Informed Search

Informed Search

search strategy:
which node (in queue) to expand

- uninformed
  - distance to goal not taken into account
- informed
  - information about cost to goal is taken into account
  - usually heuristic, e.g.,
    estimate „value“ of the constellation of chess pieces

Best-first search

- evaluation function for each node
  - Heuristic Functions
    - f: States → Numbers
    - f(n): expresses the quality of the state n
    - allows to express problem-specific knowledge
    - can be used in a generic way
- expand most desirable unexpanded node first
  - queuing based on f(n)
  - order the nodes in fringe in decreasing order
- special cases:
  - greedy best-first search
  - A* search

Romanian path finding problem

Based e.g. on GPS info.
No map needed.

Searching for good path from Arad to Bucharest,
what is a reasonable “desirability measure” to expand nodes on the fringe?

Greedy best-first search

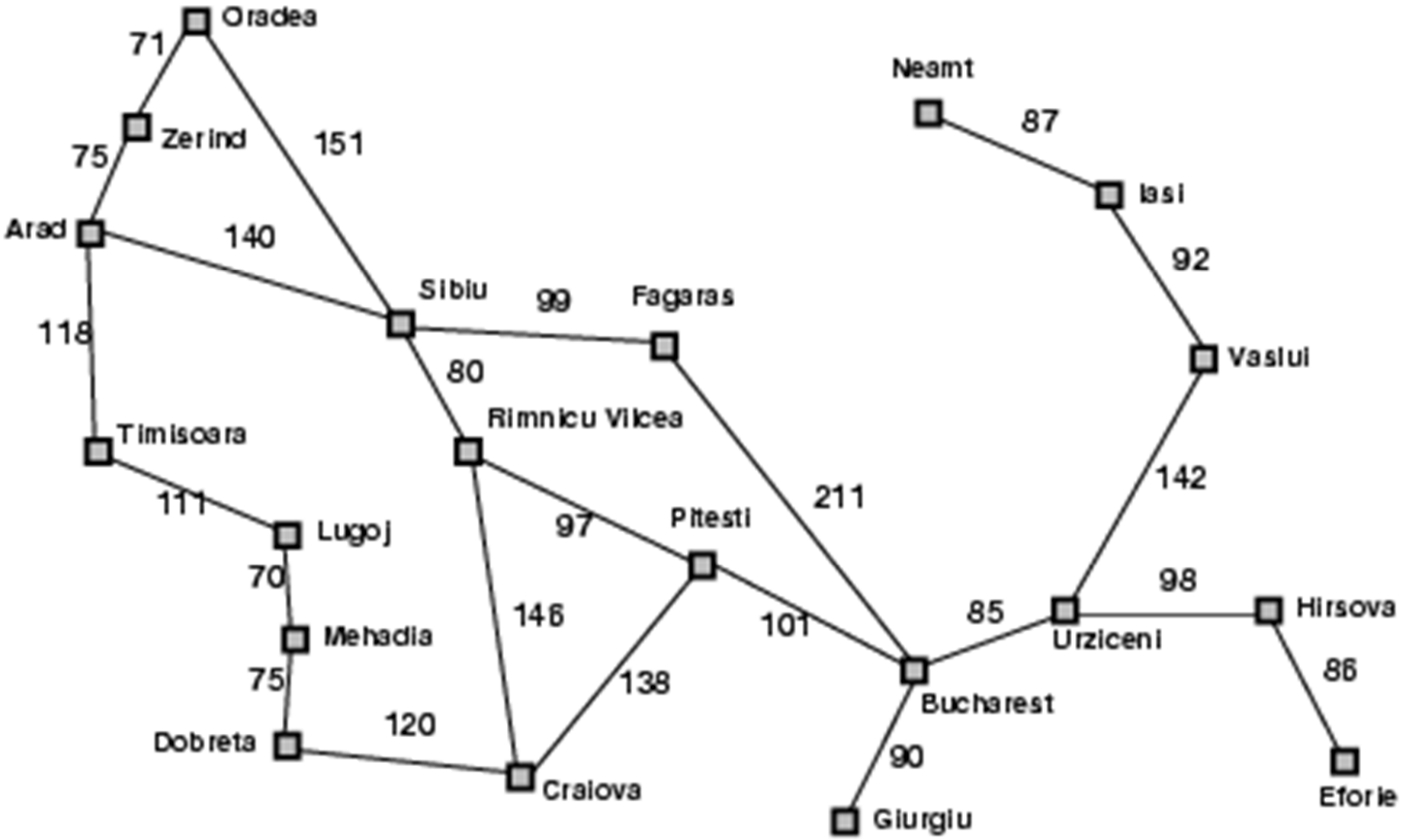
Evaluation function f(n) at node n

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

- Greedy best-first search expands the node that appears to have shortest path to goal
- Intuition: those nodes may lead fast to the solution

Similar to Depth-First Search:
It prefers to follow a single path to goal (guided by the heuristic), backing up when it hits a dead-end.

Example

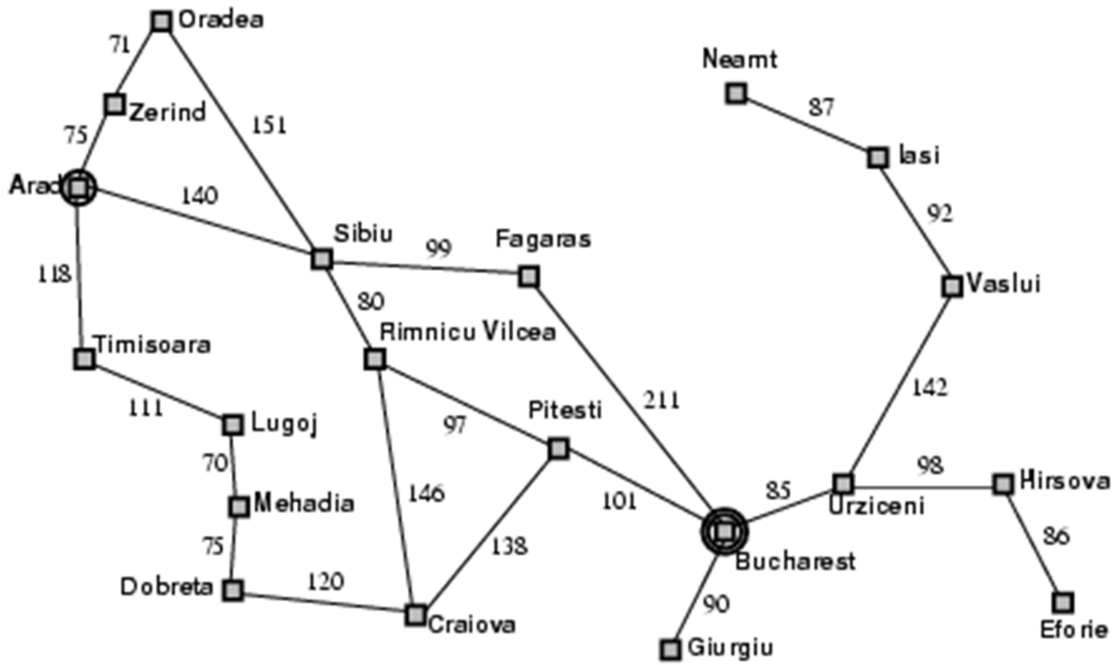


Straight–line distance
to Bucharest

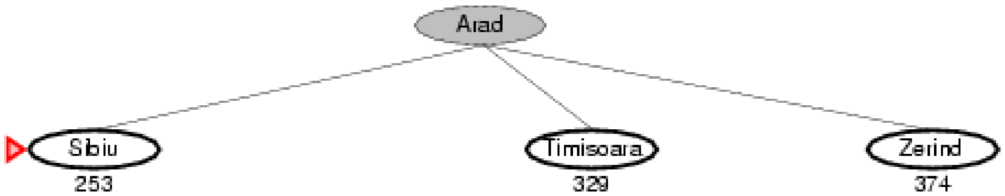
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

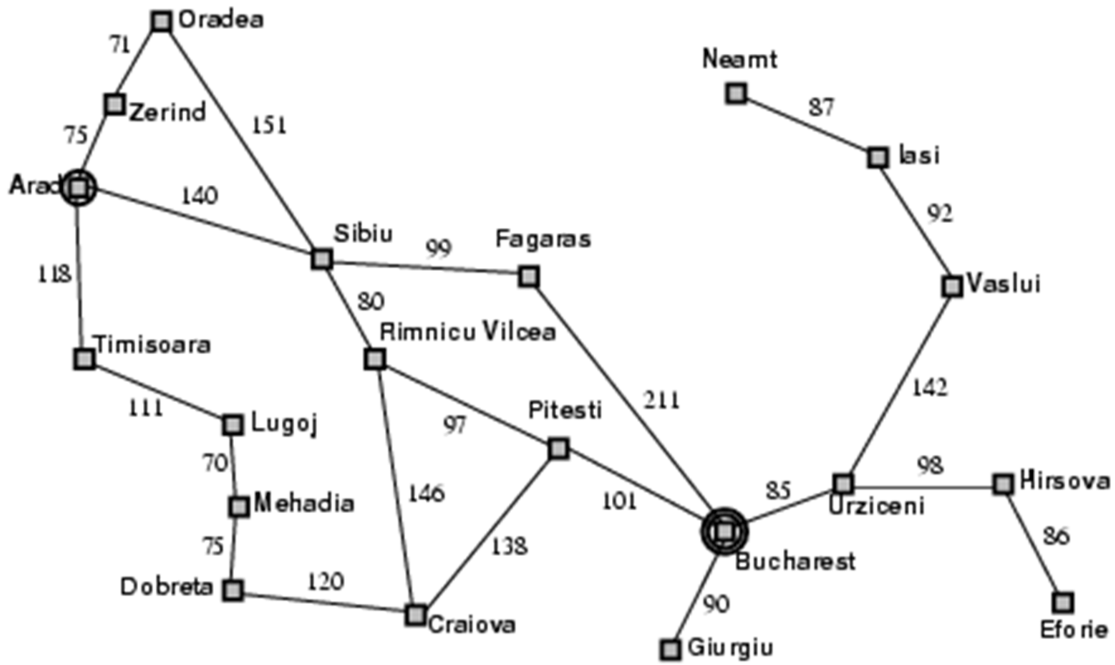
Example


Arad
366

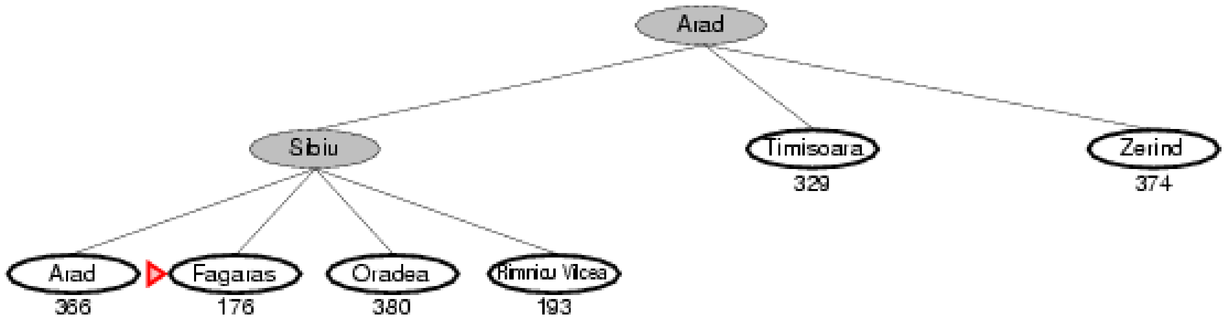

Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Dobreta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
98
Hirsova
85
Urziceni
86
Bucharest
90
Giurgiu
Eforie

