COMP338: **ARTIFICIAL INTELLIGENCE**

Solving Problems by Searching – Informed Search

Dr. Radi Jarrar

Informed Search

- Use domain knowledge!
- Are we getting close to the goal?
- Use a heuristic function that estimates how close a state is to the goal
- A heuristic does NOT have to be perfect!

Informed Search

- **Informed Search** strategies use problem-specific knowledge (beyond the definition of the problem itself)
- They can find solutions more efficiently than an uninformed strategy
- Best-First Search is a general approach that is considered in Informed Search

Informed Search

- It is an instance of the general tree-search or graph-search (Please check the book!) algorithms in which a node is selected for expansion based on an **evalution function f(n)**
- The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first

Informed Search

- Best-first graph search implementation is identical to that for uniform-cost search, except for the use of *f* instead of *g* to order the priority queue
- Most best-first algorithms include as a component of *f* a **heuristic function**, denoted h(n):

h(n) = estimated cost of the cheapest path from the state at node n to a goal state

Informed Search

- h(n) depends on the state at node n
- In Romania's example, the cost of the cheapest path can be represented as a straight line to Bucharest
- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm
- For now, consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then `h(n)=0`

Greedy Best-First Search

- Expands the node that is closest to the goal (with the aim to lead to the solution more quickly)
- It evaluates nodes by using the heuristic function: $f(n)=h(n)$ only (meaning it disregards the actual edge weights in a weighted graph)

Greedy Best-First Search

- Consider the route-finding problems in Romania. We use the **straight line distance** heuristic, which we will call h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest (from other cities)

Greedy Best-First Search

- Evaluation function h(n) (i.e., heuristic)
- h(n) estimates the cost from n to the closest goal
- Example: hSLD(n) = straight-line distance from n to *Sault Ste Marie*
- Greedy search expands the node that appears to be closest to goal

Greedy Best-First Search - Algorithm

- Initialise a tree with the root is source node
- If open list = empty, then return fail. Else, add the current node to the closed list
- Expand the node with the lowest h(x) from the open list for exploration
- If the child node is the target, then return success. Else, examine the new node (either has not been open or closed list), and then add it to the open list for exploration

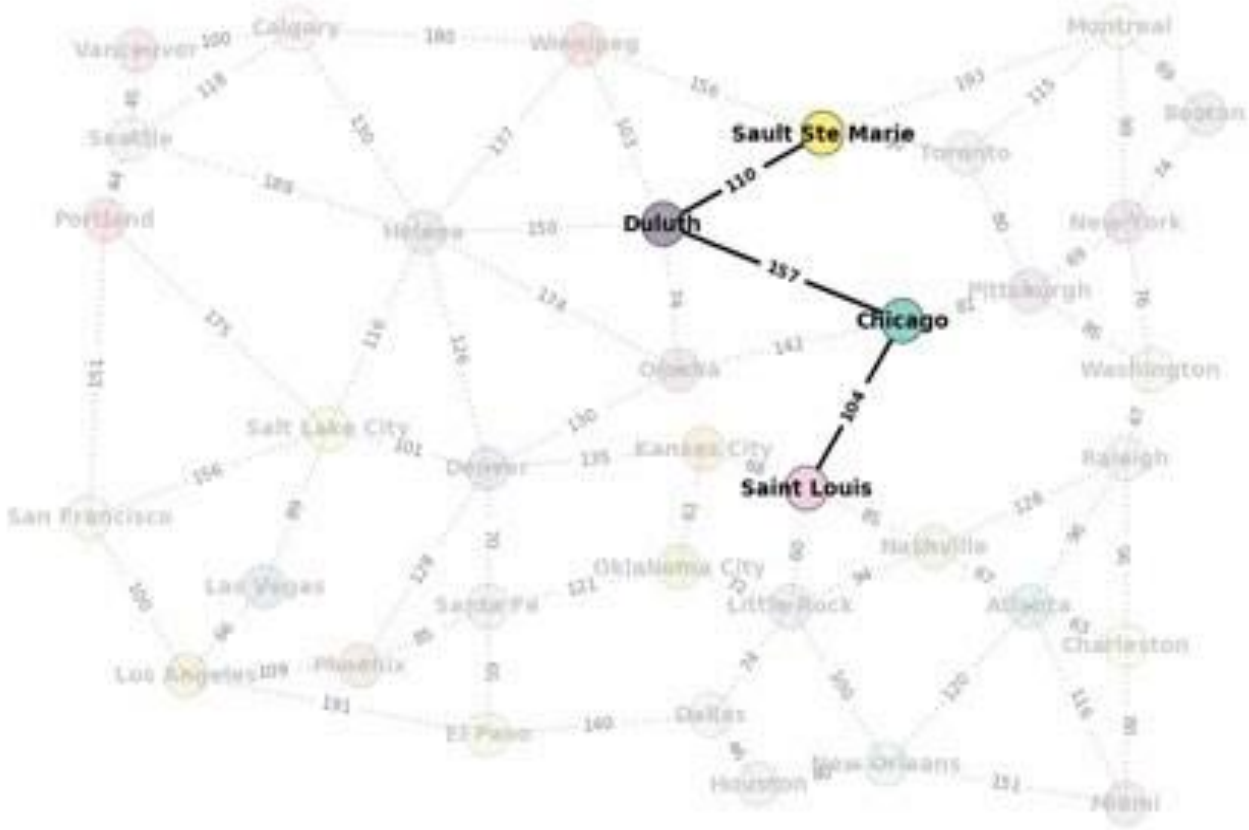
Greedy Best-First Search- Algorithm

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Examples using the map

Start: Saint Louis

Goal: Sault Ste Marie



Greedy search

Greedy best-first search - Example

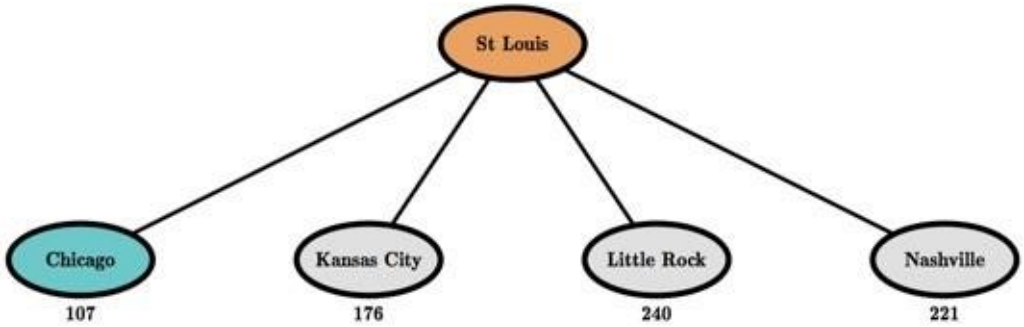
The initial state:

Atlanta	272
Boston	240
Calgary	334
Charleston	322
Chicago	107
Dallas	303
Denver	270
Duluth	110
El Paso	370
Helena	254
Houston	332
Kansas City	176
Las Vegas	418
Little Rock	240
Los Angeles	484
Miami	389
Montreal	193
Nashville	221
New Orleans	322
New York	195
Oklahoma City	237
Omaha	150
Phoenix	396
Pittsburgh	152
Portland	452
Raleigh	251
Saint Louis	180
Salt Lake City	344
San Francisco	499
Santa Fe	318
Sault Ste Marie	0
Seattle	434
Toronto	90
Vancouver	432
Washington	238
Winnipeg	156

Greedy best-first search - Example

After expanding St Louis:

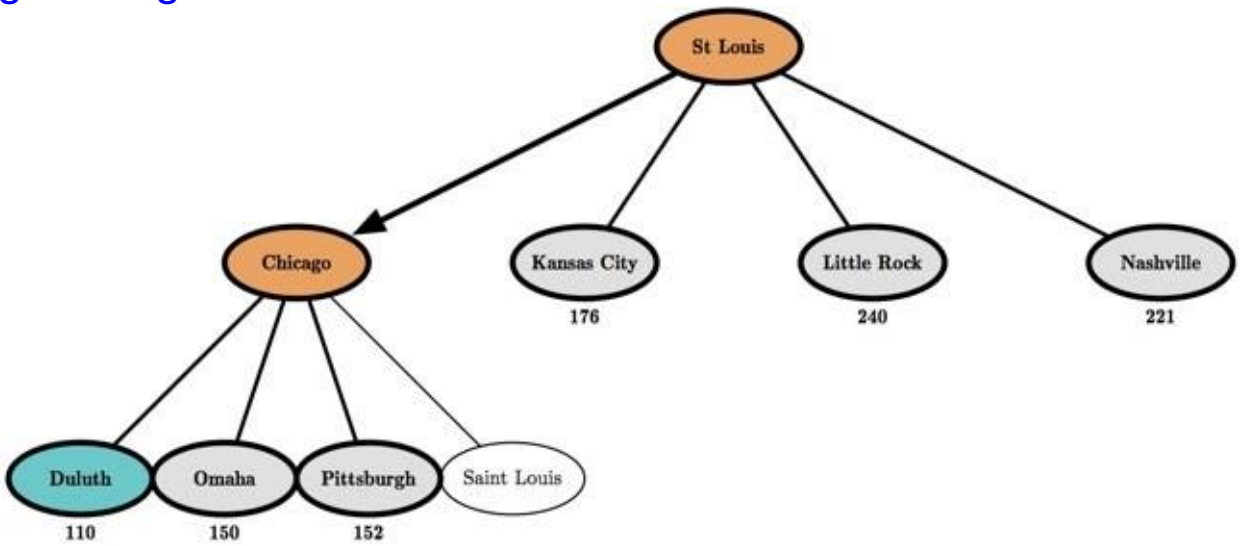
Atlanta	272
Boston	240
Calgary	334
Charleston	322
Chicago	107
Dallas	303
Denver	270
Duluth	110
El Paso	370
Helena	254
Houston	332
Kansas City	176
Las Vegas	418
Little Rock	240
Los Angeles	484
Miami	389
Montreal	193
Nashville	221
New Orleans	322
New York	195
Oklahoma City	237
Omaha	150
Phoenix	396
Pittsburgh	152
Portland	452
Raleigh	251
Saint Louis	180
Salt Lake City	344
San Francisco	499
Santa Fe	318
Sault Ste Marie	0
Seattle	434
Toronto	90
Vancouver	432
Washington	238
Winnipeg	156



Greedy best-first search - Example

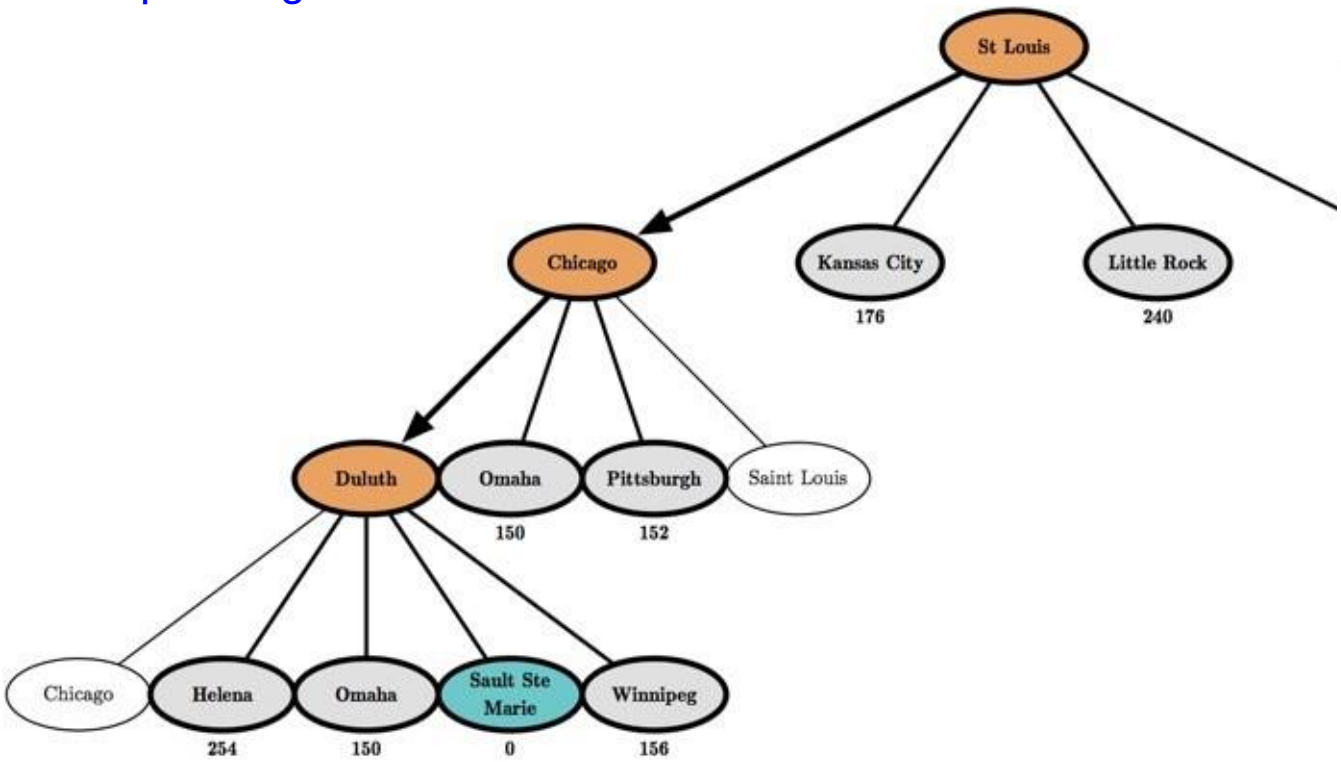
After expanding Chicago:

Atlanta	272
Boston	240
Calgary	334
Charleston	322
Chicago	107
Dallas	303
Denver	270
Duluth	110
El Paso	370
Helena	254
Houston	332
Kansas City	176
Las Vegas	418
Little Rock	240
Los Angeles	484
Miami	389
Montreal	193
Nashville	221
New Orleans	322
New York	195
Oklahoma City	237
Omaha	150
Phoenix	396
Pittsburgh	152
Portland	452
Raleigh	251
Saint Louis	180
Salt Lake City	344
San Francisco	499
Santa Fe	318
Sault Ste Marie	0
Seattle	434
Toronto	90
Vancouver	432
Washington	238
Winnipeg	156



