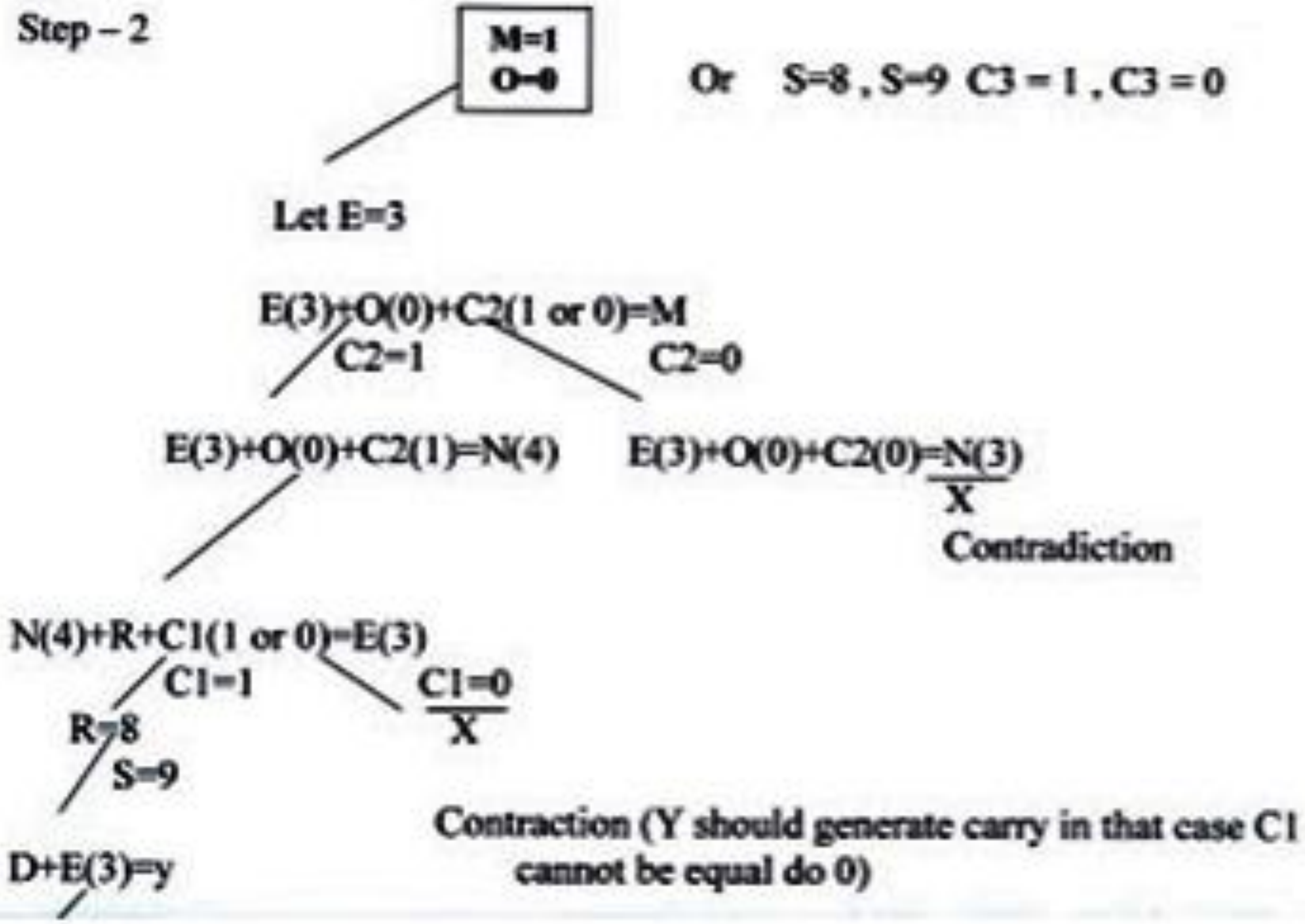
Solution not Satisfied

(To Satisfy Y should generate carry)

Contradiction for value of 0 Comes

X

After Step 1 we derive are more conclusion that Y contradiction should generate a Carry. That is D+2>9



Step – 2

M=1
O=0

Or S=8 , S=9 C3 = 1 , C3 = 0

Let E=3

E(3)+O(0)+C2(1 or 0)=M

C2=1

C2=0

E(3)+O(0)+C2(1)=N(4)

E(3)+O(0)+C2(0)=N(3)

X

Contradiction

N(4)+R+C1(1 or 0)=E(3)