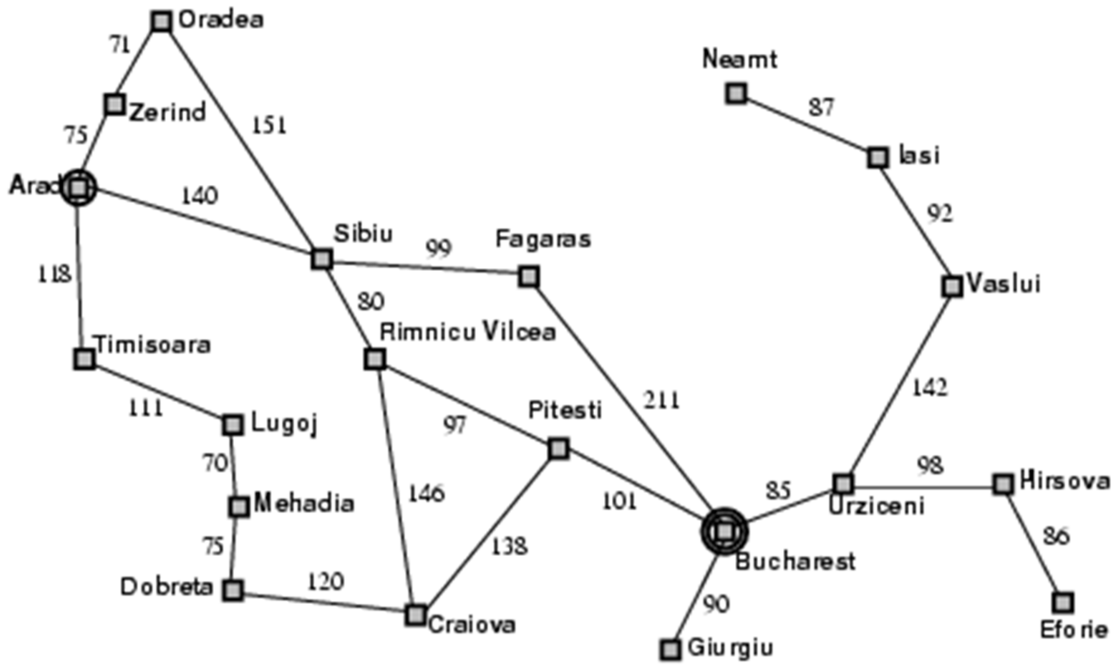
Example


Arad
Sibiu
253
Timisoara
329
Zerind
374

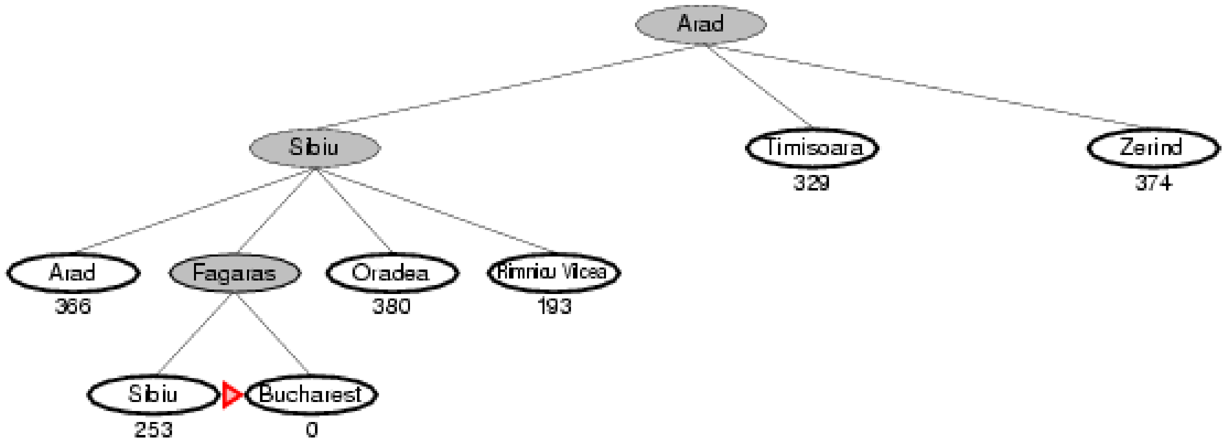

Oradea
71
Zerind
75
151
Neamt
87
Iasi
92
Arad
140
Sibiu
99
Fagaras
118
80
Vaslui
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
142
70
Mehadia
146
101
85
Urziceni
98
Hirsova
75
138
86
Bucharest
Dobreta
120
Craiova
90
Giurgiu
Eforie

Example

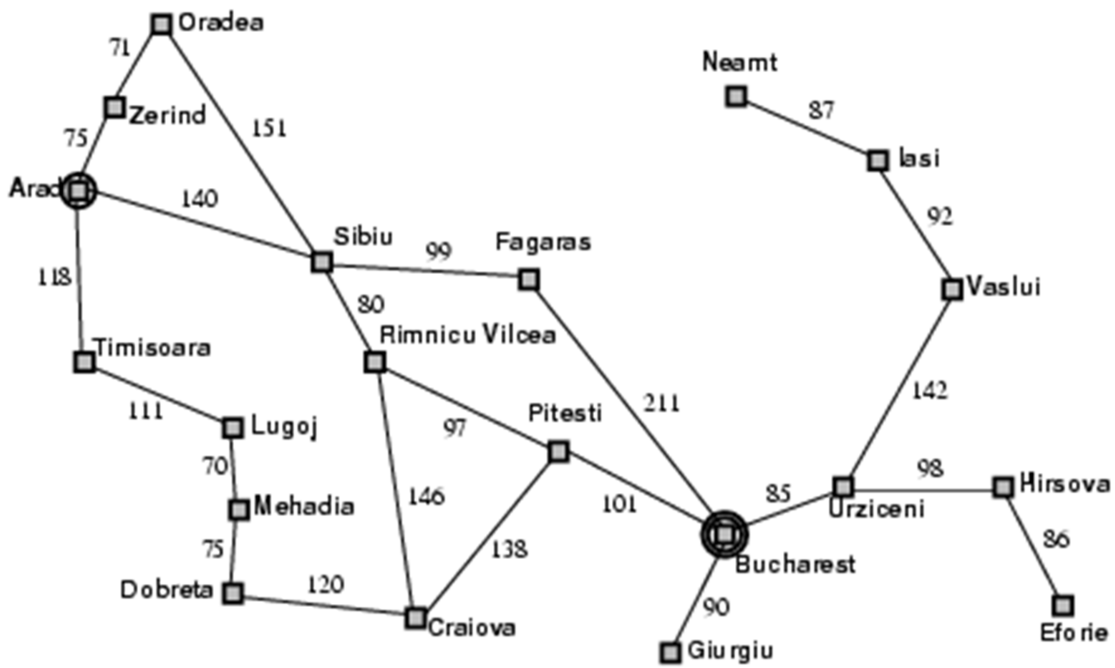

Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193


Oradea
71
Neamt
Zerind
87
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
Lugoj
97
Pitesti
211
70
98
Mehadia
146
85
Hirsova
101
Urziceni
75
138
86
Bucharest
Dobreta
120
90
Craiova
Eforie
Giurgiu

Example



Solution:
Arad-Sibiu-Fagaras-Bucharest
140+99+211 = 450 km



Is it optimal?

Properties of greedy best-first search

- ***not optimal***
- time: $O(b^m)$
- space: $O(b^m)$

b: maximum branching factor of the search tree
m: maximum depth of the state space (may be ∞)

A variation:
Single-Source Shortest Path Problem

problem of finding

- shortest paths
- from a source vertex v
- to ***all other*** vertices in the graph

Dijkstra's algorithm

- solution to the single-source shortest path problem
- both for directed and undirected ***weighted*** graphs
- but all edges must have *non-negative weights*
- using a greedy approach

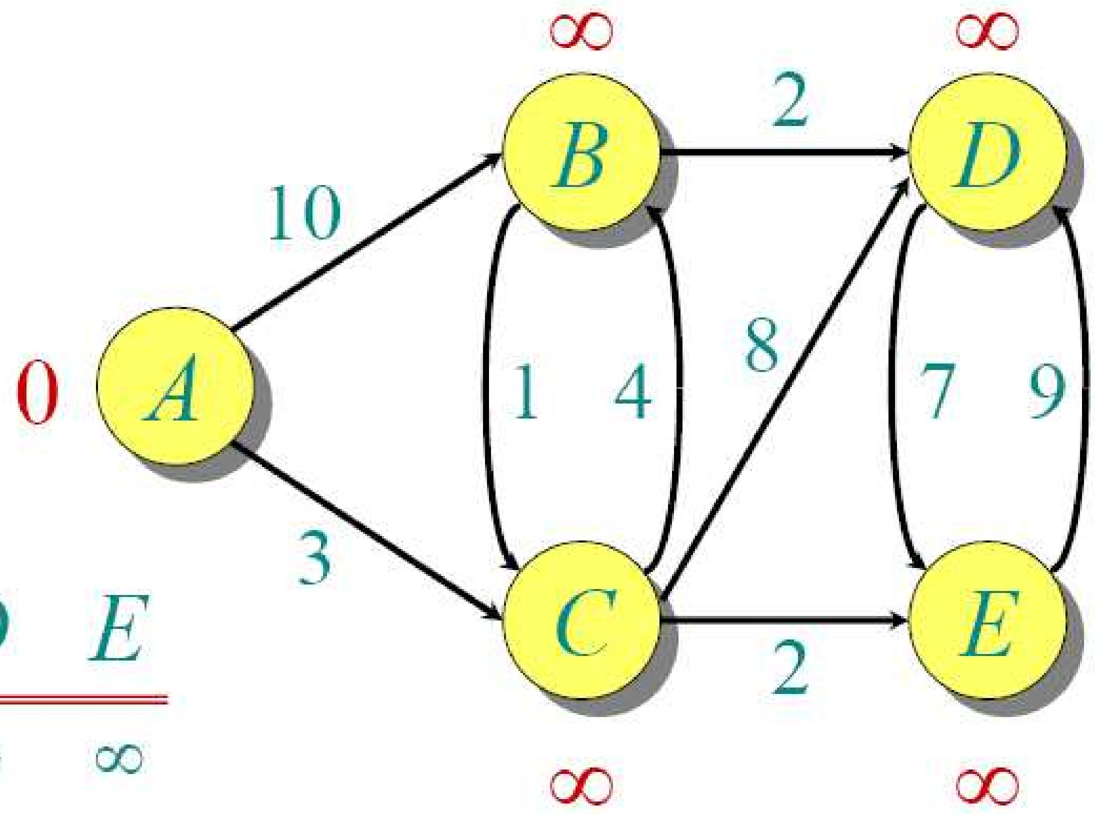
Input: weighted graph G={E,V} and source vertex v∈V,
all edge weights are nonnegative