Greedy best-first search - Example

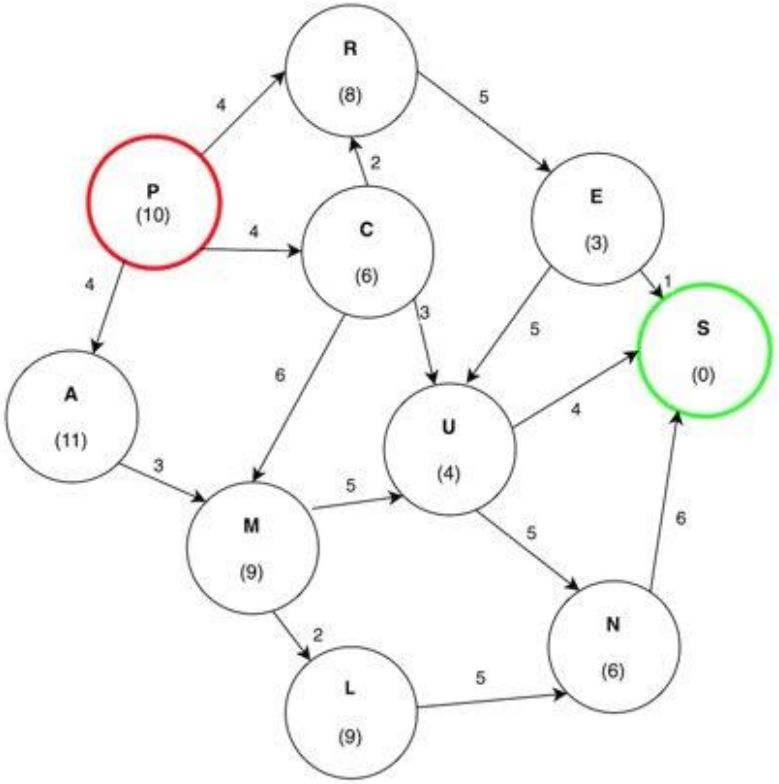
After expanding Duluth:



Atlanta	272
Boston	240
Calgary	334
Charleston	322
Chicago	107
Dallas	303
Denver	270
Duluth	110
El Paso	370
Helena	254
Houston	332
Kansas City	176
Las Vegas	418
Little Rock	240
Los Angeles	484
Miami	389
Montreal	193
Nashville	221
New Orleans	322
New York	195
Oklahoma City	237
Omaha	150
Phoenix	396
Pittsburgh	152
Portland	452
Raleigh	251
Saint Louis	180
Salt Lake City	344
San Francisco	499
Santa Fe	318
Sault Ste Marie	0
Seattle	434
Toronto	90
Vancouver	432
Washington	238
Winnipeg	156

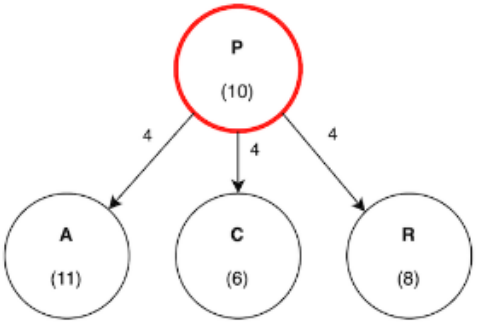
Greedy Best-First Search - Example

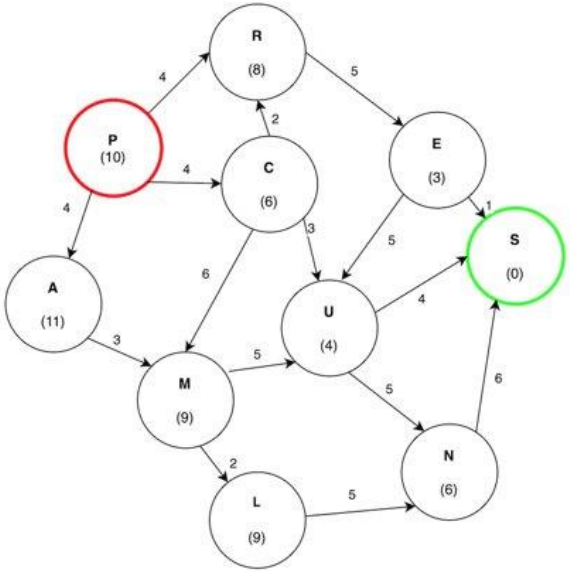
- Consider the following graph, we want to get from P to S
- We only care about the heuristic value not the actual path to the next node



Greedy Best-First Search - Example

- Expand node P

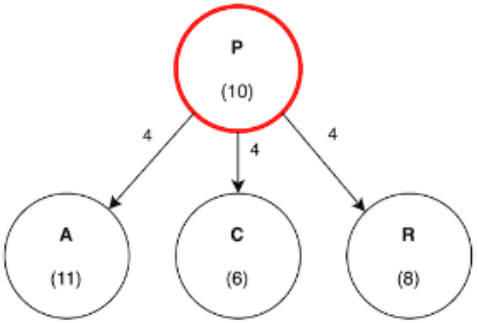


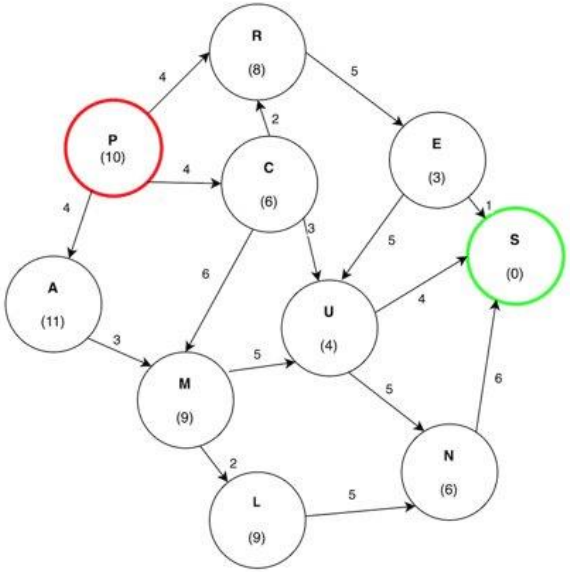


- Note that the least cost is at C

Greedy Best-First Search - Example

- Expand node P



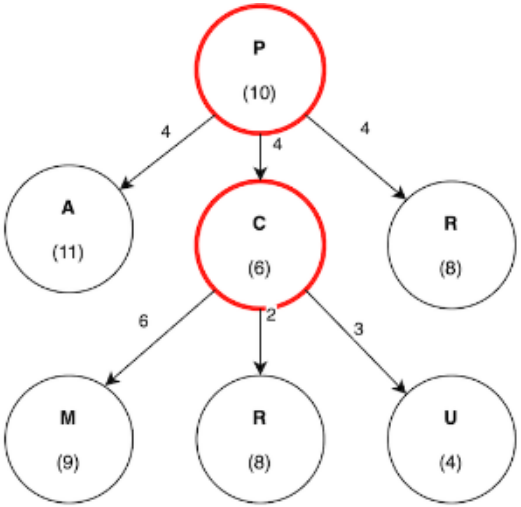


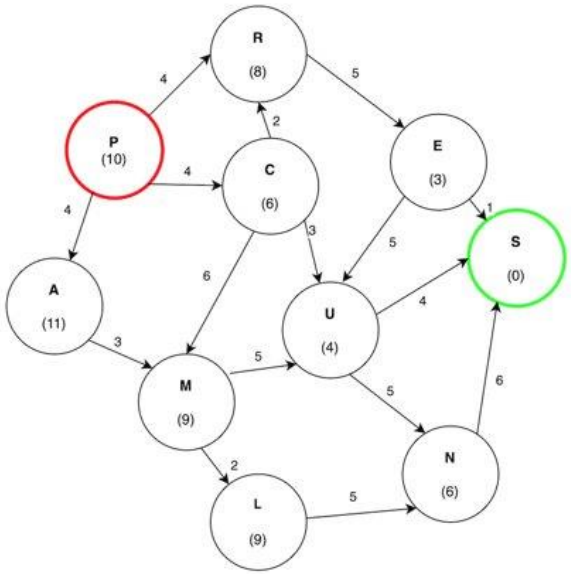
- Note that the least cost is at C

Closed list
P

Greedy Best-First Search - Example

- Expand node C



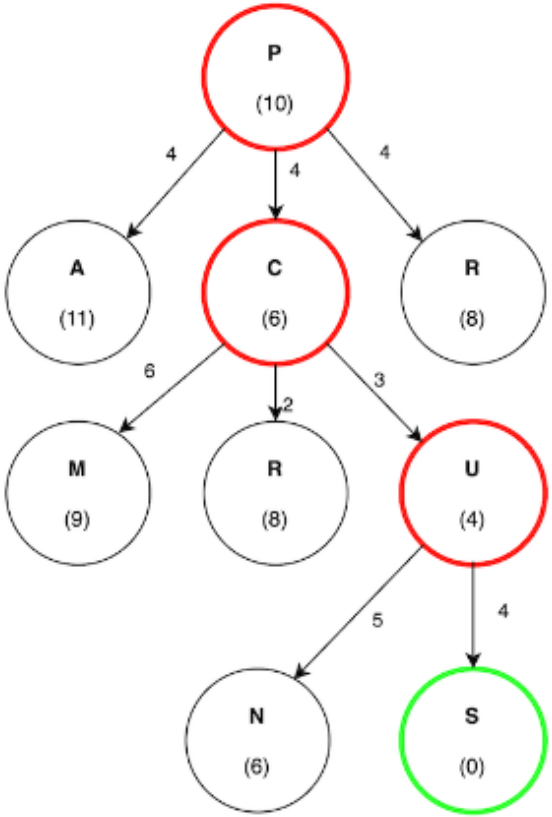


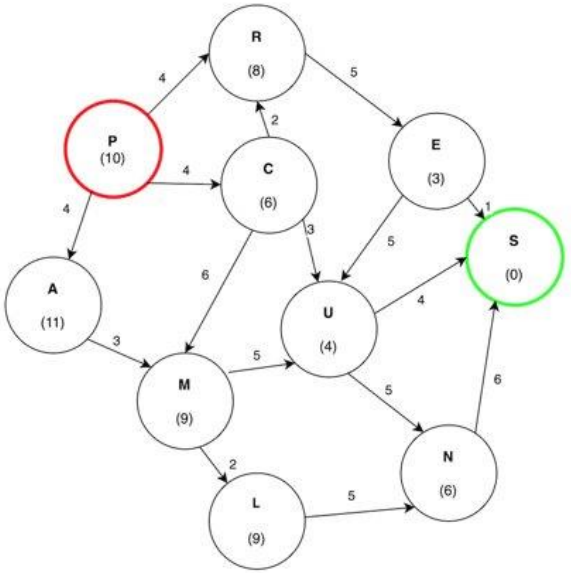
- Note that the least cost is at U

Closed list
P
C

Greedy Best-First Search - Example

- Expand node U



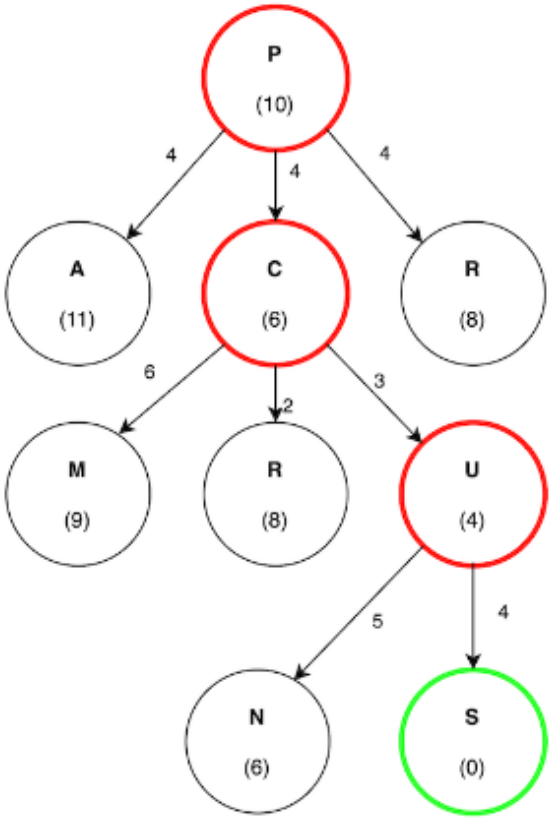


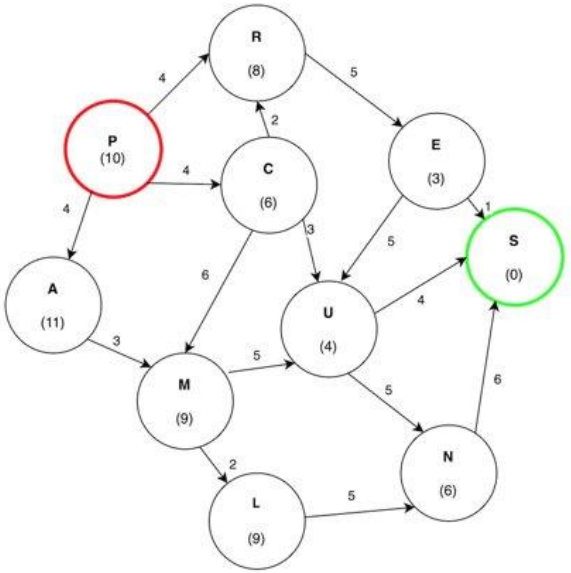
- Note that the least cost is at S

Closed list
P
C
U

Greedy Best-First Search - Example

- Expand node U



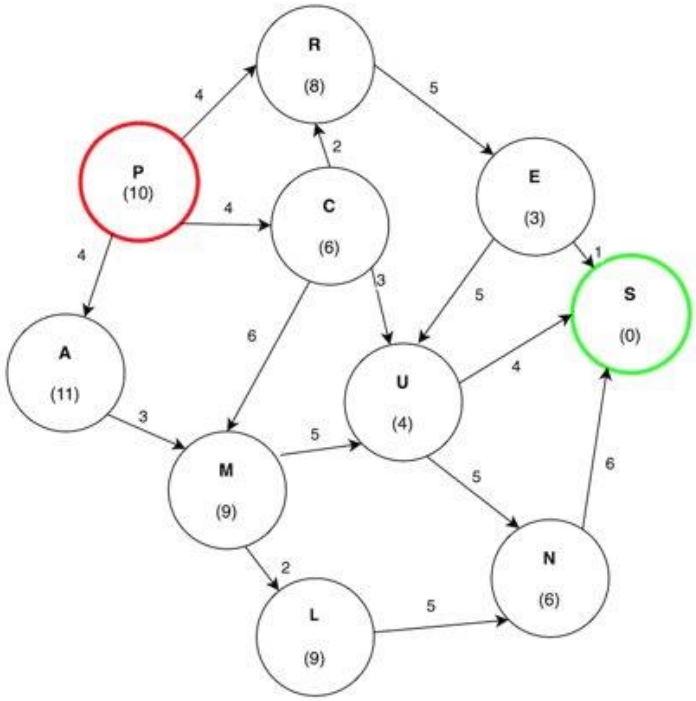


- Note that the least cost is at S (Goal)
- The total goal to the S evaluates to 11 (P -> C -> U -> S)