C1=1

C1=0

X

R=8

S=9

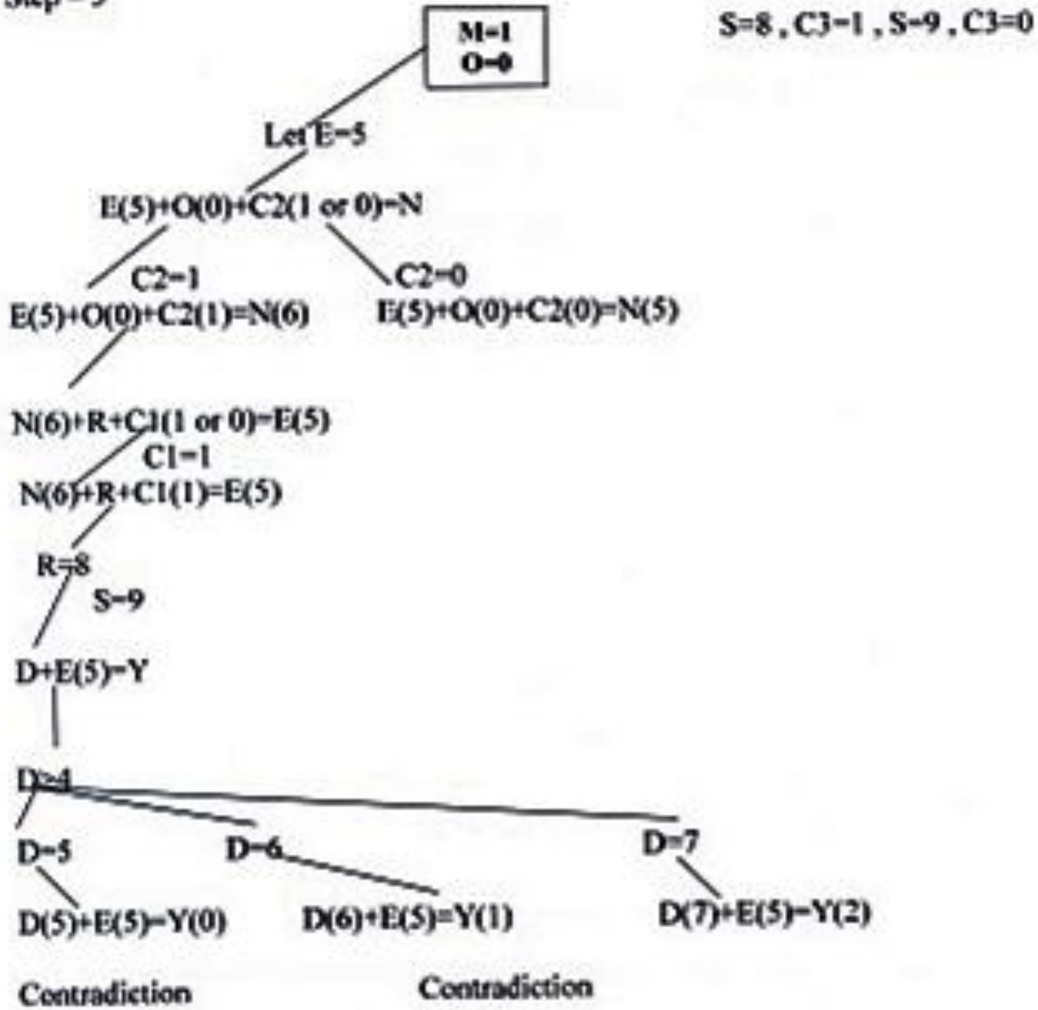
Contraction (Y should generate carry in that case C1 cannot be equal do 0)

D+E(3)=y

D>6(Controduction)

After Step 2 , we found that C1 cannot be Zero, Since Y has to generate a carry to satisfy goal state. From this step onwards, no need to branch for C1=0.

Step – 3



At Step (4) we have assigned a single digit to every letter in accordance with the constraints & production rules.

Now by backtracking , we find the different digits assigned to different letters and hence reach the solution state.

Solution!!!

- Y= 2
- D=7
- S=9
- R=8
- N=6
- E=5
- O=0
- M=1
- C1=1
- C2=0
- C3=0

C3 (0) C2 (1) C1 (1)

S (9) E (5) N (6) D (7)

+ M (1) O (0) R (8) E (5)

M (1) O (0) N (6) E (5) Y (2)

Assignment

- Why not best-first search algorithm is not adequate for searching AND-OR graphs?

More Cryptarithmetic Problem

BLACK	7 9 2 0 8
GREEN	5 3 4 4 6
------------	----------------
ORANGE	1 3 2 6 5 4

CRASH	3 6 8 4 5
HACKER	5 8 3 9 2 6
------------	-------------------
REBOOT	6 2 0 7 7 1

CROSS	9 6 2 3 3
ROADS	6 2 5 1 3
------------	----------------
DANGER	1 5 8 7 4 6