Output: lengths of shortest paths (or the shortest paths themselves) from a given source vertex v∈V to all other vertices

Dijkstra Pseudocode

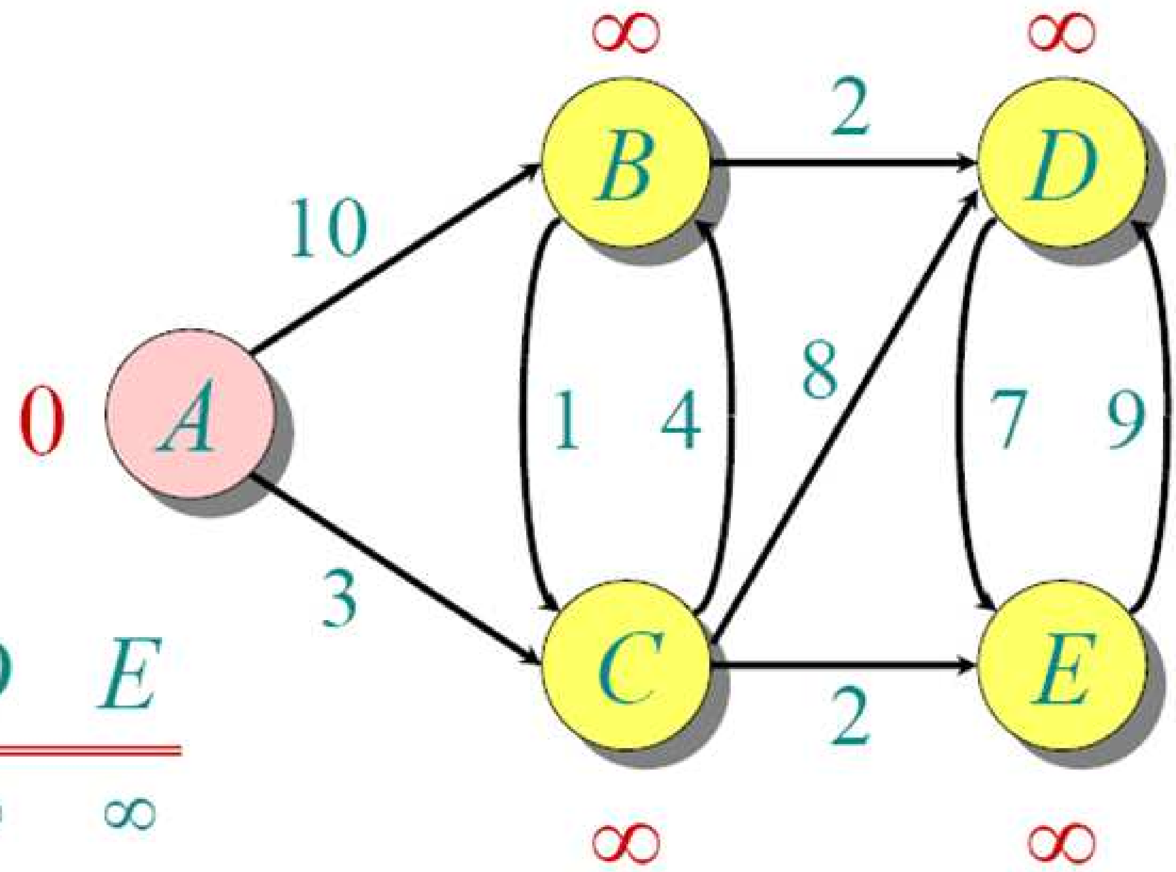
dist[s] ←0	(distance to source vertex is zero)
for all v ∈ V–{s}	
do dist[v] ←∞	(set all other distances to infinity)
S←∅	(S, the set of visited vertices is initially empty)
Q←V	(Q, the queue initially contains all vertices)
while Q ≠∅	(while the queue is not empty)
do u ← mindistance(Q,dist)	(select the element of Q with the min. distance)
S←S∪{u}	(add u to list of visited vertices)
for all v ∈ neighbors[u]	
do if dist[v] > dist[u] + w(u, v)	(if new shortest path found)
then d[v] ←d[u] + w(u, v)	(set new value of shortest path)
return dist	

Initialize:

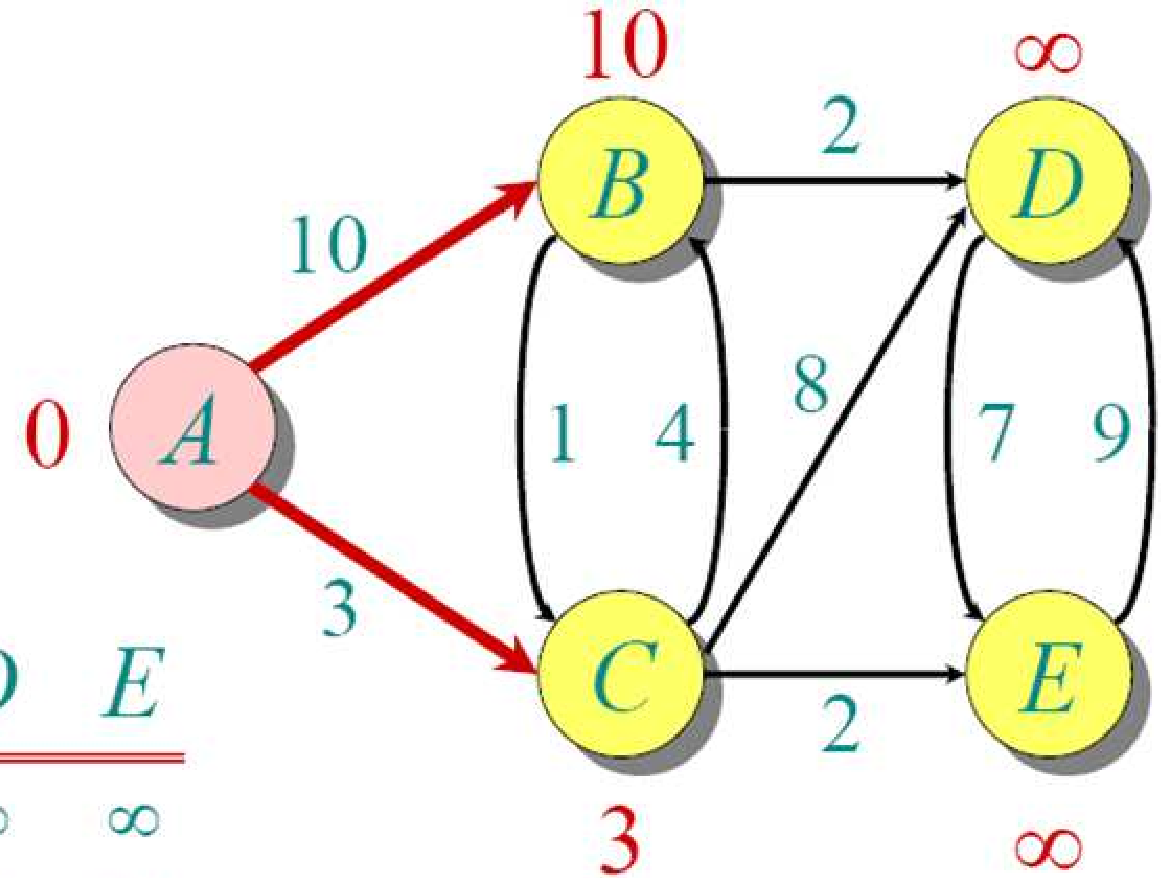


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

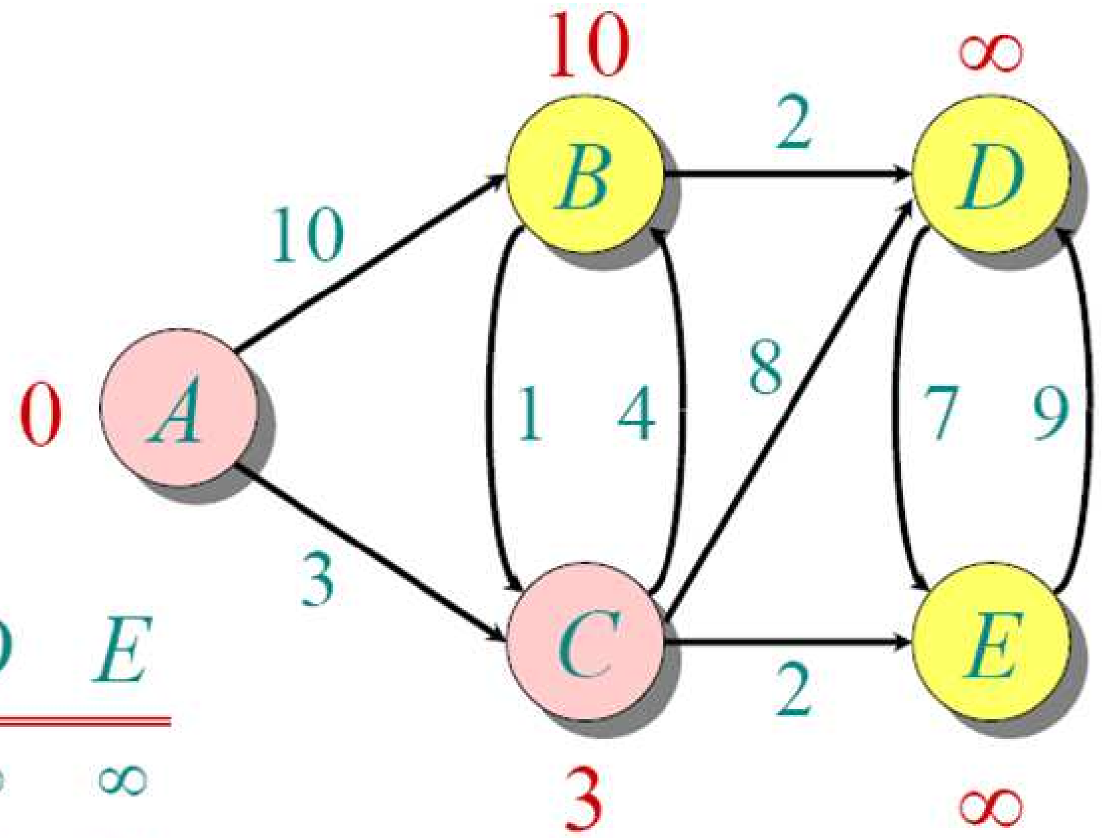


Q:	A	B	C	D	E
	0	∞	∞	∞	∞



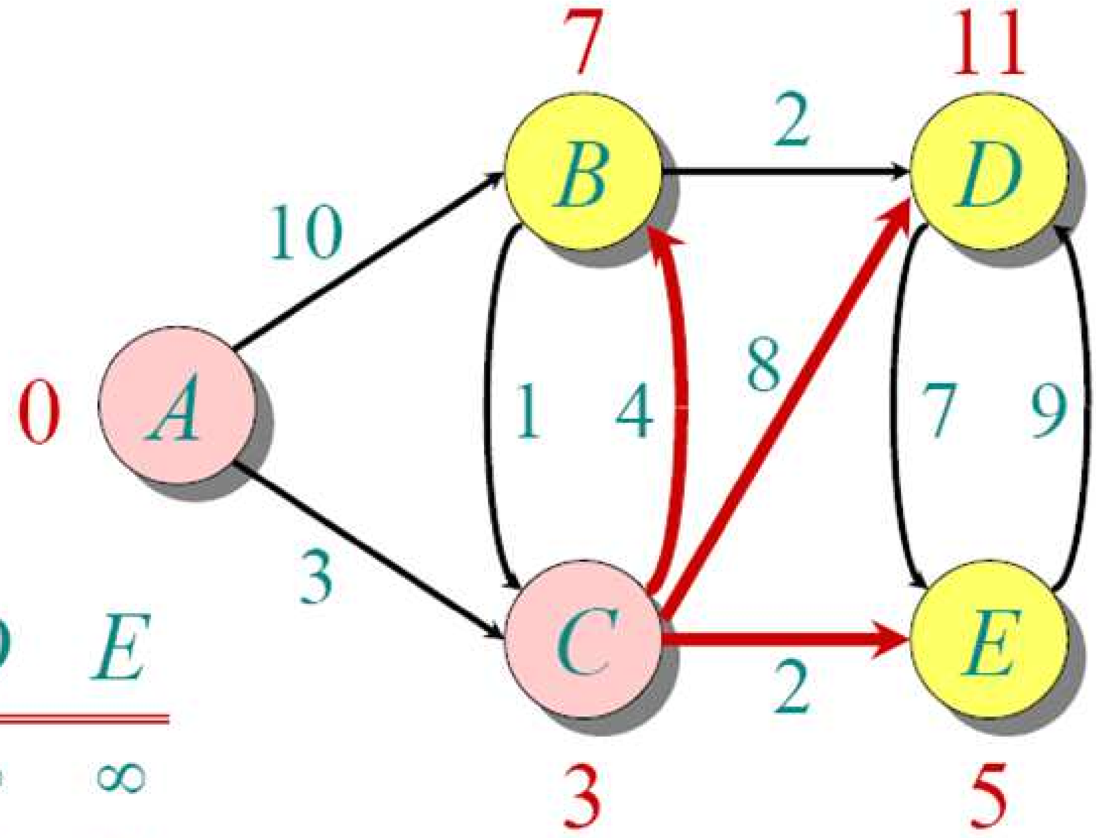
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }



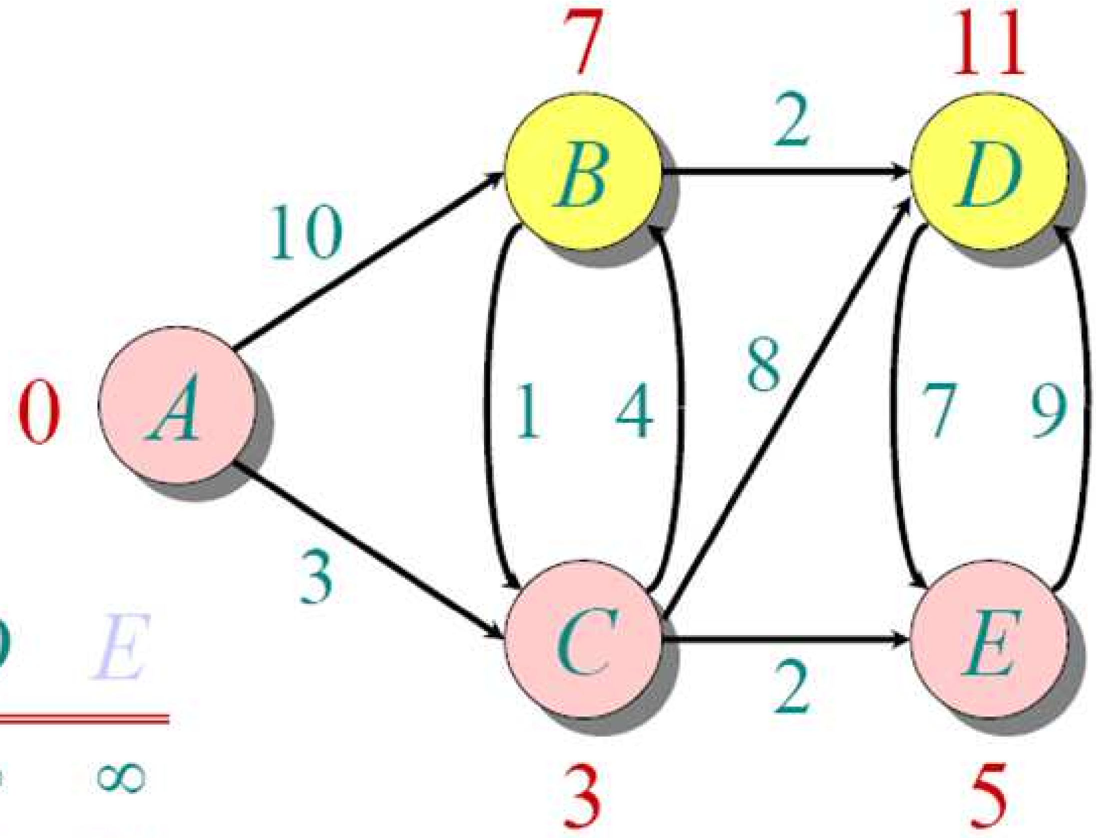
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }



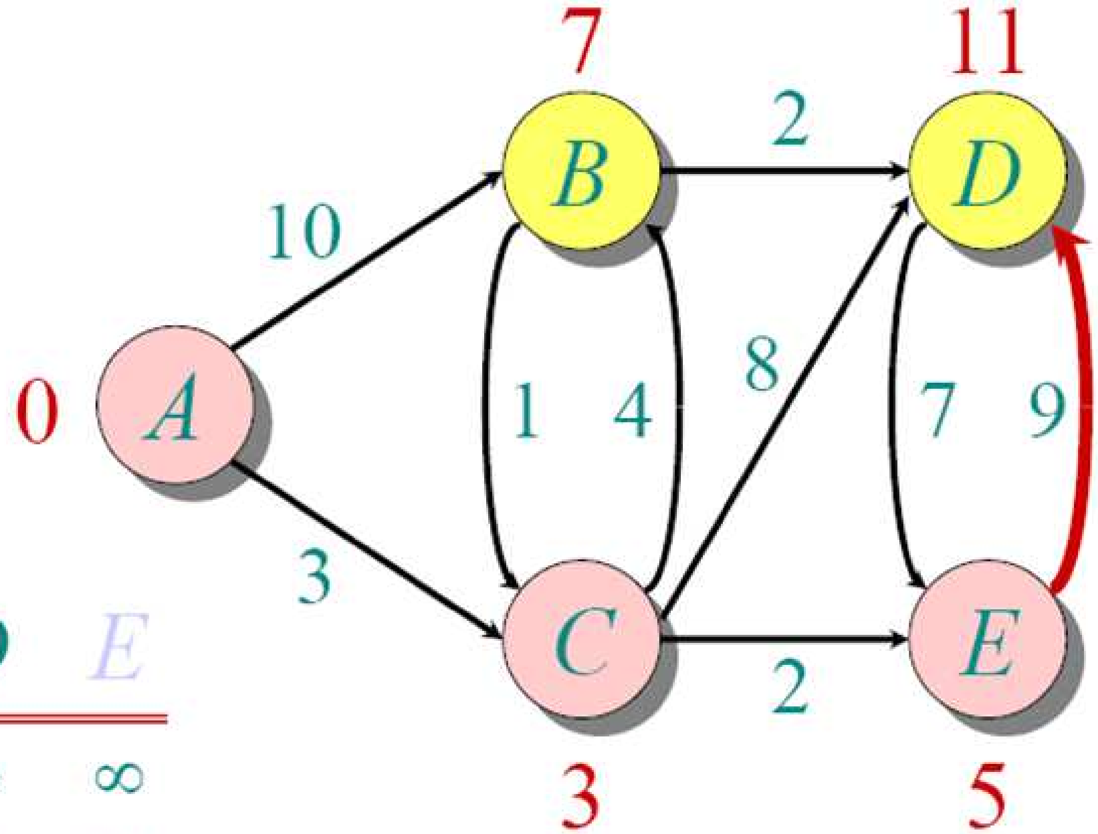
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }



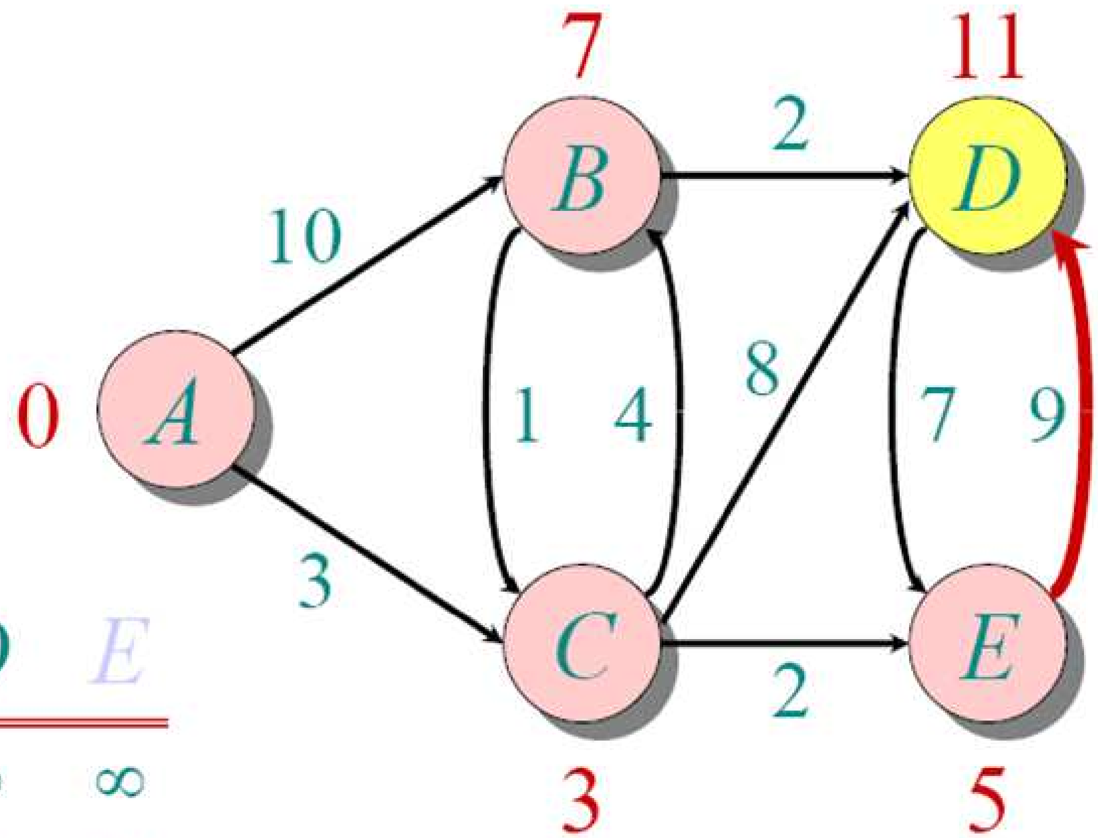
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