Closed list
P
C
U

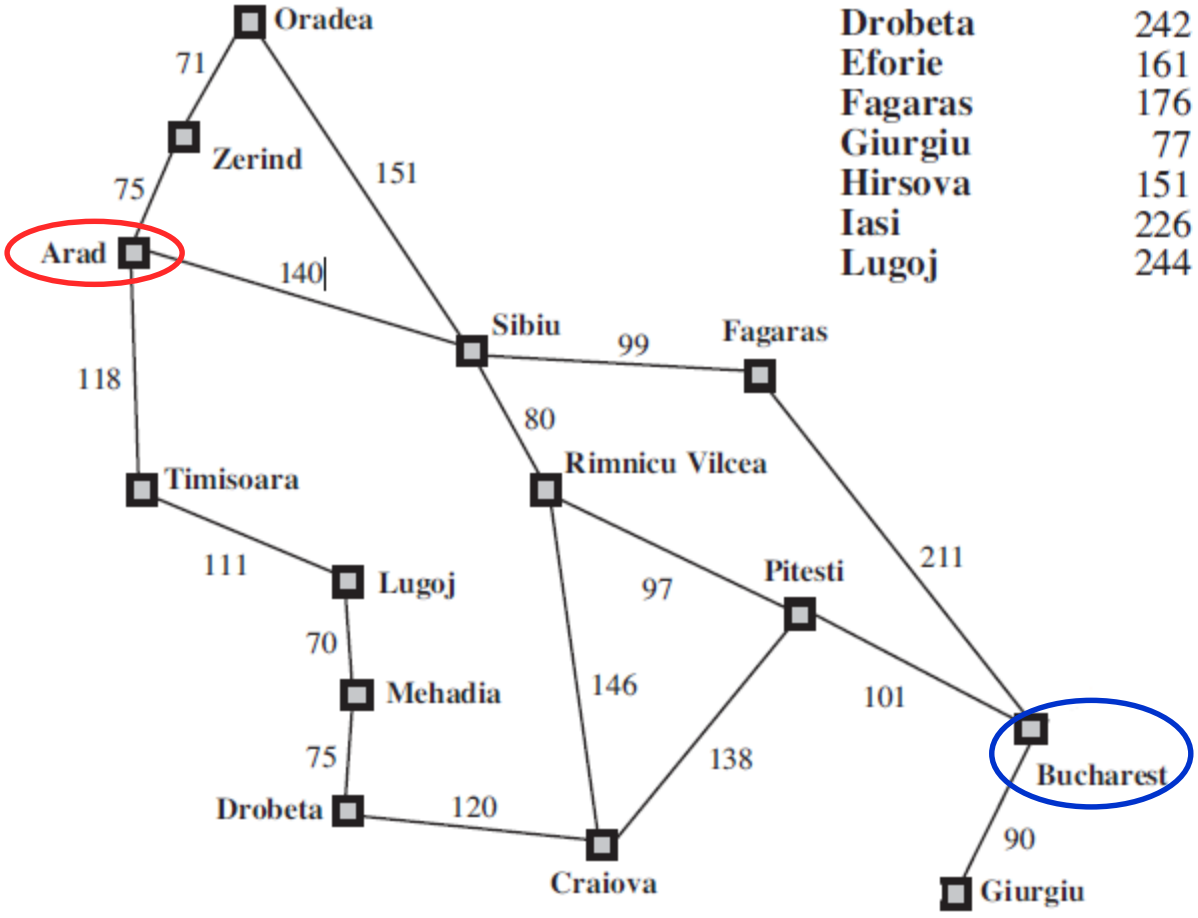
Greedy Best-First Search - Example

- Consider the actual path P->R->E->S
  - What is cost?
- Greedy Best-First Search ignored this path because it depends solely on the heuristic value



Greedy Best-First Search

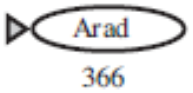
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



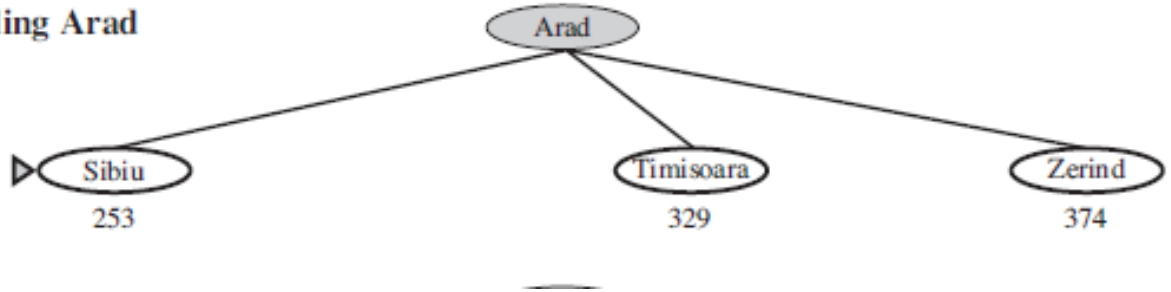
Greedy Best-First Search

- It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti
- This shows why the algorithm is called "greedy"—at each step it tries to get as close to the goal as it can

(a) The initial state



(b) After expanding Arad

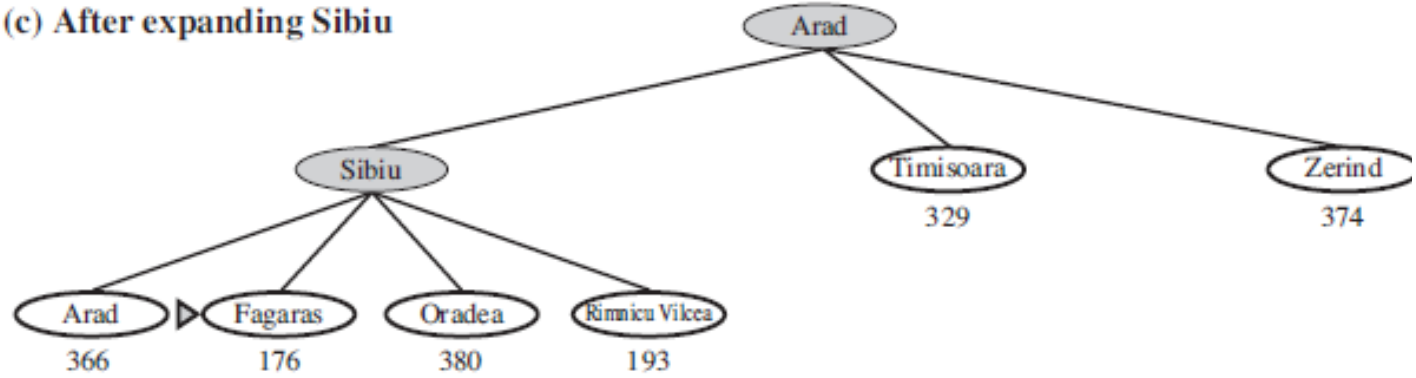


Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

(a) The initial state

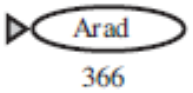
(b) After expanding Arad

(c) After expanding Sibiu



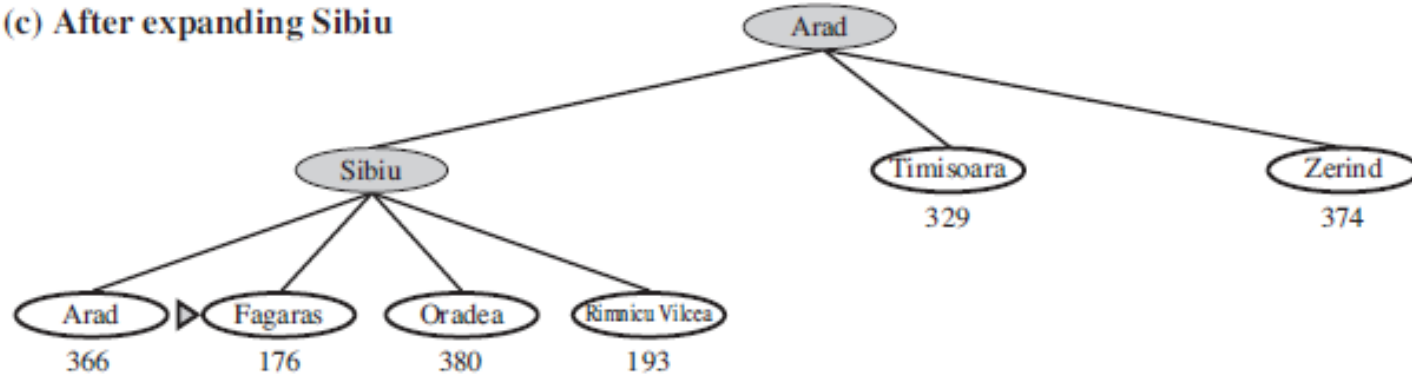
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

(a) The initial state

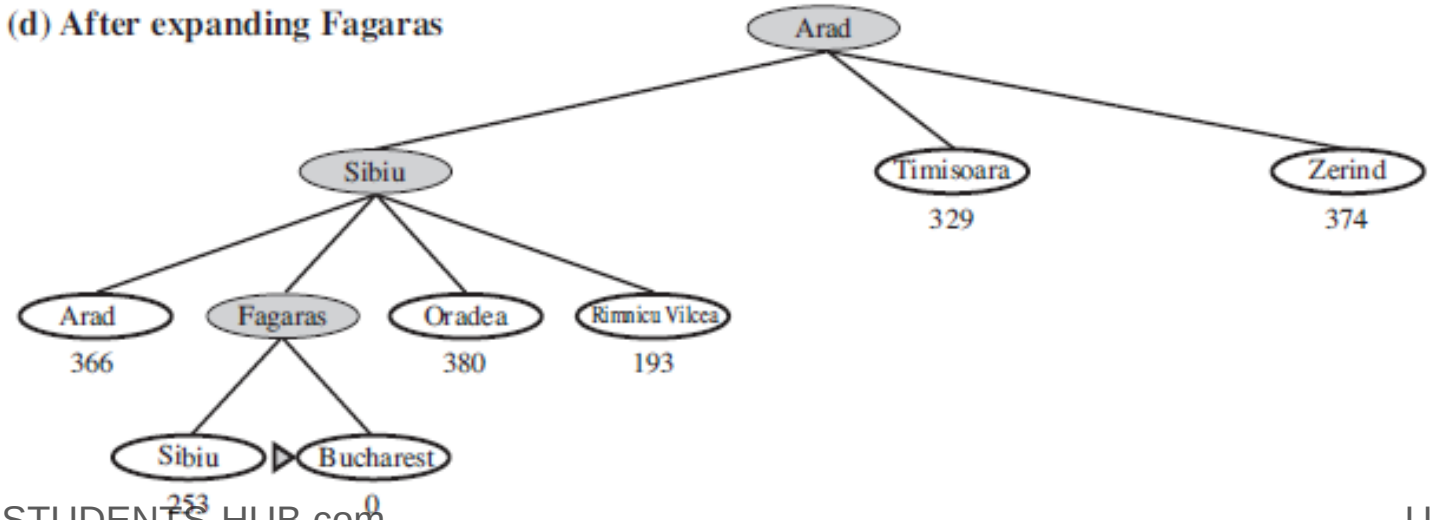


(b) After expanding Arad

(c) After expanding Sibiu



(d) After expanding Fagaras



Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Search

- Complete?

Greedy Best-First Search

- Complete?
  - No – can get stuck in loops, e.g., Iasi -> Neamt -> Iasi -> Neamt -> …
- Optimal?

Greedy Best-First Search

- Complete?
  - No – can get stuck in loops, e.g., Iasi -> Neamt -> Iasi -> Neamt -> …
- Optimal?
  - No (not guaranteed to find lowest cost solution).
- Time?

Greedy Best-First Search

- Complete?
  - No – can get stuck in loops, e.g., Iasi -> Neamt -> Iasi -> Neamt -> …
- Optimal?
  - No (not guaranteed to find lowest cost solution).
- Time?
  - $O(b^m)$, (in worst case). A good heuristic function can enhance the performance (m is max depth of search space).
- Space?

Greedy Best-First Search

- Complete?
  - No – can get stuck in loops, e.g., Iasi -> Neamt -> Iasi -> Neamt -> …
- Optimal?
  - No (not guaranteed to find lowest cost solution).
- Time?
  - $O(b^m)$, (in worst case). A good heuristic function can enhance the performance (m is max depth of search space).
- Space?
  - O(b m) keeps all nodes in memory.

A* ALGORITHM

A* search

- A variant of Best-First Search
- It evaluates nodes by combining *g(n)*, the cost to reach the node, and *h(n)*, the cost to get from the node to the goal:

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

- Can be seen also as a variant of Dijkstra's algorithm, but it differs in the evaluation evaluation function that is used to determine which node to explore next.