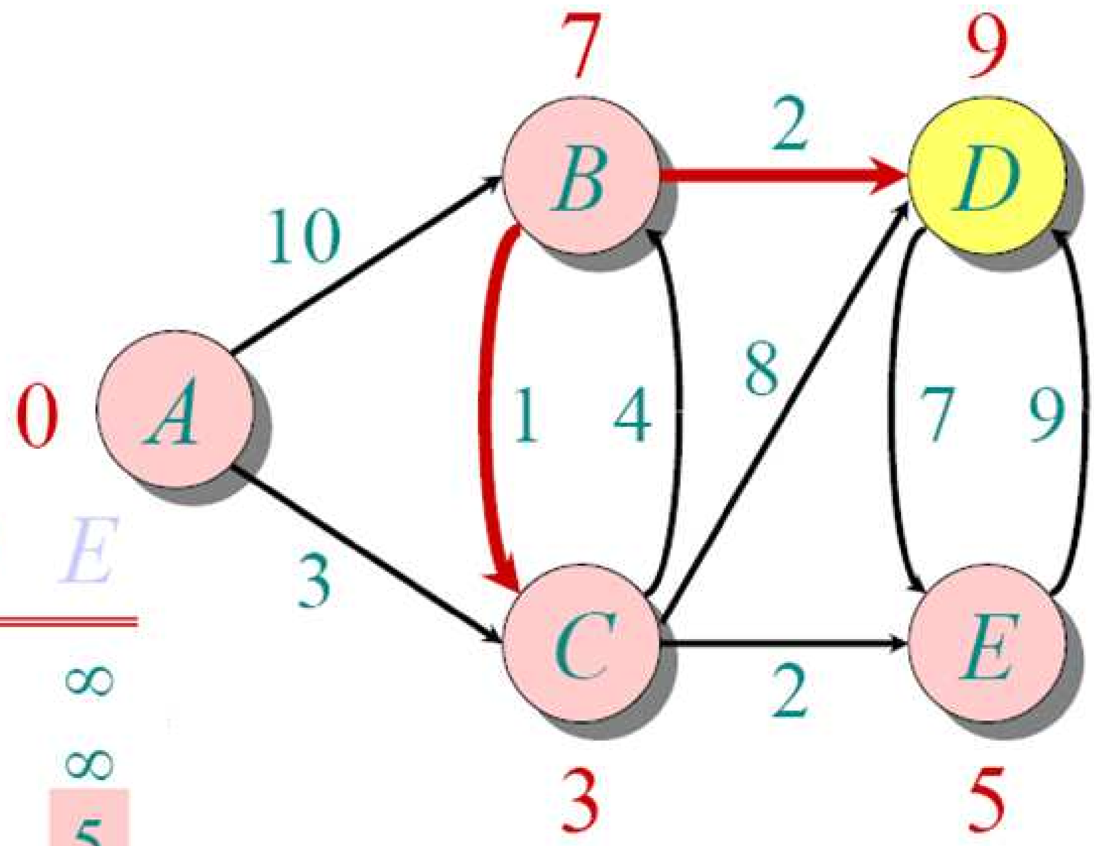
S: { A, C, E }



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

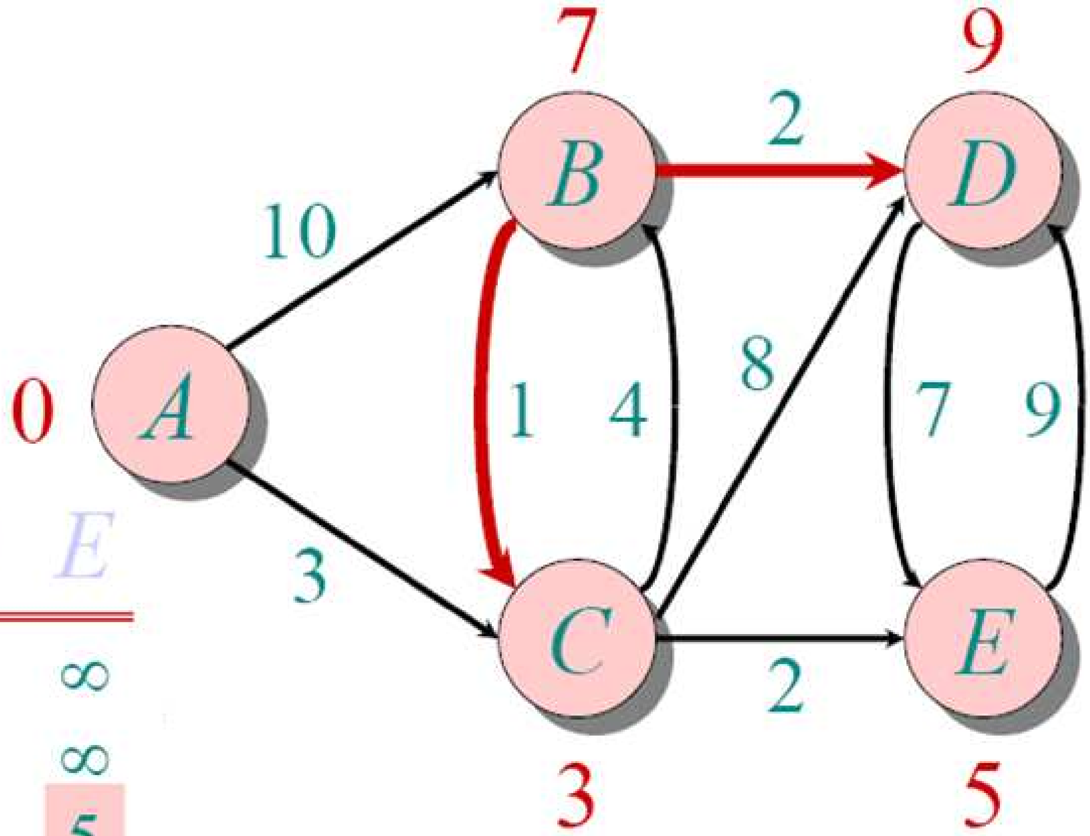
S: { A, C, E, B }

Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				9	



S: { A, C, E, B }

Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	



S: { A, C, E, B, D }

Implementations and Runtimes

- simplest
  - vertices in an array or linked list
  - runtime $O(|V|^2 + |E|)$
- sparse graphs: #edges << $\#nodes^2$
  - store graph in an adjacency list using a priority queue with binary heap
  - runtime $O((|E|+|V|) \log |V|)$
  - note: many „problem solving“ searches are of this type, hence interesting for performance

Excursus: Priority Queue

PQueue: data with priority, resp. weights

- operations
  - insert
  - extractMin
- property
  - for two elements in the queue, x and y,
  - if x has a lower priority value than y,
  - x will be extracted before y

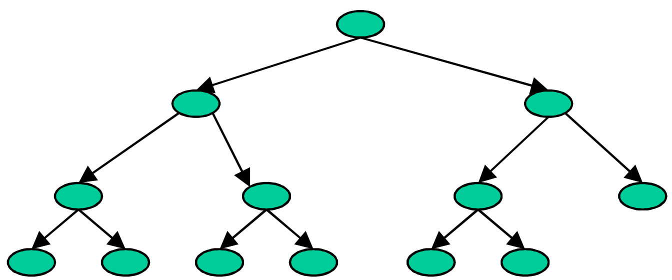
Potential Implementations

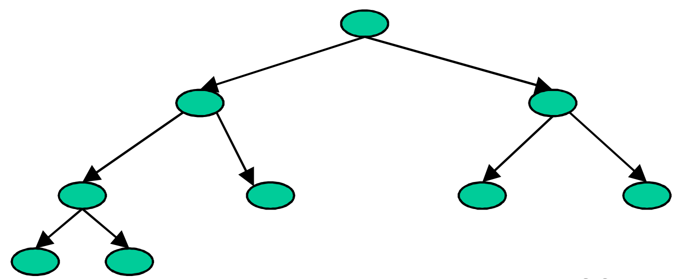
	insert	extractMin
Unsorted list (Array, Linked-List)	O(1)	O(n)
Sorted list (Array, Linked-List)	O(n)	O(1)

Better Alternative: Binary Heap

- Heap
  - O(log n) worst case for insert and extractMin
  - O(1) average insert

- binary heap: complete binary tree
  - binary tree, i.e., b=2
  - completely filled; exception bottom level, which is filled left to right
- examples

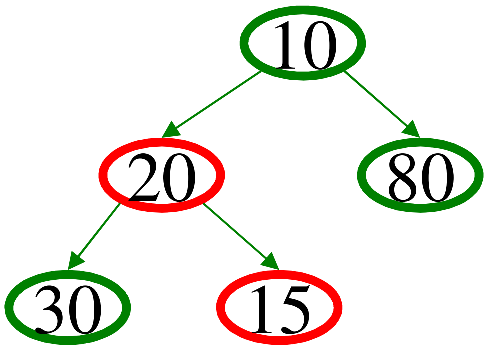




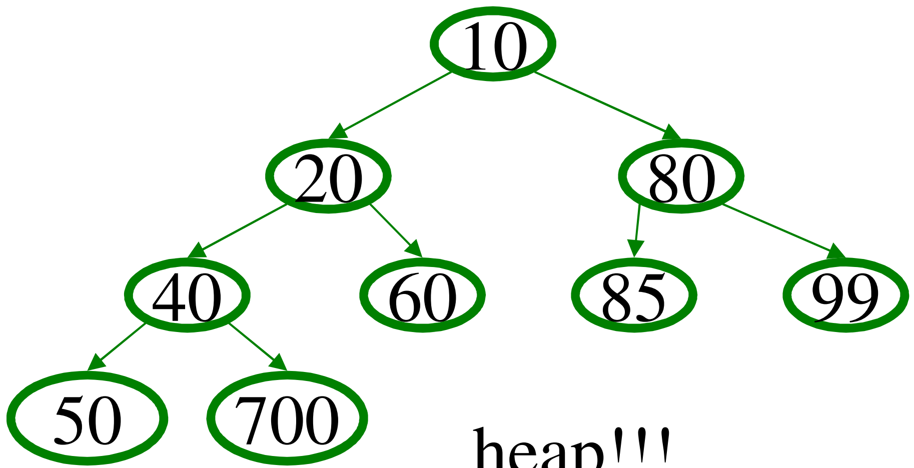
Binary Heap

Heap order property

- For every non-root node X
- the value in the parent of X
- is less than (or equal to) the value in X



not a heap

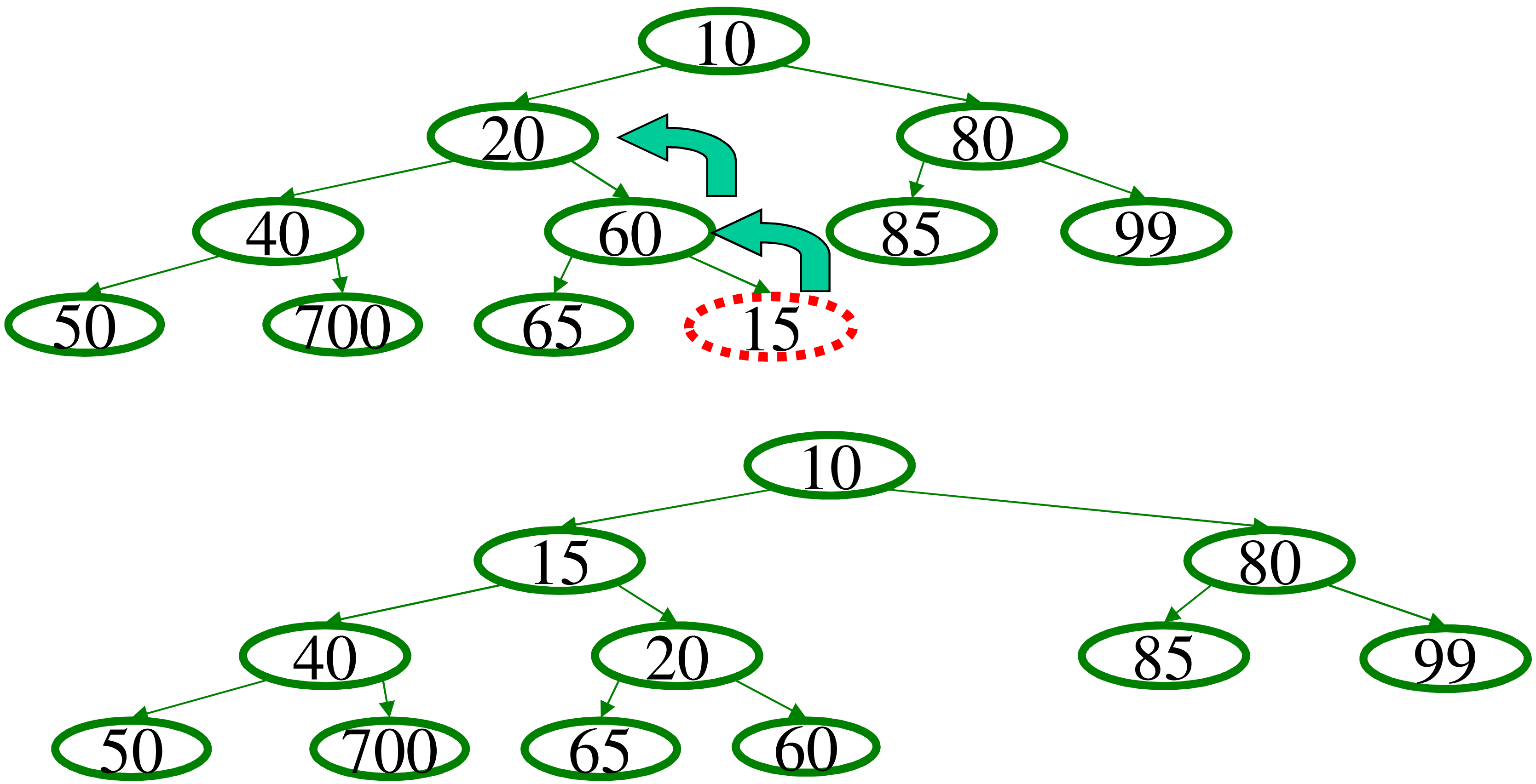


heap!!!

Insert(val)

Basic Idea:

- put val at next open leaf position
- percolate up
- i.e., repeatedly exchanging node until no longer needed

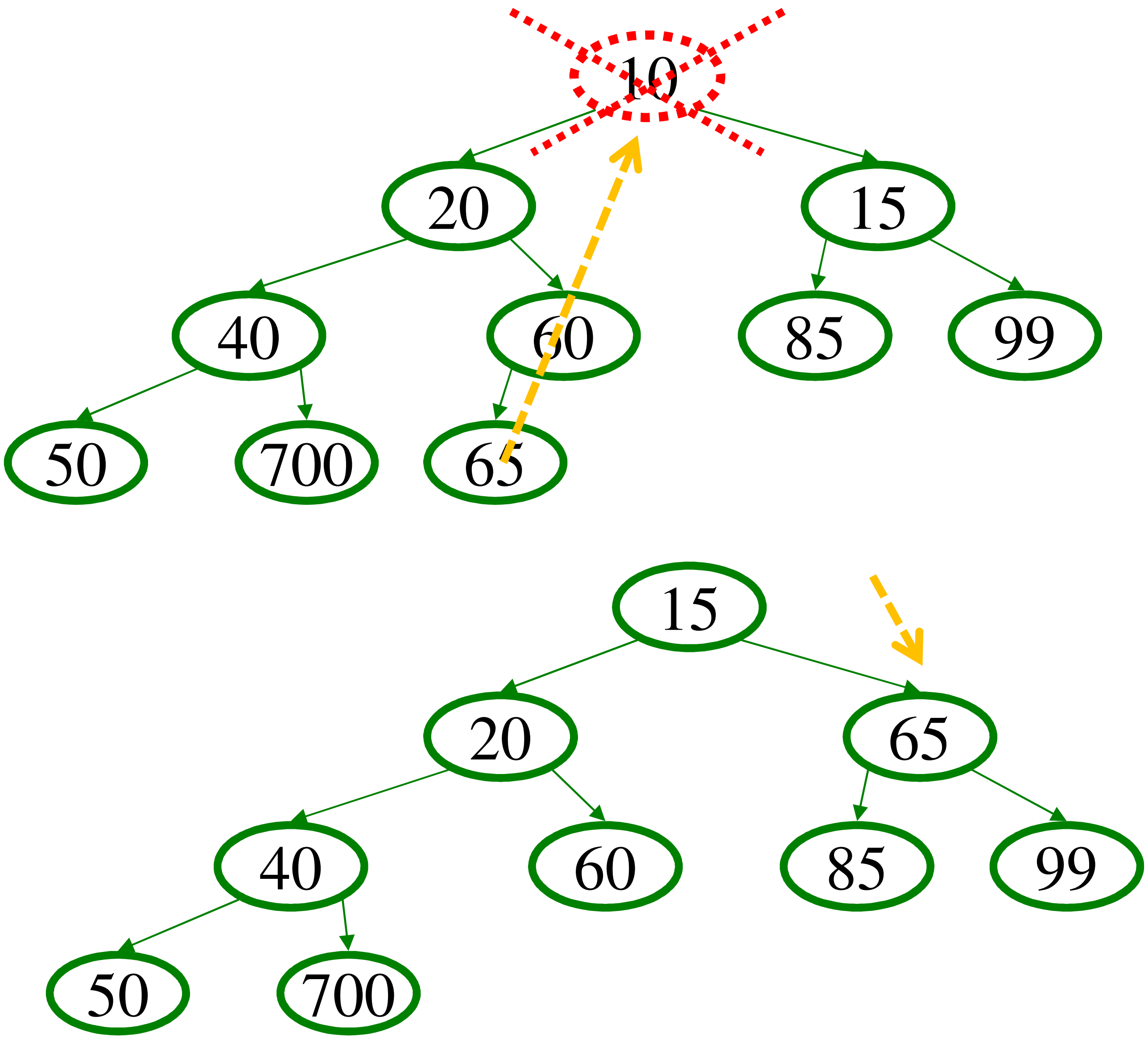


extractMin

Basic Idea:

1. remove root (that is always the min!)
2. put “last” leaf node at root
3. find smallest child of node
4. swap node with its smallest child if needed.
5. repeat steps 3 & 4 until no swaps needed (aka percolate down)

extractMin: percolate down



Back to Dijkstra's Algorithm

Correctness, i.e., complete? optimal?

Yes

Here the core elements of the proof...

Correctness Dijkstra

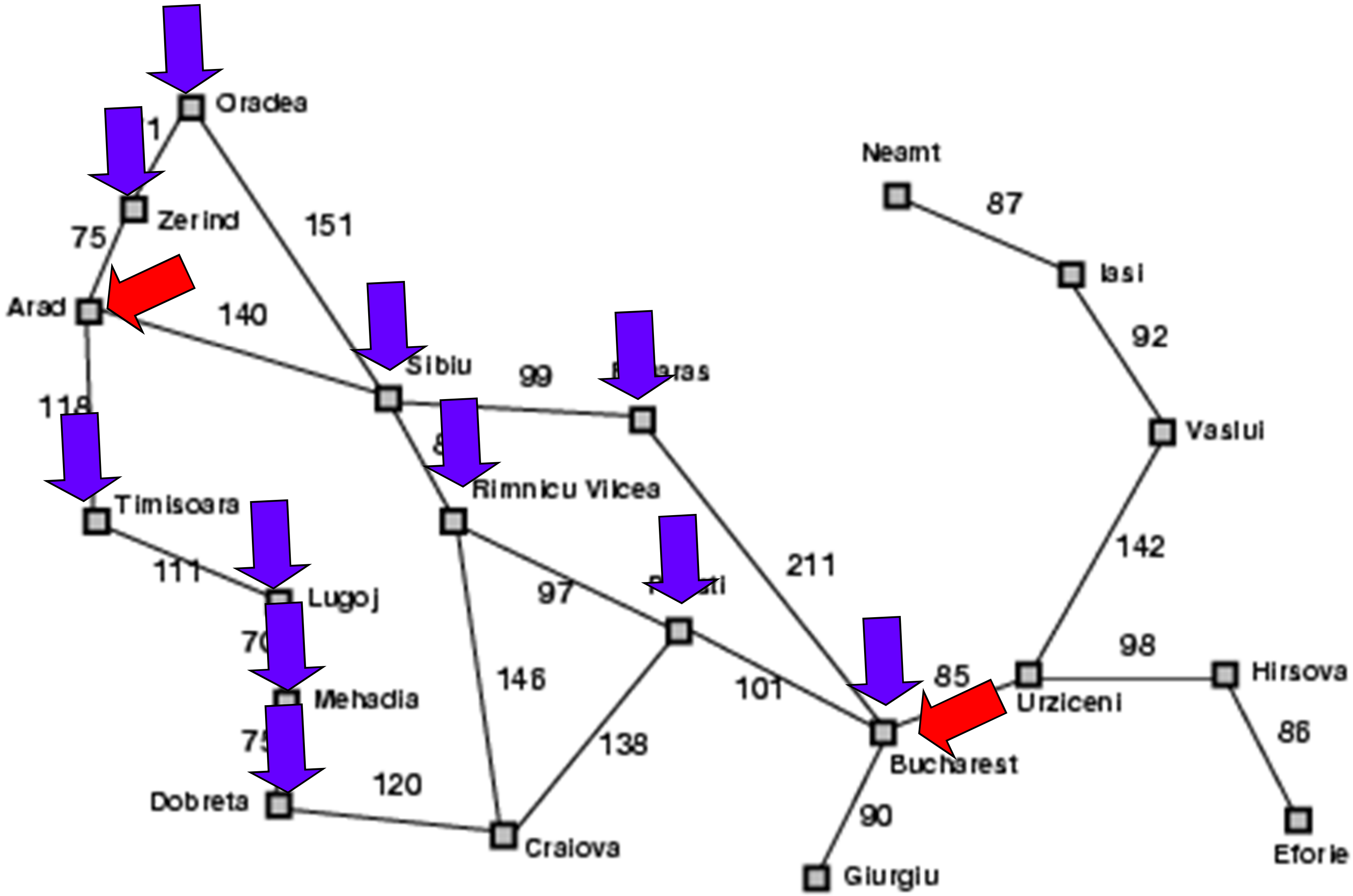
Lemma 1: Triangle inequality
If $\delta(u,v)$ is the shortest path length between u and v,
$\delta(u,v) \leq \delta(u,x) + \delta(x,v)$