A* search

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

- *f(n) = estimated cost of the cheapest solution through n*
- *g(n) = cost to reach node n from the start state*
- *h(n) = cost to reach from node n to goal node (heuristic)*

- At each point during the search, only those node that have the lowest value of f(n) are expanded
- The algorithm terminates when the goal node is found

A* search

- A reasonable way to find the cheapest solution is to try first the node with the lowest value of g(n) + h(n)
- A* search is identical to the Uniform-Cost-Search (except A∗ uses g + h instead of g.)
- It is both complete and optimal

A* search - Algorithm

- Given a weighted graph, you need an open list (using priority queue) to store the next node to explore and a closed list to store the nodes that have been already visited.
- To find the lowest cost path, construct a search tree as follows:
  - *Initialize a tree with the root node being the start node S.*
  - *Remove the top node from the open list for exploration.*
  - *Add the current node to the closed list.*
  - *Add all nodes that have an incoming edge from the current node as child nodes in the tree.*
  - *Update the lowest cost to reach the child node.*
  - *Compute the evaluation function for every child node and add them to the open list.*

Optimality Conditions

- The heuristic function h(x) should be admissible heuristic: meaning it should never overestimate the cost of reaching a goal.
- *Overestimation* is the case in which we depend on h(n) alone. We might get to the shortest path but it might, on the other hand, cause the path to go longer than the optimal path.

Optimality Conditions

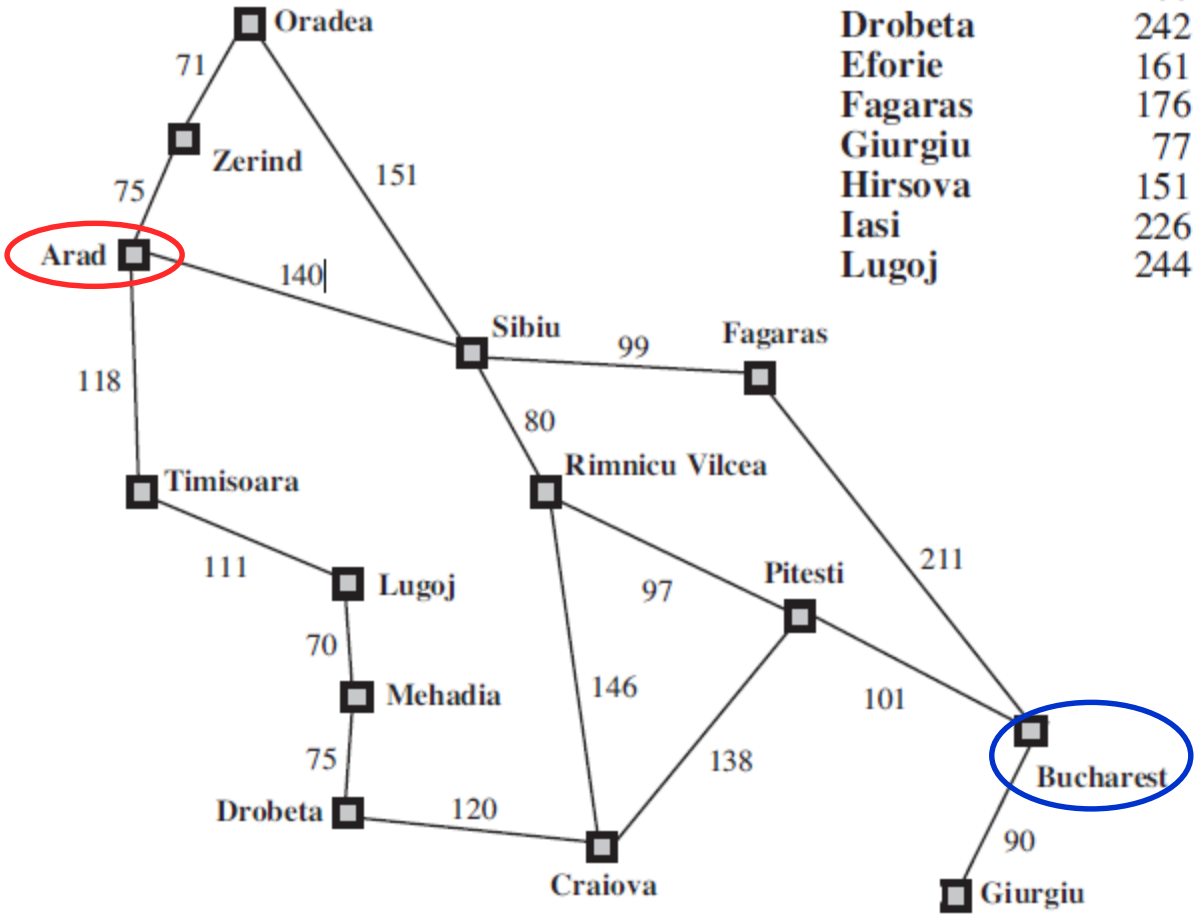
- Admissible heuristics are optimistic as they think the cost of solving the problem is less than it actually is
- The straight-line distance h_{SLD} is an example of admissible heuristics
- The Straight-line distance, on the route to Bucharest, is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate

Example – A* Search

E.g., hSLD(In(Arad))=366

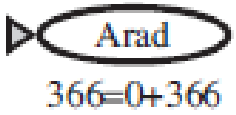
Straight-Line Distances

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



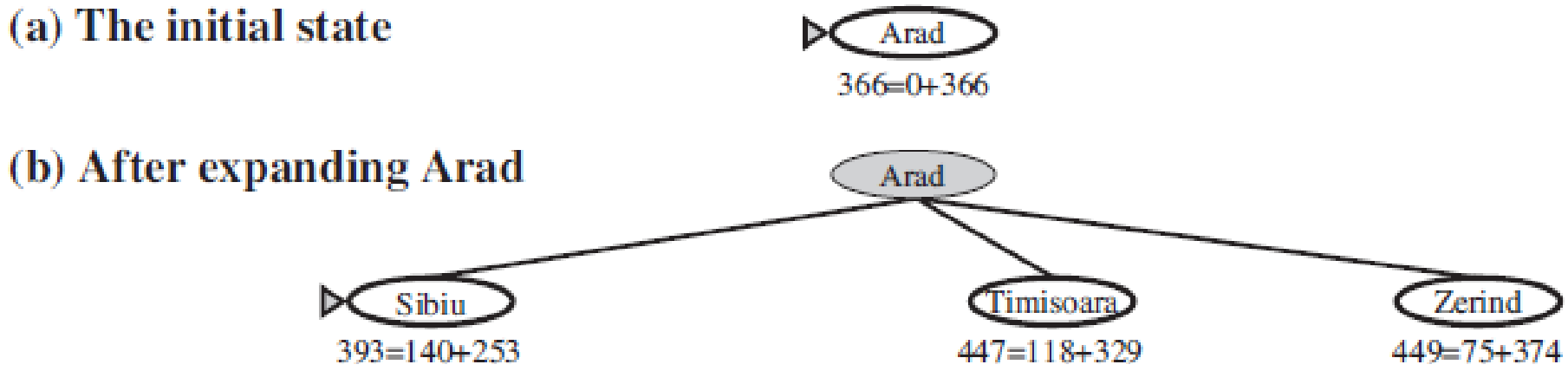
Example – A* Search

(a) The initial state



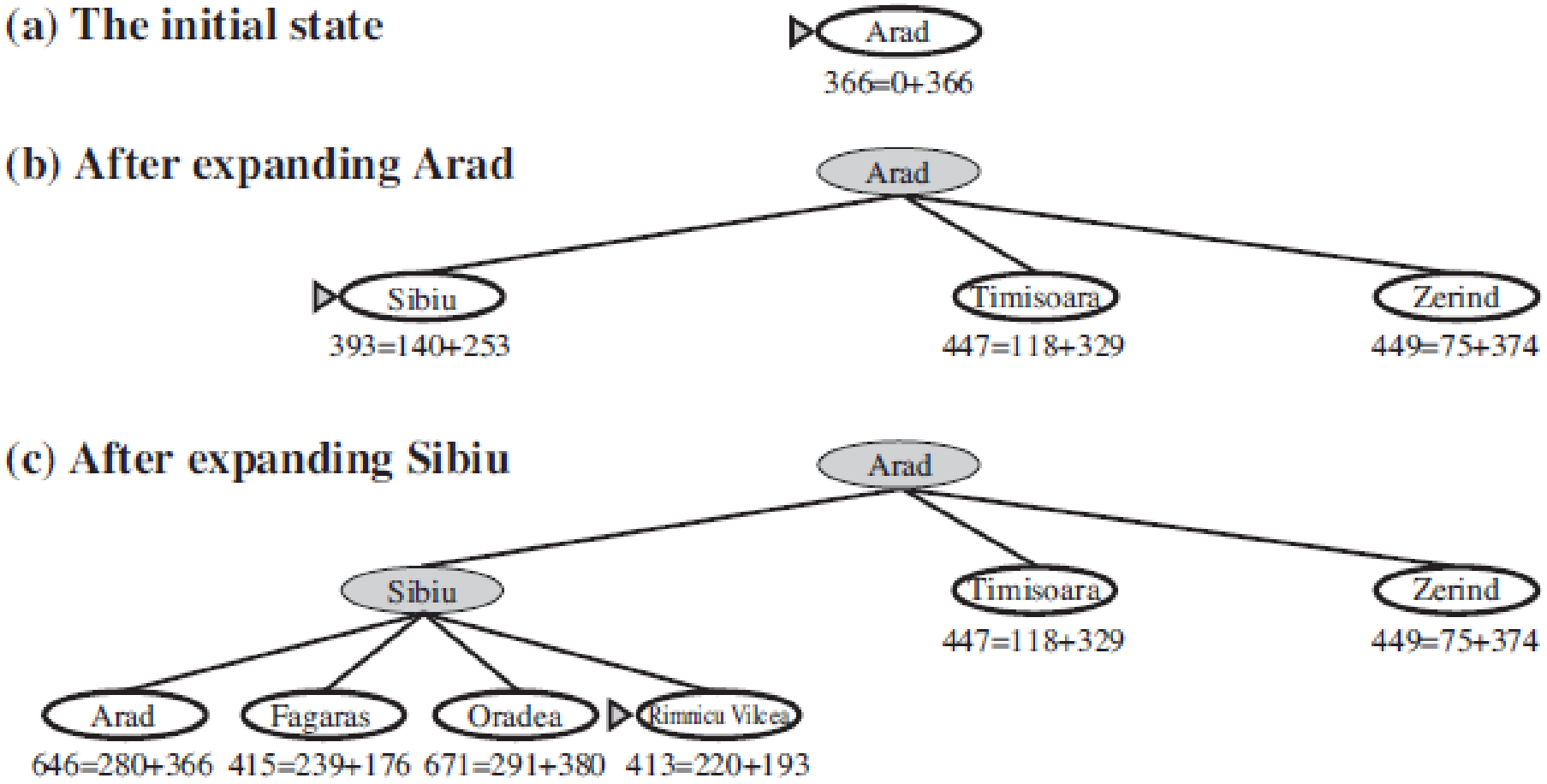
– The values shown here are the g(n) the step cost and h(n) which are the H_{sld}

Example – A* Search



- The values shown here are the g(n) the step cost and h(n) which are the H_{sld}

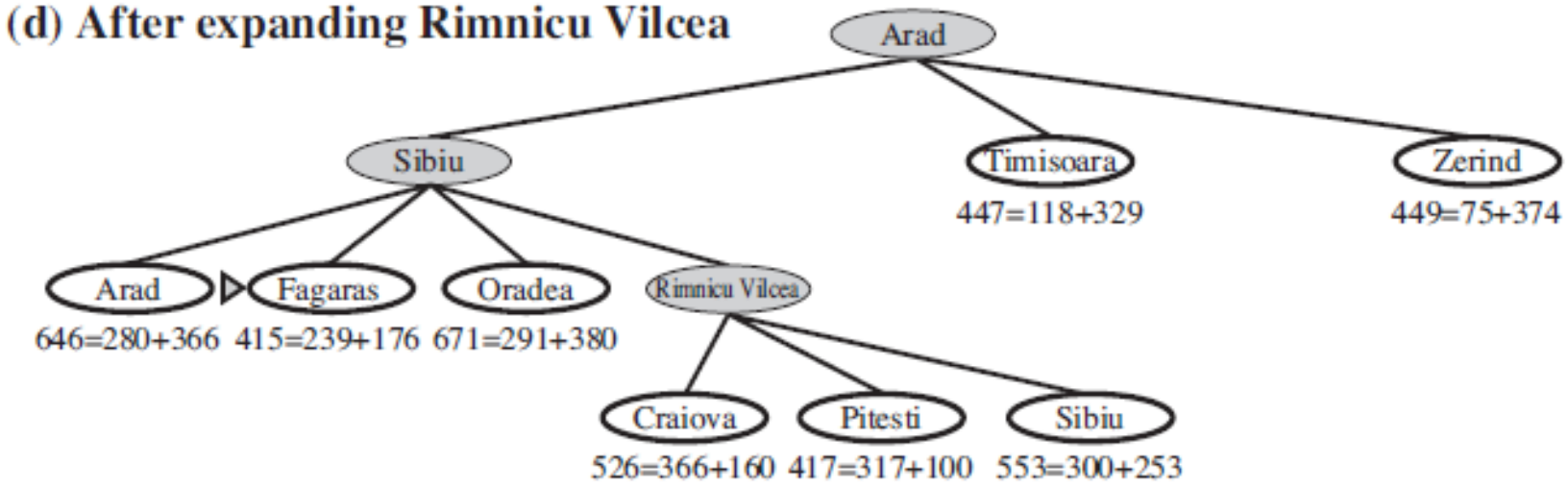
Example – A* Search



– The values shown here are the g(n) the step cost and h(n) which are the H_{sld}

Example – A* Search

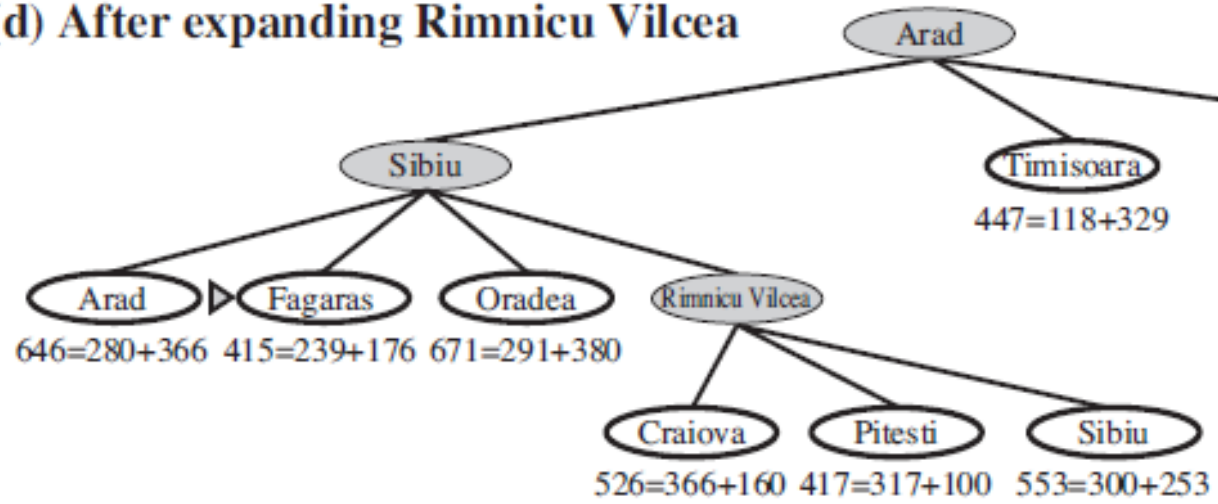
Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417). Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.



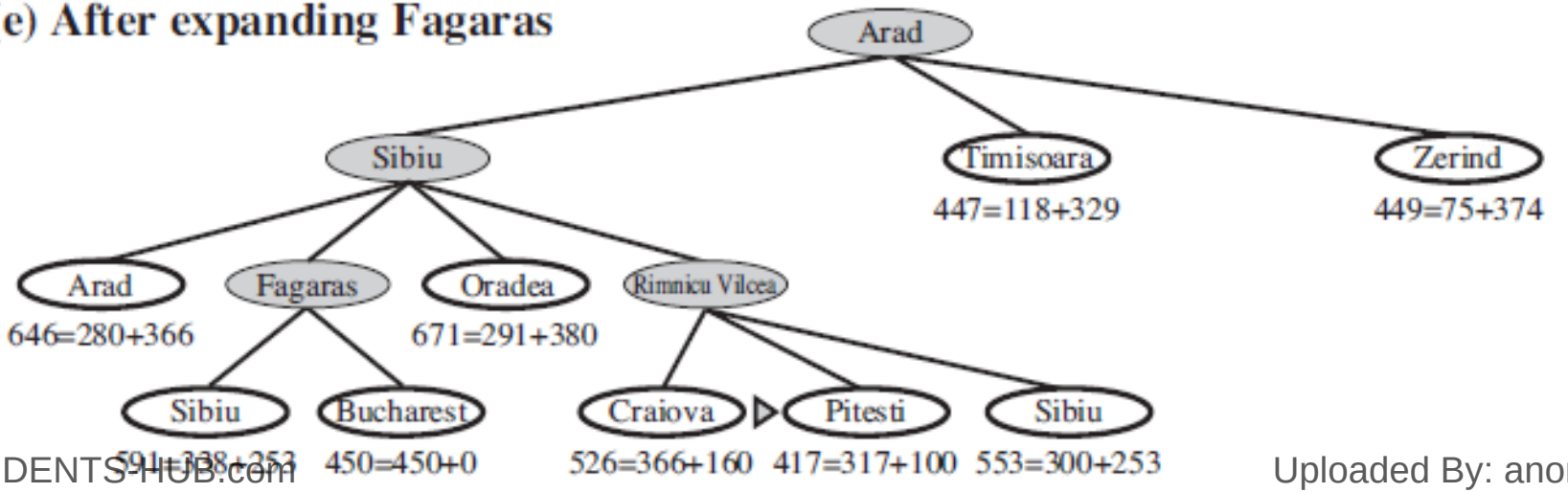
Example – A* Search

Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417). Another way to say this is that 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.

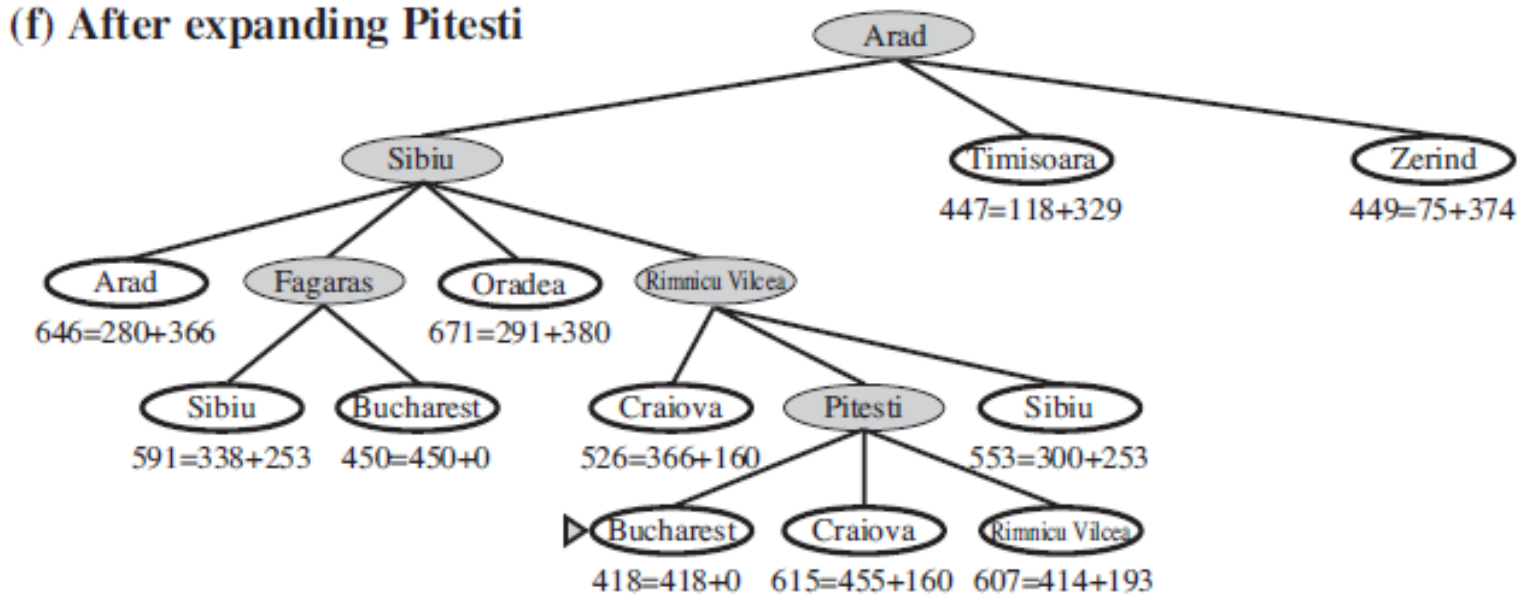
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras

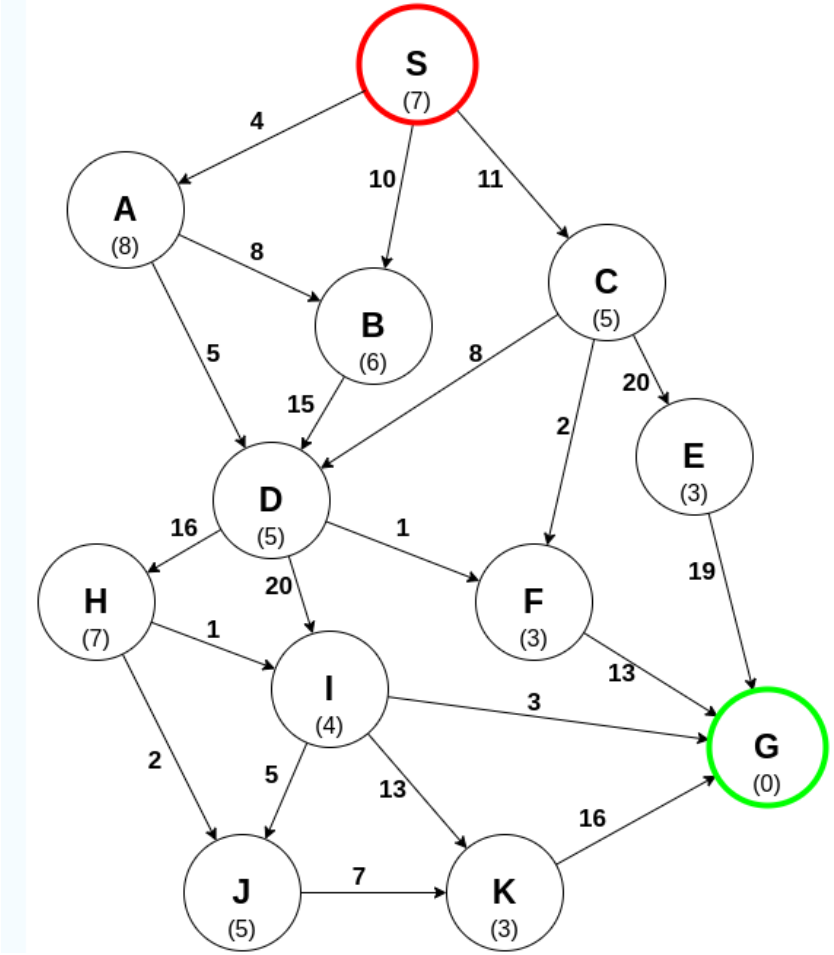


Example – A* Search



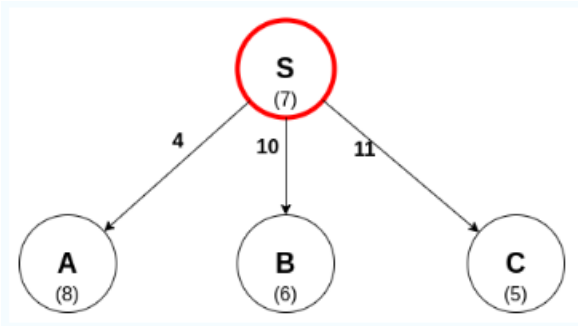
Example – A* Search

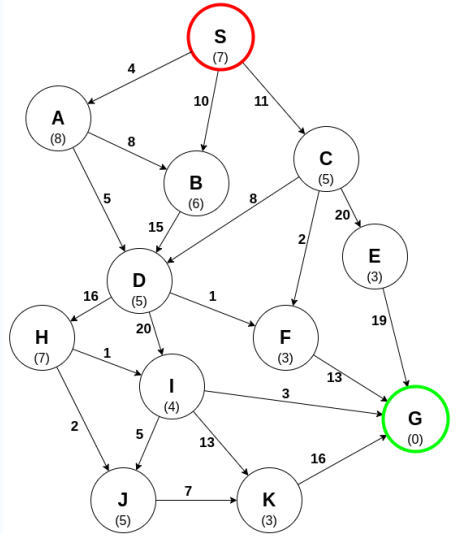
- Consider the following graph, we want to get from S to G