Lemma 2: Subpaths
The subpath of any shortest path is itself a shortest path

key insight:

- anytime we put a new vertex in S,
- we already know the shortest path to it

Dijkstra nice but...



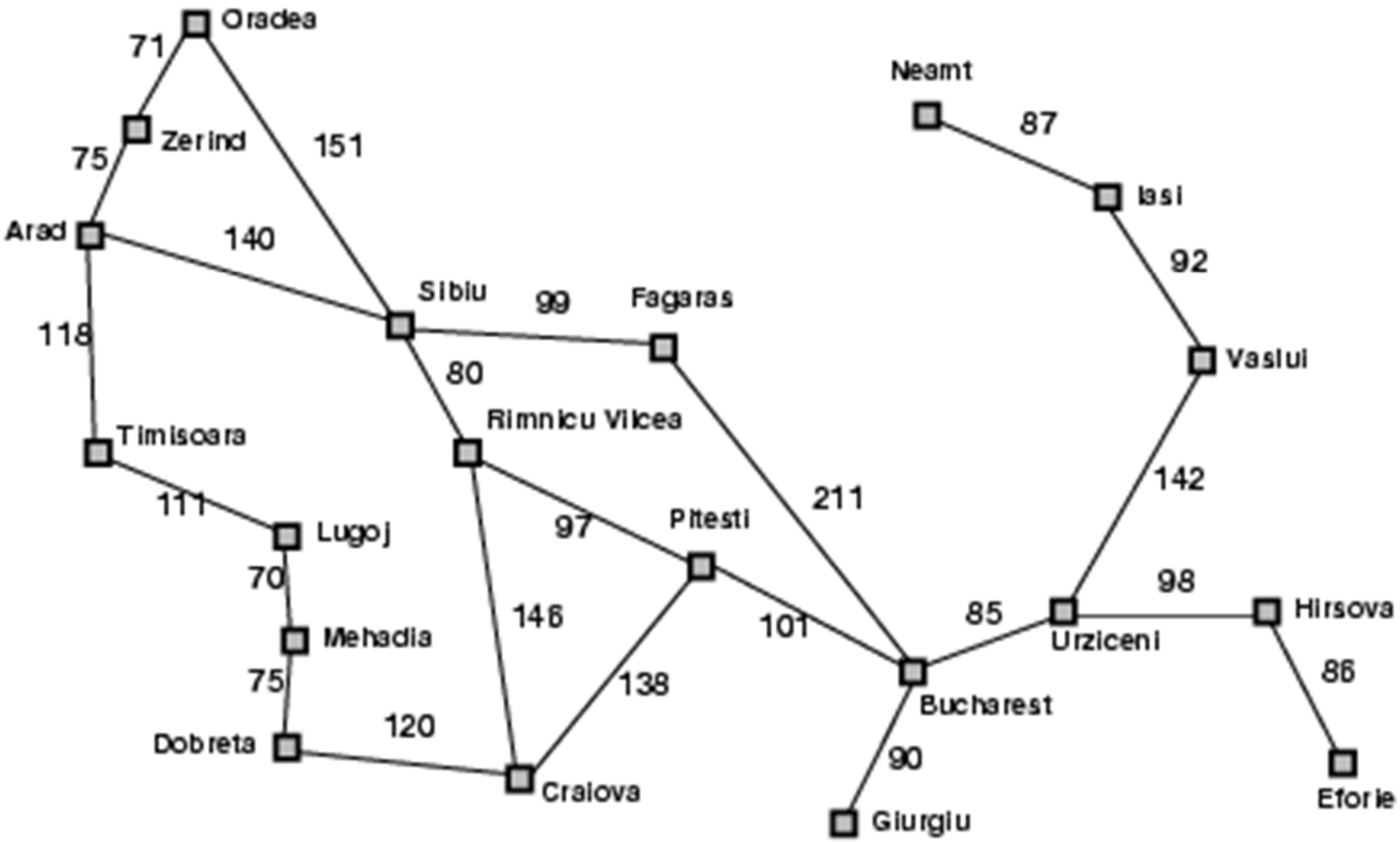
Problem: what about single goal?

- can be used
- but is not the most efficient

Alternative: A* search

- idea
  - follow most promising path (like greedy search)
  - but do not expand paths that are already expensive
- evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated cost from n to goal
  - $f(n)$ = estimated total cost of path through n to goal

Example

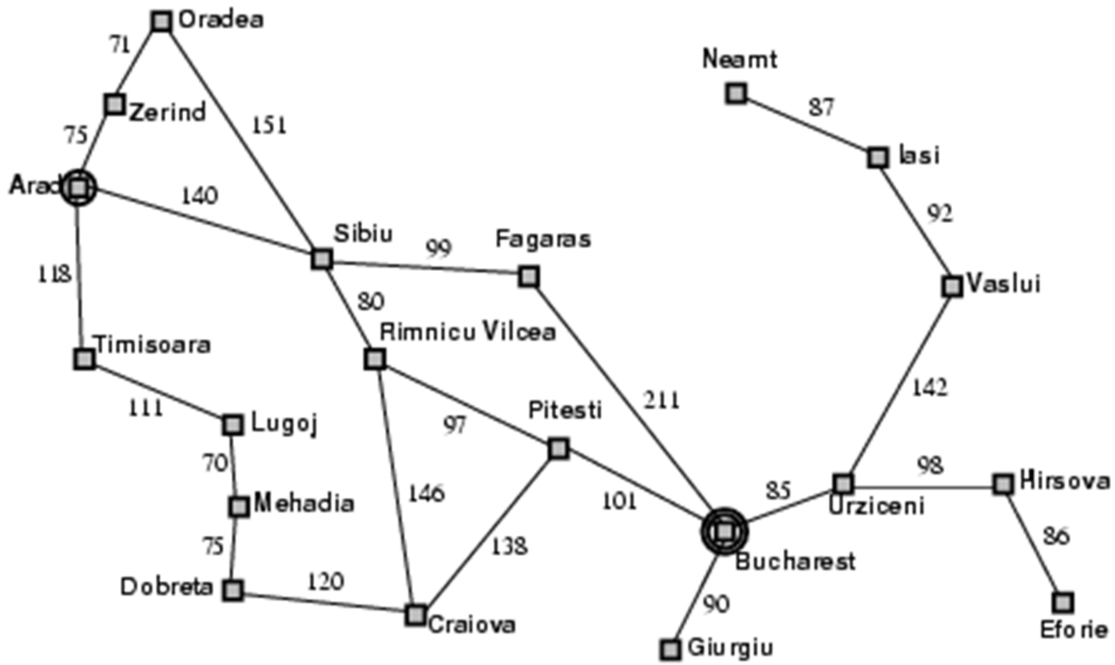


Straight–line distance to Bucharest

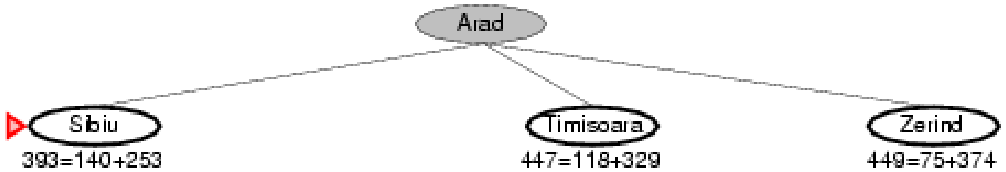
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

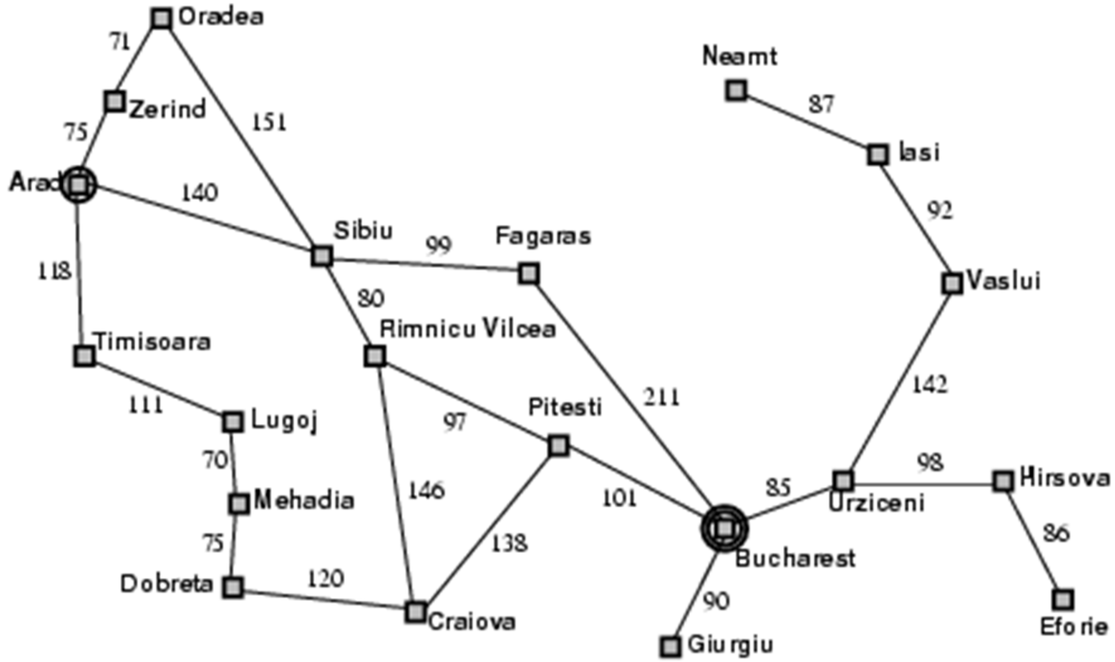
h(n) = straight line distance to Bucharest