Example – A* Search

- Consider the following graph, we want to get from S to G





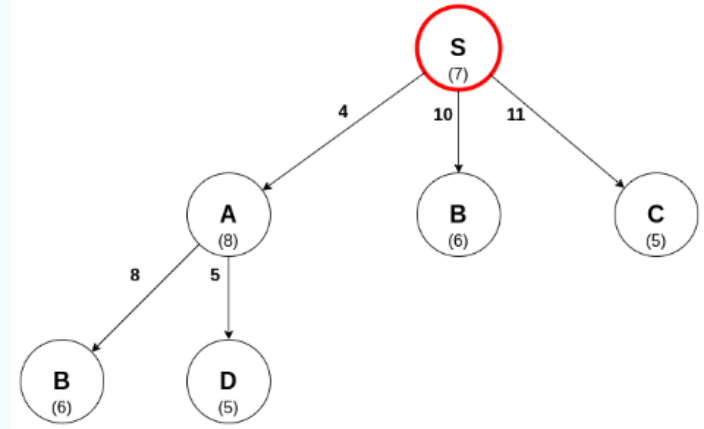
- A has the least cost

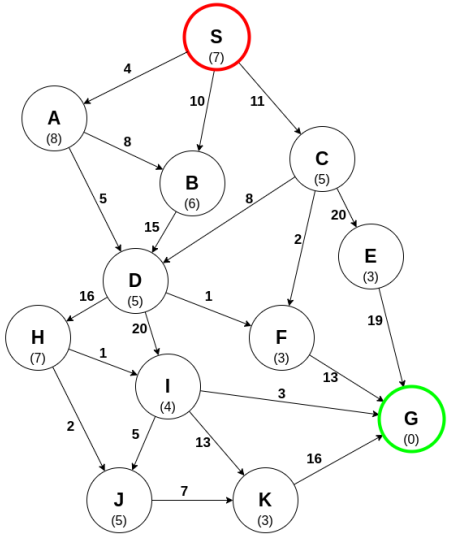
Node	Cost
A	12
B	16
C	16

Closed list
S

Example – A* Search

- Consider the following graph, we want to get from S to G





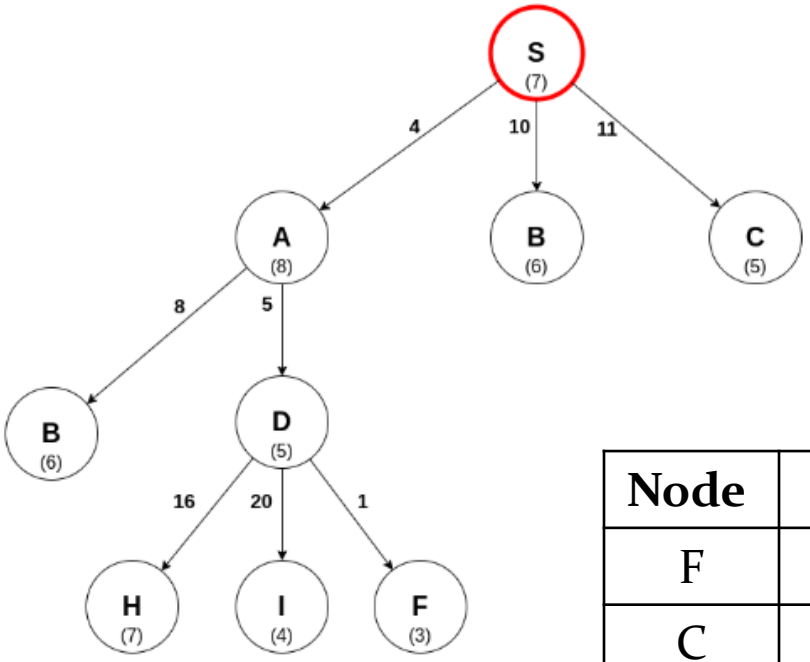
- D has the least cost

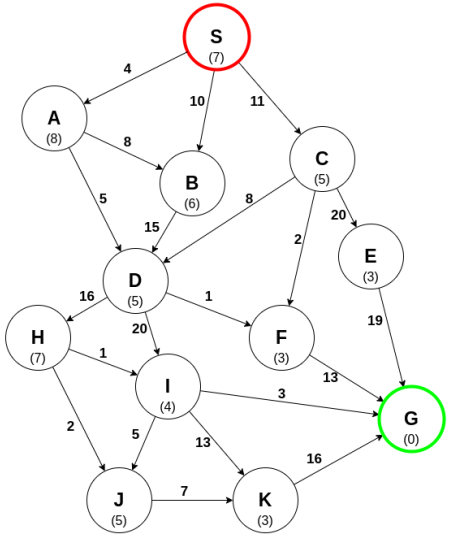
Node	Cost
D	14
C	16
B	16
B	18

Closed list
S
A

Example – A* Search

- Consider the following graph, we want to get from S to G





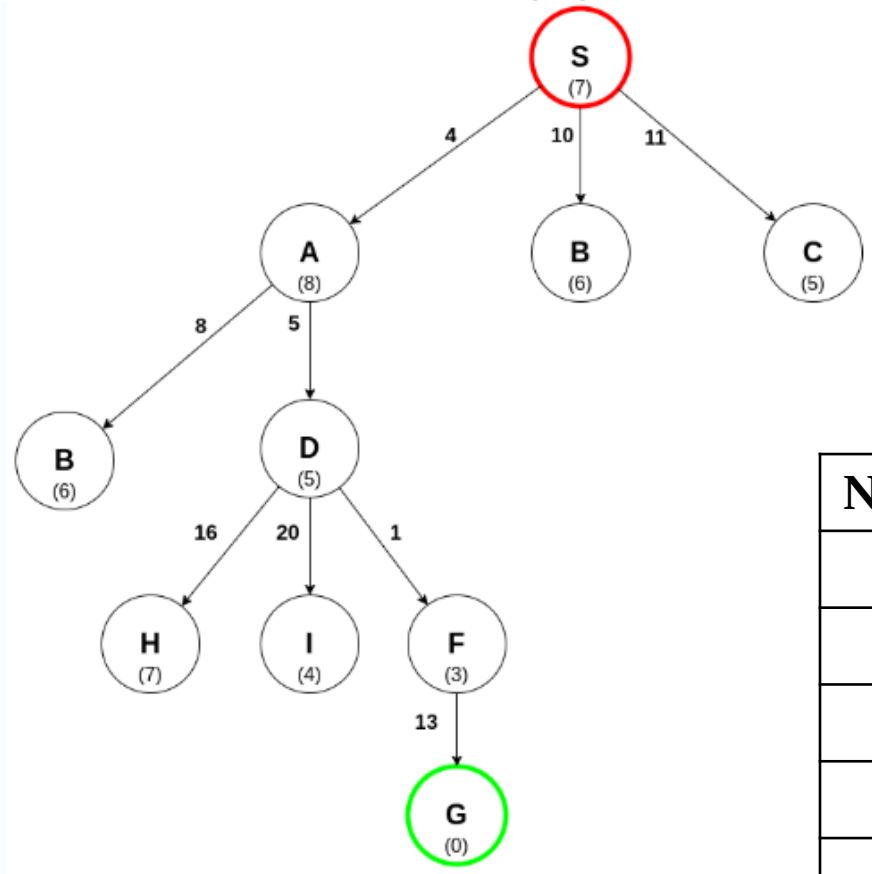
- Explore F

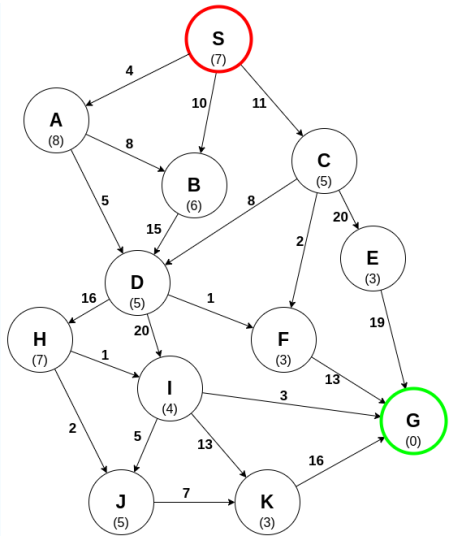
Node	Cost
F	13
C	16
B	16
B	18
H	32
I	33

Closed list
S
A
D

Example – A* Search

- Consider the following graph, we want to get from S to G





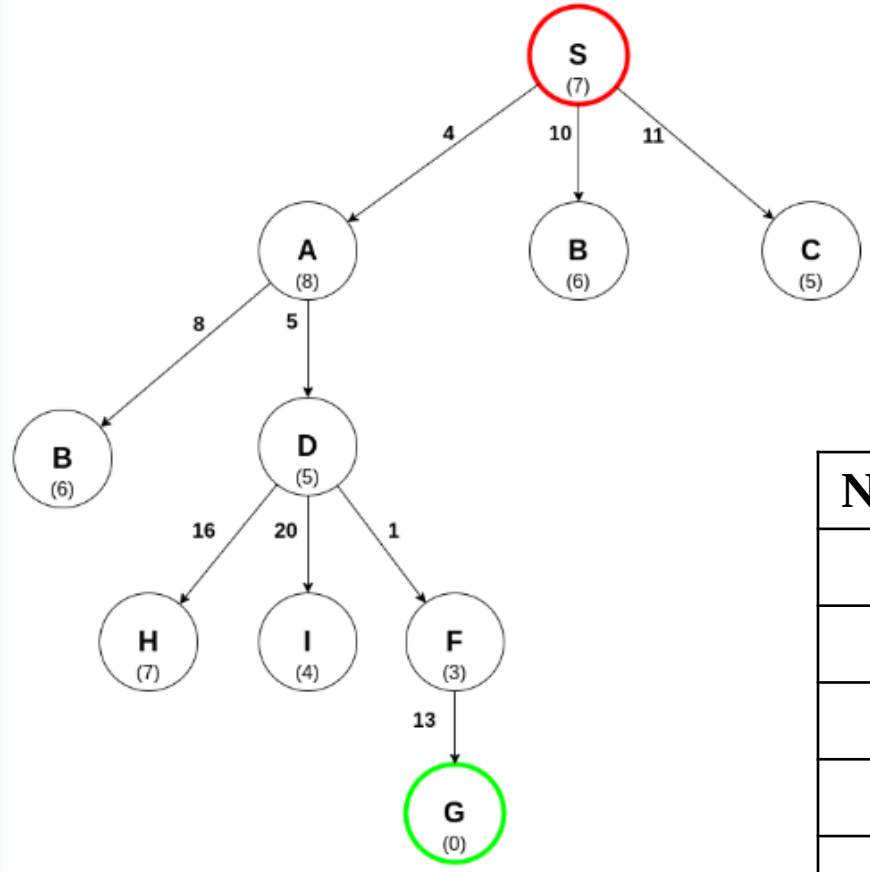
Node	Cost
B	16
C	16
B	18
G	23
H	32
I	33

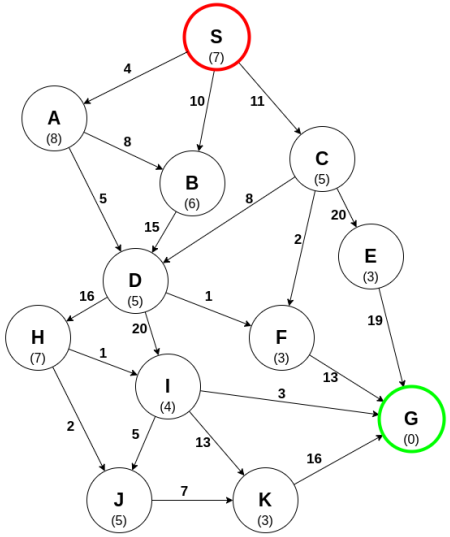
Closed list
S
A
D
F

- G has been found but there are nodes with less cost to be explored before so explore B

Example – A* Search

- Consider the following graph, we want to get from S to G





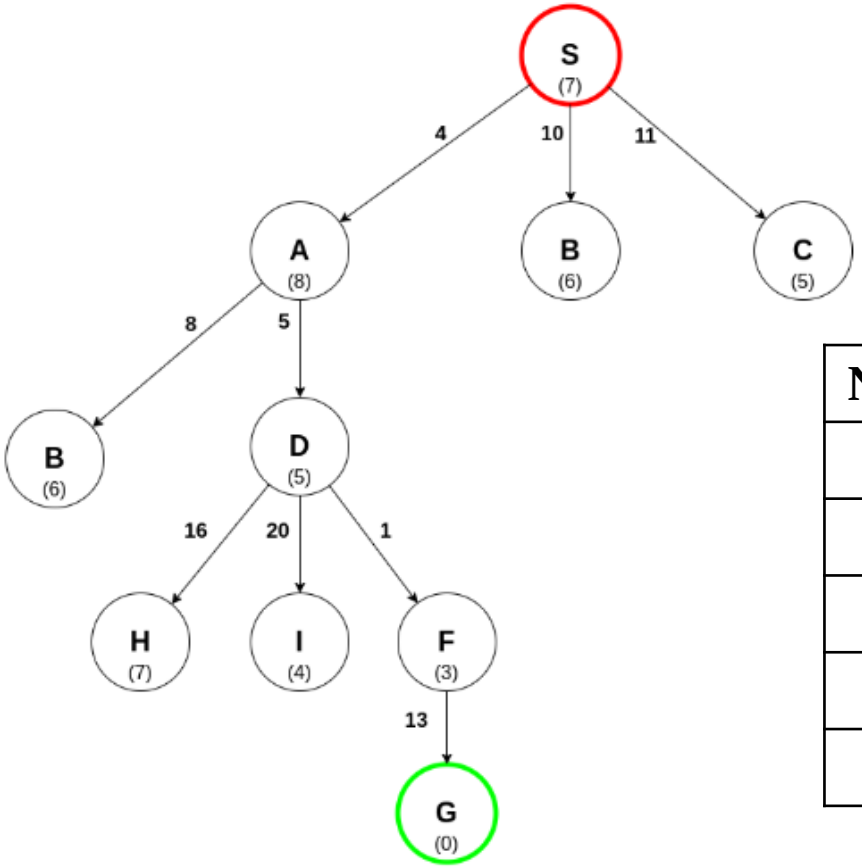
Node	Cost
C	16
B	18
G	23
H	32
I	33

Closed list
S
A
D
F
B

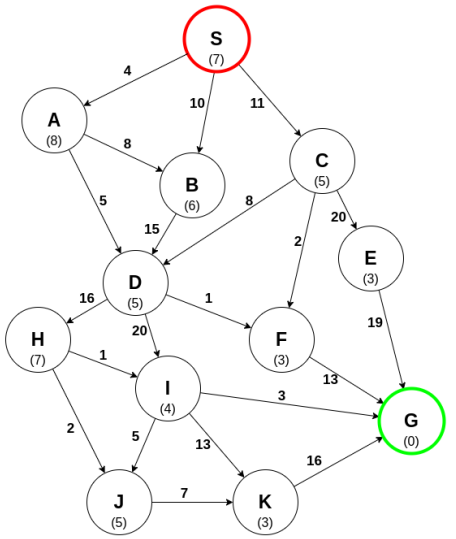
- Now Explore C

Example – A* Search

- Consider the following graph, we want to get from S to G



Node	Cost
B	18
G	23
H	32
I	33
E	36

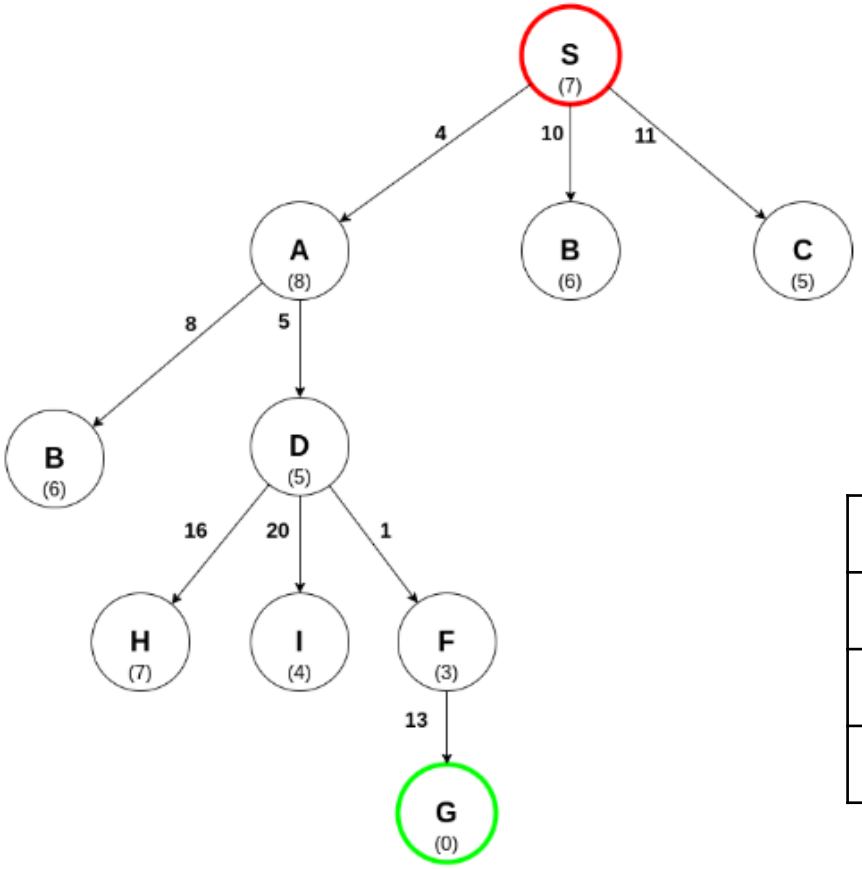


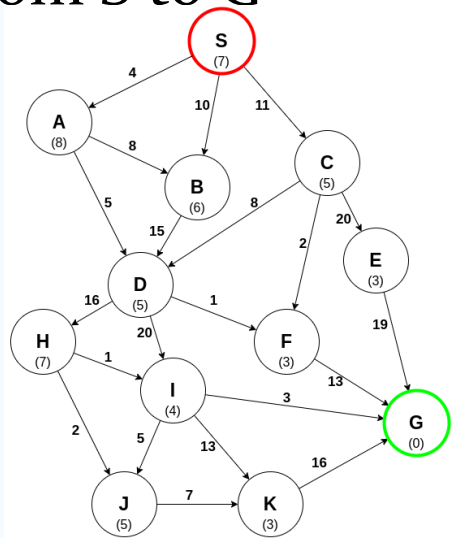
Closed list
S
A
D
F
B
G

- The next node in the open list is **B (again)**. Since **B** has already been explored, meaning a shortest path to **B** has been found. So it is not explored again and the algorithm continues to the next candidate

Example – A* Search

- Consider the following graph, we want to get from S to G





Node	Cost
H	32
I	33
E	36

Closed list
S
A
D
F
B
C
G

- The next node in the open list is **G** meaning the shortest path to **G** has been found

A* Search - Properties

- Complete: Yes
- Optimal: Yes
- A* is optimal if heuristic h(n) is **admissible** (tree version)
- Time complexity: exponential $O(b^d)$
- Space complexity: A* search keeps all nodes in memory. A* has the worst case of $O(b^d)$
  - This is because A* must keep track of the nodes evaluated so far as well the discovered nodes to be evaluated

Optimality of A* Search

- The first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have h=0) and all later goal nodes will be at least as expensive
- The fact that f-costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map

A* Search - optimality

- A* will find the optimal solution
- The first solution found is the optimal
- A* is optimally efficient
  - No other algorithm is guaranteed to expand fewer nodes than A*
- A* is not always “the best” algorithm
  - Optimality refers to the expansion of nodes, other criteria might be more relevant such as memory consumption
  - It generates and keeps all nodes in memory
  - Improved in variations of A*

A* Search - Algorithm

Algorithm 24 A^* Algorithm

Input: A graph

Output: A path between start and goal nodes

1: **repeat**
2: Pick n_{best} from O such that $f(n_{best}) \leq f(n), \forall n \in O$.
3: Remove n_{best} from O and add to C.
4: If $n_{best} = q_{goal}$, EXIT.
5: Expand n_{best}: for all $x \in \text{Star}(n_{best})$ that are not in C.
6: **if** $x \notin O$ **then**
7: add x to O.
8: **else if** $g(n_{best}) + c(n_{best}, x) < g(x)$ **then**
9: update x's backpointer to point to n_{best}
10: **end if**
11: **until** O is empty

Review - **Best-first search**

- Driven by the evaluation function $f(n)$ to guide the search.
- incorporates a **heuristic function** $h(n)$ in $f(n)$
- heuristic function measures a potential of a state (node) to reach a goal
- **Special cases** (differ in the design of evaluation function): – **Greedy search**
- $f(n) = h(n)$
  $f(n) = g(n) + h(n)$
- – **A* algorithm**
  + **iterative deepening** version of A* : **IDA***

Review - **Best-first search**

- The problem with the **greedy search** is that it can keep expanding paths that are already very expensive.
- The problem with the **uniform-cost search** is that it uses only past exploration information (path cost), no additional information is utilized
- **A* search**

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

  - $g(n)$ - cost of reaching the state
  - $h(n)$ - estimate of the cost from the current state to a goal
  - $f(n)$ - estimate of the path length
- • **Additional A*condition**: admissible heuristic $h(n) <= h^*(n)$ for all n

Review - **Best-first search**

- **Optimality of A***
- In general, a heuristic function $h(n)$:
  Can overestimate, be equal or underestimate the true distance of a node to the goal $h^*(n)$
- **Admissible heuristic condition**
- **– Never overestimate the distance to the goal:**
  $h(n) <= h^*(n)$ for all n
- **Example:** the straight-line distance in the travel problem never overestimates the actual distance

IDA*

Memory-Bounded Heuristic Search-IDA*

- Memory requirements of A* search is reduced by adapting the idea of iterative deepening to the heuristic research
- This results in iterative-deepening A* (IDA*)
- The main difference between IDA* and standard iterative deepening is that the cut-off used is the f-cost (g+h) rather than the depth; at each iteration, the cut-off value is the smallest f-cost of any node that exceeded the cut-off on the previous iteration

IDA*

- **Solves minimum cost-path problems with heuristics**
- **Iterative deepening version of A***
- **Idea:**
  - Performs **limited-cost depth-first search** for the current evaluation function limit
    - Keeps expanding nodes in the depth-first manner up to the evaluation function limit
- Progressively increases the **evaluation function limit** (instead of the depth limit)

IDA*

- IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes
- It suffers from the same difficulties with real valued costs as does the iterative version of uniform-cost search

IDA* - Algorithm

- In the first iteration, the value "**f-cost limit**" – **cut-off value** $f(n_o) = g(n_o) + h(n_o) = h(n_o)$, is determined where no is the start node
- Expand nodes using the **DFS** and backtrack whenever f(n) for an expanded node n exceeds the cut-off value
- If this search does not succeed, determine the **lowest f-value** among the nodes that were visited but not expanded
- Use this f-value as the **new limit value – cut-off value** and do another depth-first search
- This procedure is repeated until a goal node is found

RECURSIVE-BFS

Recursive-Best First Search (RBFS)

- A simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space
- It aims at solving the memory issue inherits with A* search
- Idea: Similar to iterative deepening search, but the cutoff is *f-cost* (g+h) at each iteration, rather than depth first

Recursive-Best First Search (RBFS)

- Recursive-BFS and Simple Memory Bounded (SMA*) are called memory bounded heuristic search algorithms

Recursive-Best First Search (RBFS)

- Aims to find the best alternative over fringe nodes, which are not children
- As the recursion is resolved, RBFS replaces the f-value of each node along the path with a **backed-up value**—the best f-value of its children
- In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth re-expanding the subtree at some later time
- It stores the best alternative over fringe nodes, which are not children

Recursive-Best First Search (RBFS)

- 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.

RBFS - Algorithm

function RBFS (*problem, node, f-limit*) **returns** a solution or failure and a new f-cost limit

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*) *successors* ← [ ]
for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

add CHILD-NODE(*problem*, *node*, *action*) into *successors*
if *successors* is empty **then**
    return *failure*, ∞
for each s **in** *successors* **do**

/* update f with value from previous search, if any */

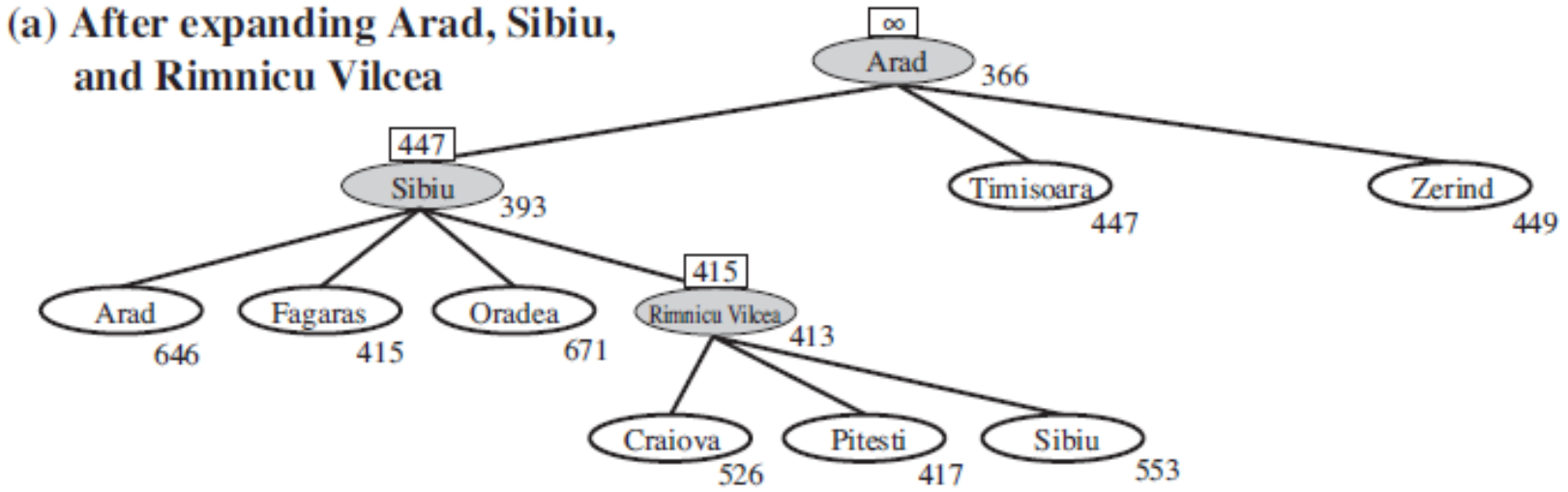
s.f ← max (s.g + s.h, node.f))

loop do

best ← the lowest f-value in *successors*
if *best*.f > *f-limit* **then**
    return *failure, best.f*
alternative ← the second lowest *f*-value among *successors result, best.f* ← RBFS (*problem, best*, min(*f-limit*,*alternative*))

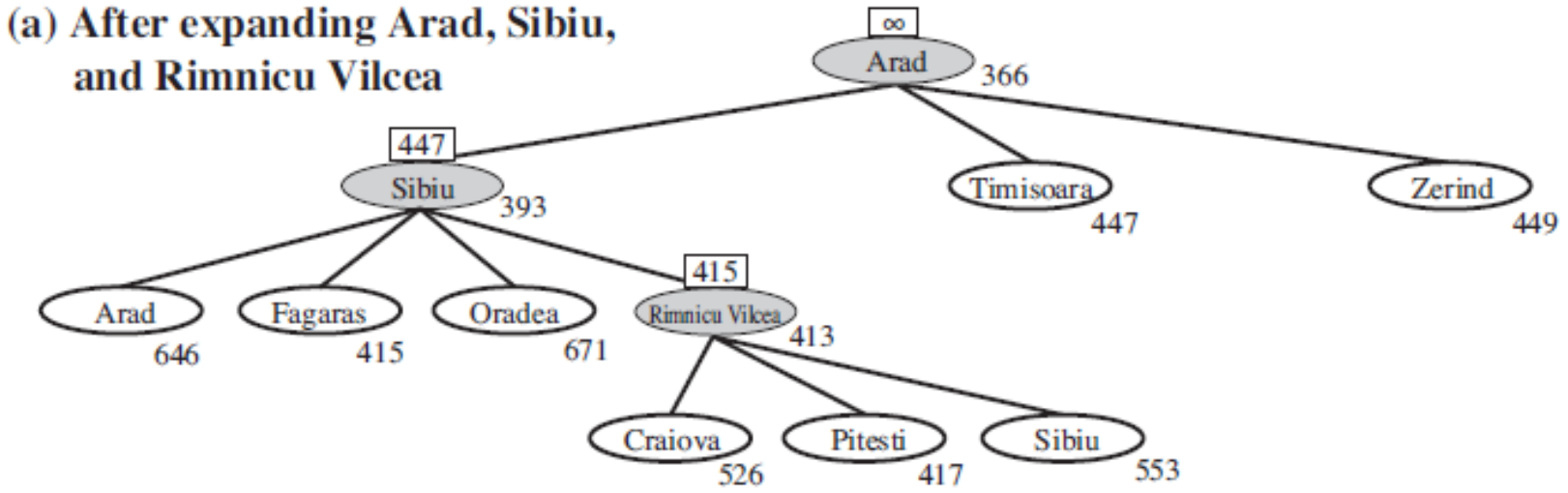
if *result* ≠ *failure* **then**
    return *result*

Recursive-Best First Search (RBFS)



- **Shortest path to Bucharest.**
- **The f-limit value for each recursive call is shown at the top of each node**
- **Every node is labelled with its f-cost**

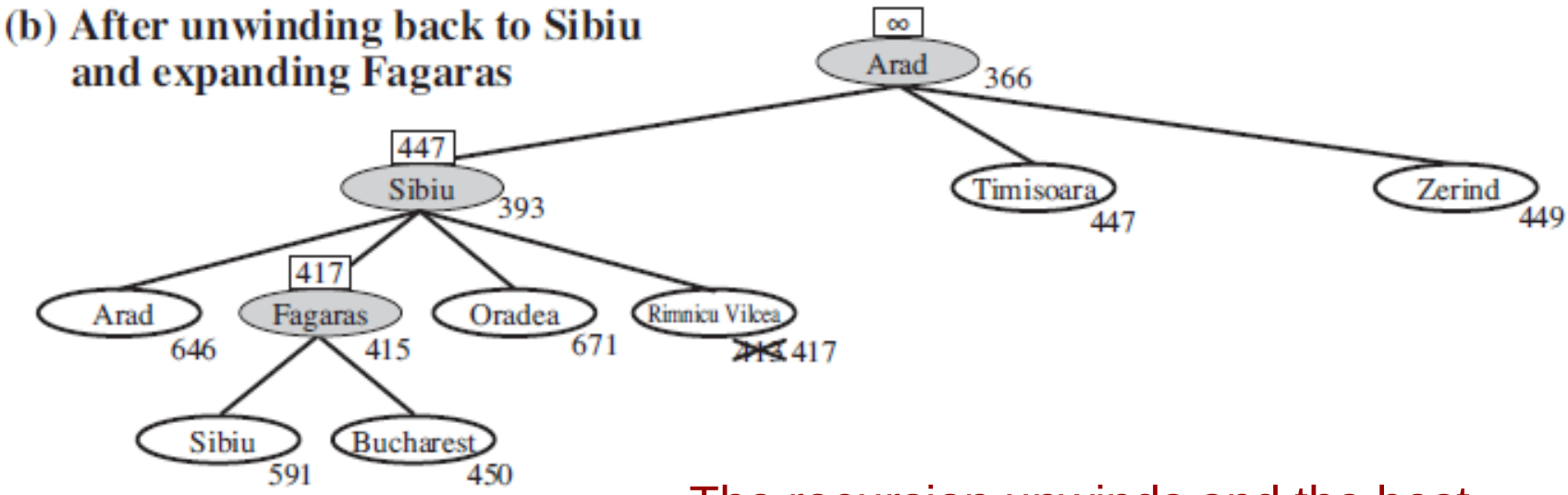
Recursive-Best First Search (RBFS)



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).

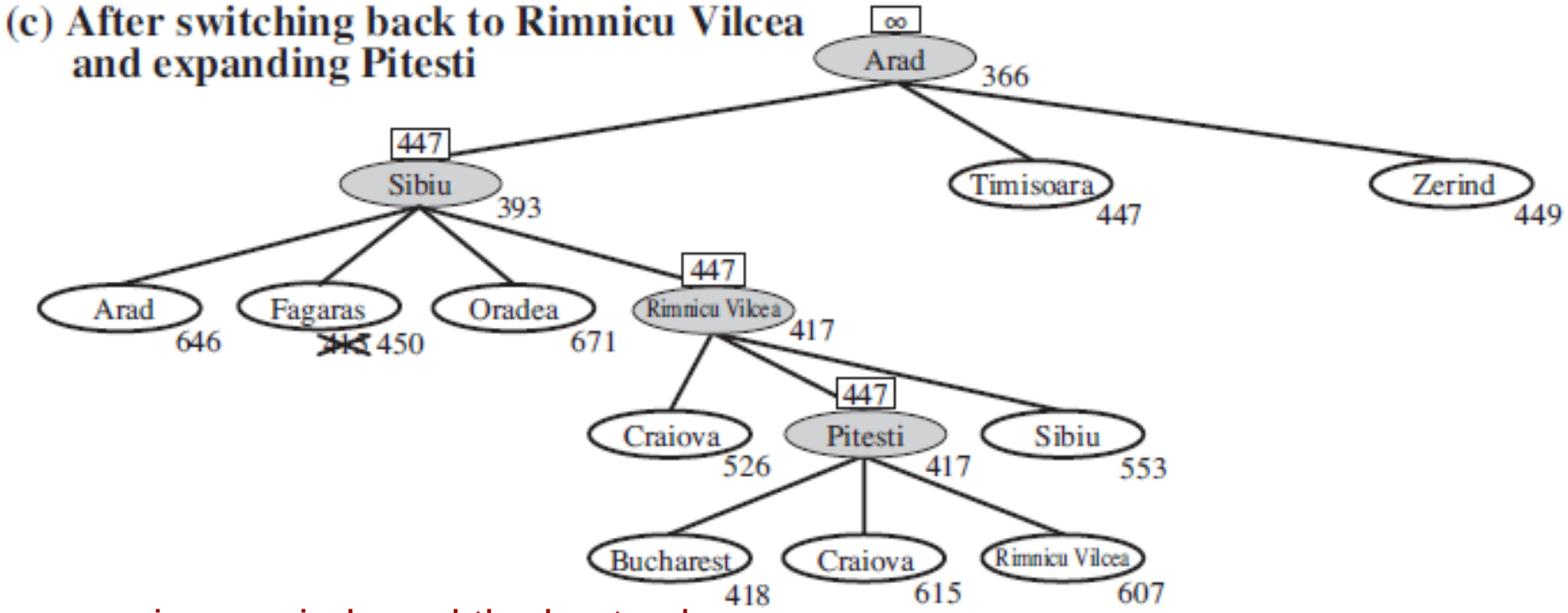
- **Shortest path to Bucharest.**
- **The f-limit value for each recursive call is shown at the top of each node**
- **Every node is labelled with its f-cost**

Recursive-Best First Search (RBFS)



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 value of 450.

Recursive-Best First Search (RBFS)



The recursion unwinds and the best value of the 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.

Recursive-Best First Search (RBFS)

- RBFS is somewhat more efficient than IDA*
- It still suffers from excessive node regeneration
- RBFS follows the path via Rimnicu Vilcea, then “changes its mind” and tries Fagaras, and then changes its mind back again. These mind changes occur because every time the current best path is extended, its f-value is likely to increase—h is usually less optimistic for nodes closer to the goal

Recursive-Best First Search (RBFS)

- Each mind change corresponds to an iteration of IDA*
- Complete: Yes, as A* algorithm
- Like A* tree search, RBFS is an **optimal** algorithm if the heuristic function h(n) is admissible
- Its space complexity is **linear** in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize**: it depends the accuracy of the heuristic function and on how often the best path changes as nodes are expanded**
- Memory O(bd)

SMA*

SMA*

- Stands for Simplified Memory-Bounded A*
- 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 first delete the oldest node(s) first.
- SMA* finds the optimal reachable solution given the memory constraint.
- But time can still be exponential.

SMA*

- SMA* proceeds just like A*, expanding the best leaf until memory is full
- Until here, it cannot add a new node to the search tree without dropping an old one
- SMA* always drops the *worst* leaf node—the one with the highest f-value from the fringe

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
```

SMA*

- SMA* is complete if there is any reachable solution—that is, if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes)
- It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution

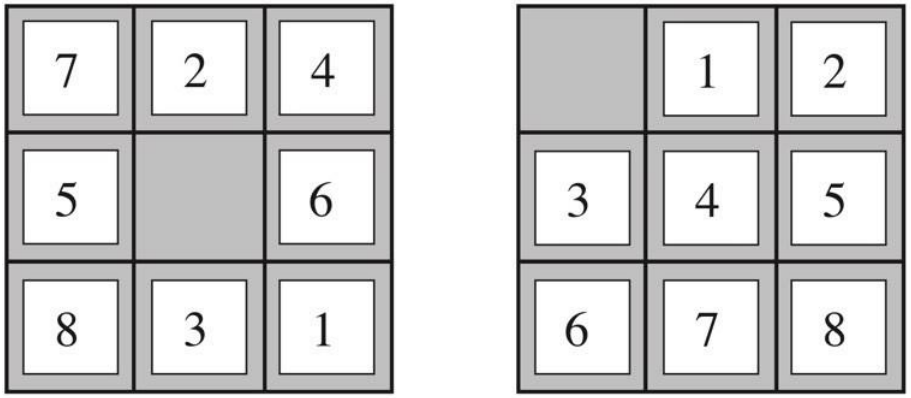
SMA*

- SMA* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the explored set
- Often is better than A* and IDA* (trade-off between time and space requirements)

ADMISSIBLE HEURISTICS

Admissible Heuristic

- The 8-puzzle is one of the earliest heuristic search problems
- How one you invent a good admissible heuristic function?

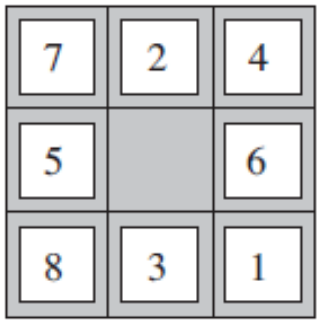


Heuristic Functions

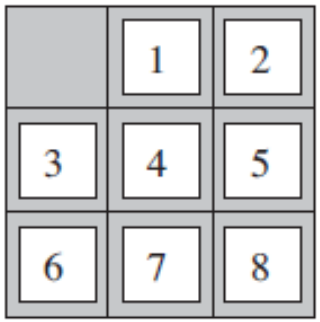
- The 8-puzzle is one of the earliest heuristic search problems
- There are 9!/2 = 181,400 reachable states.
- So the search can easily keep all of them in the memory.
- Consider a 15-puzzle, there are 16!/2 state – which is over 10 trillion states (expensive on the memory?)

Heuristic Functions

- There is a need here for a good admissible heuristic function.
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the sum of the distances of the tiles from their goal positions
  - tiles cannot move diagonally; the distance is the sum of the horizontal and vertical distances (city-block distance or Manhattan distance)
  - *h2* is also admissible because all any move can do is move one tile one step closer to the goal



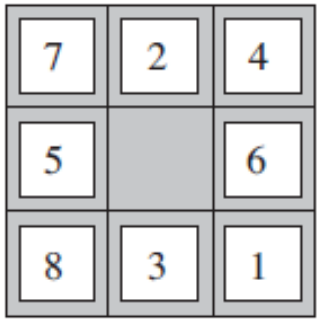
Start State



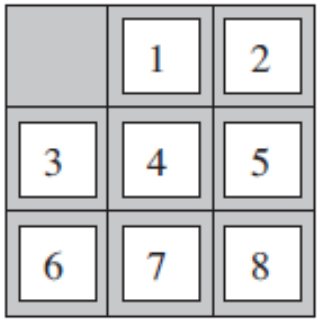
Goal State

Heuristic Functions

- There is a need here for a good admissible heuristic function.
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the sum of the distances of the tiles from their goal positions
  - tiles cannot move diagonally; the distance is the sum of the horizontal and vertical distances (city-block distance or Manhattan distance)
  - *h2* is also admissible because all any move can do is move one tile one step closer to the goal



Start State

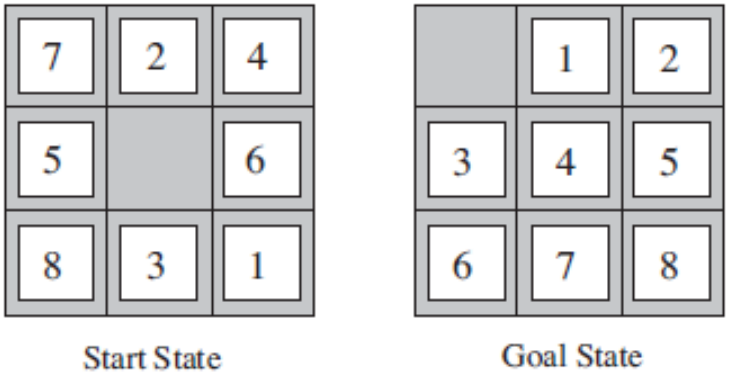


Goal State

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

Heuristic Functions

- There is a need here for a good admissible heuristic function.
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the sum of the distances of the tiles from their goal positions
  - tiles cannot move diagonally; the distance is the sum of the horizontal and vertical distances (city-block distance or Manhattan distance)
  - *h2* is also admissible because all any move can do is move one tile one step closer to the goal



A typical instance of the 8-puzzle. The solution is 26 steps long

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

Heuristic Functions

- h_2 is also admissible because all any move can do is move one tile one step closer to the goal
- Tiles 1 to 8 in the start state give a Manhattan distance of h_2 = 3+1 + 2 + 2+ 2 + 3+ 3 + 2 = 18
- Neither of these overestimates the true solution cost, which is 26

Heuristic Functions

- One way to characterise the quality of a heuristic is the **effective branching factor** b*
- If the total number of nodes generated by A* for a particular problem is N and the c_{depth} is d, then b* is the branching factor that a uniform tree of depth d would have to have in order to contain N + 1 nodes
- Thus, $N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$
- For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92

Heuristic Functions

- Experimental measurements of b* on a small set of problems can provide a good guide to the heuristic's overall usefulness
- A well designed heuristic would have a value of b* close to 1, allowing fairly large problems to be solved at reasonable computational cost
- The performance of heuristic search algorithms depends on the quality of the heuristic function. One can sometimes construct good heuristics by relaxing the problem definition, by storing precomputed solution costs for subproblems in a pattern database, or by learning from experience with the problem class

Summary

- The evaluation function for a node n is: f(n) = g(n) + h(n)
- If only g(n) is used, we get uniform-cost search
- If only h(n) is used, we get greedy best-first search
- If both g(n) and h(n) are used we get best-first search
- If both g(n) and h(n) are used with an admissible heuristic we get A∗ search
- A consistent heuristic is admissible but not necessarily vice versa

Summary

- Admissibility (always finding the shortest solution) is sufficient to guarantee solution optimality for tree search
- Consistency (narrowing options and selecting the best one) is required to guarantee solution optimality for graph search
- Heuristic search usually brings drastic improvement over uninformed search

References

- S. Russell and P. Norvig: *Artificial Intelligence: A Modern Approach* Prentice Hall, *2003, Second Edition*
- Lecture notes: Mustafa Jarrar's COMP338 – Artificial Intelligence http://www.jarrar.info/courses/AI/Jarrar.LectureNotes.Ch3.InformedSearch.pdf
- Moonis Ali: Lecture Notes on Artificial Intelligence http://cs.txstate.edu/~ma04/files/CS5346/SMA%20search.pdf
- Max Welling: Lecture Notes on Artificial Intelligence https://www.ics.uci.edu/~welling/teaching/ICS175winter12/A-starSearch.pdf
- Kathleen McKeown: Lecture Notes on Artificial Intelligence http://www.cs.columbia.edu/~kathy/cs4701/documents/InformedSearch-AR-print.ppt
- Franz Kurfess: Lecture Notes on Artificial Intelligence http://users.csc.calpoly.edu/~fkurfess/Courses/Artificial-Intelligence/F09/Slides/3-Search.ppt
- Nulifer Ondor: Lecture notes on Artificial Intelligence https://pages.mtu.edu/~nilufer/
- Milos Hauskrecht: Lecture notes on Artificial Intelligence https://people.cs.pitt.edu/~milos/