A* search example

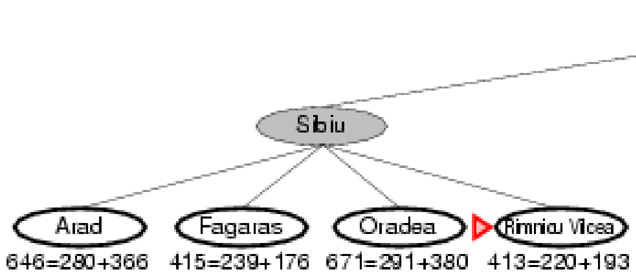


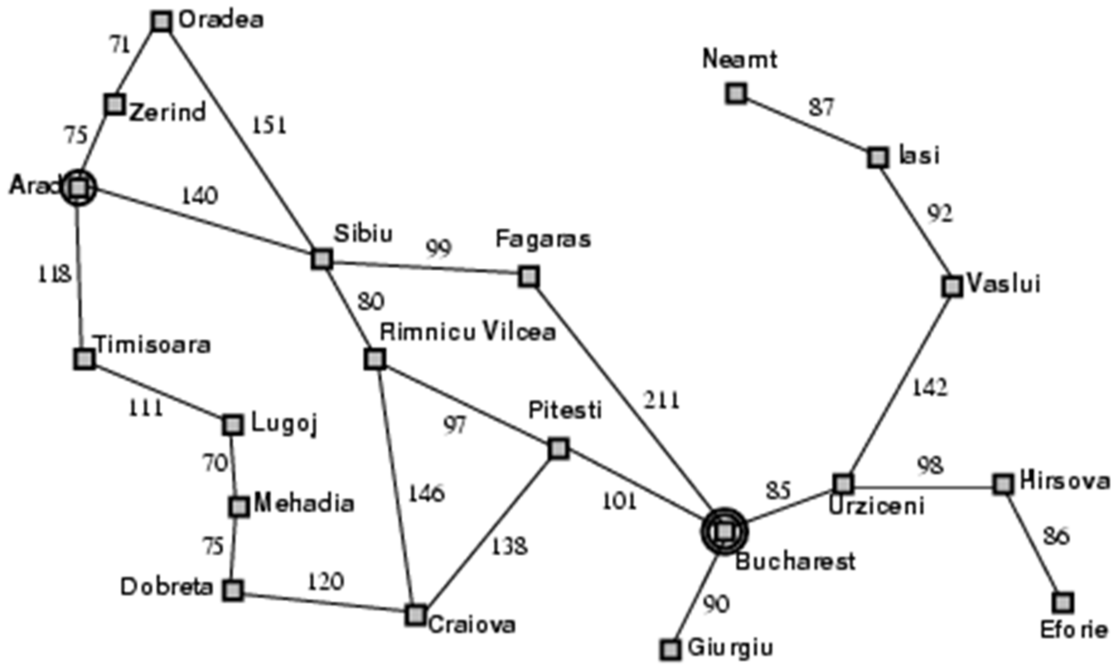
A* search example



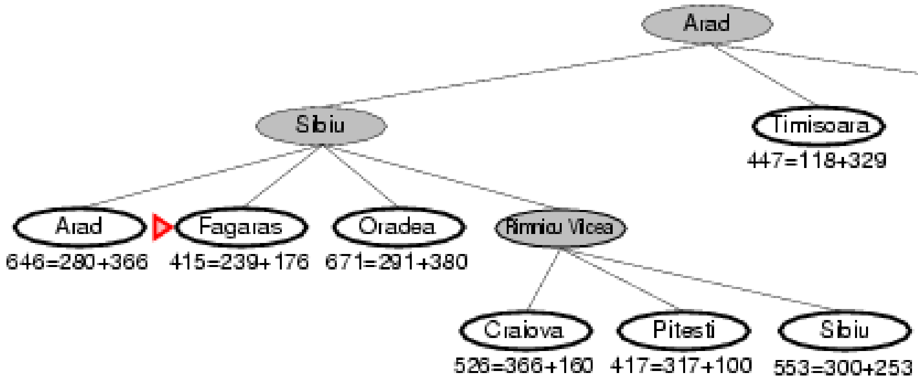


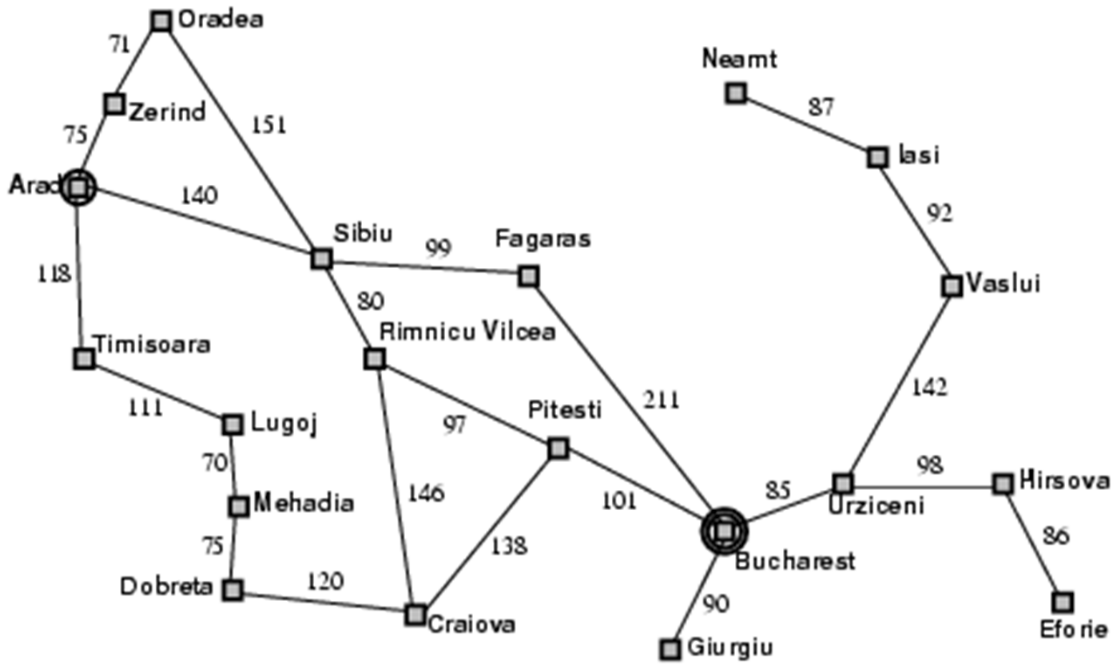
A* search example



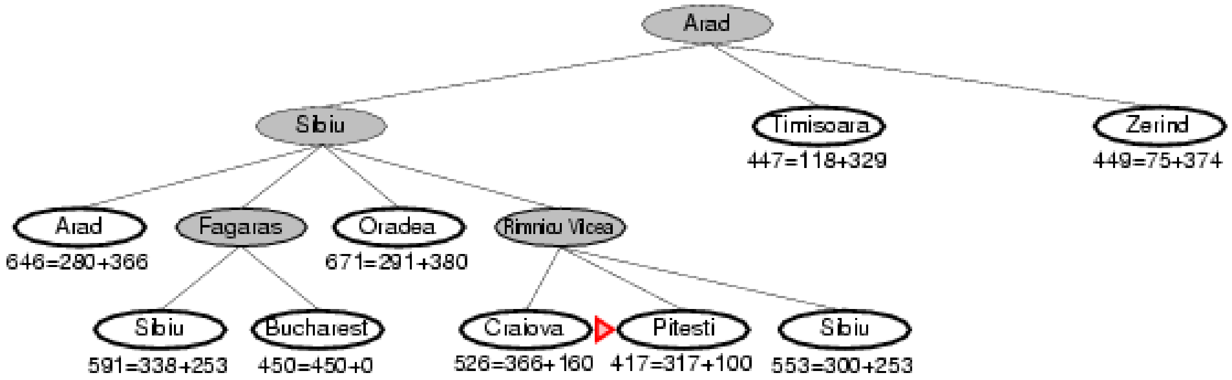


A* search example

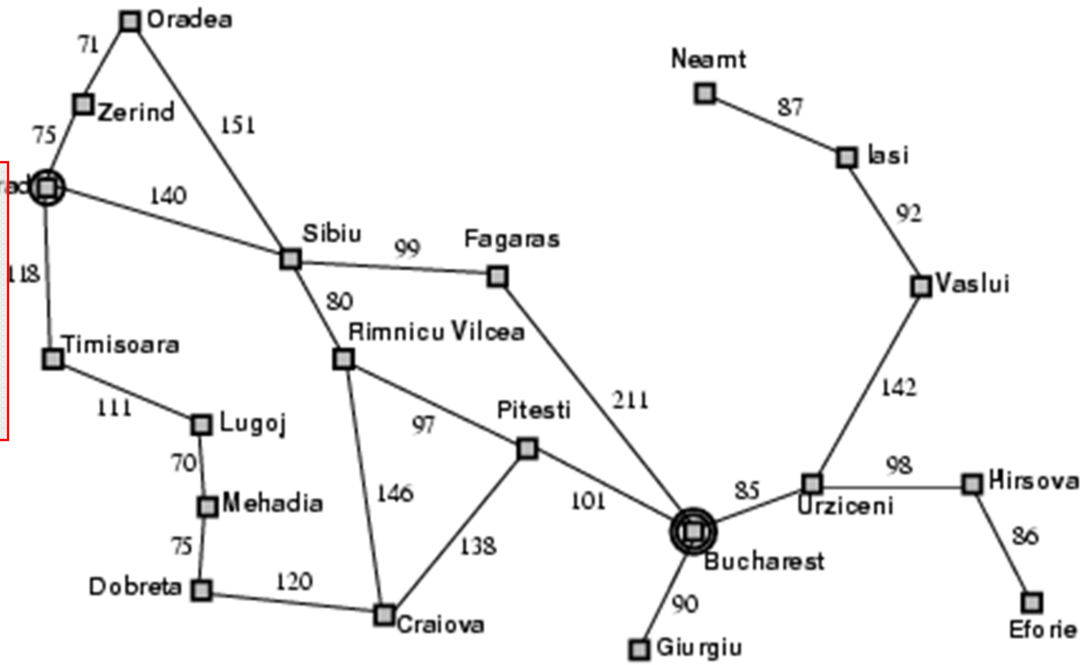




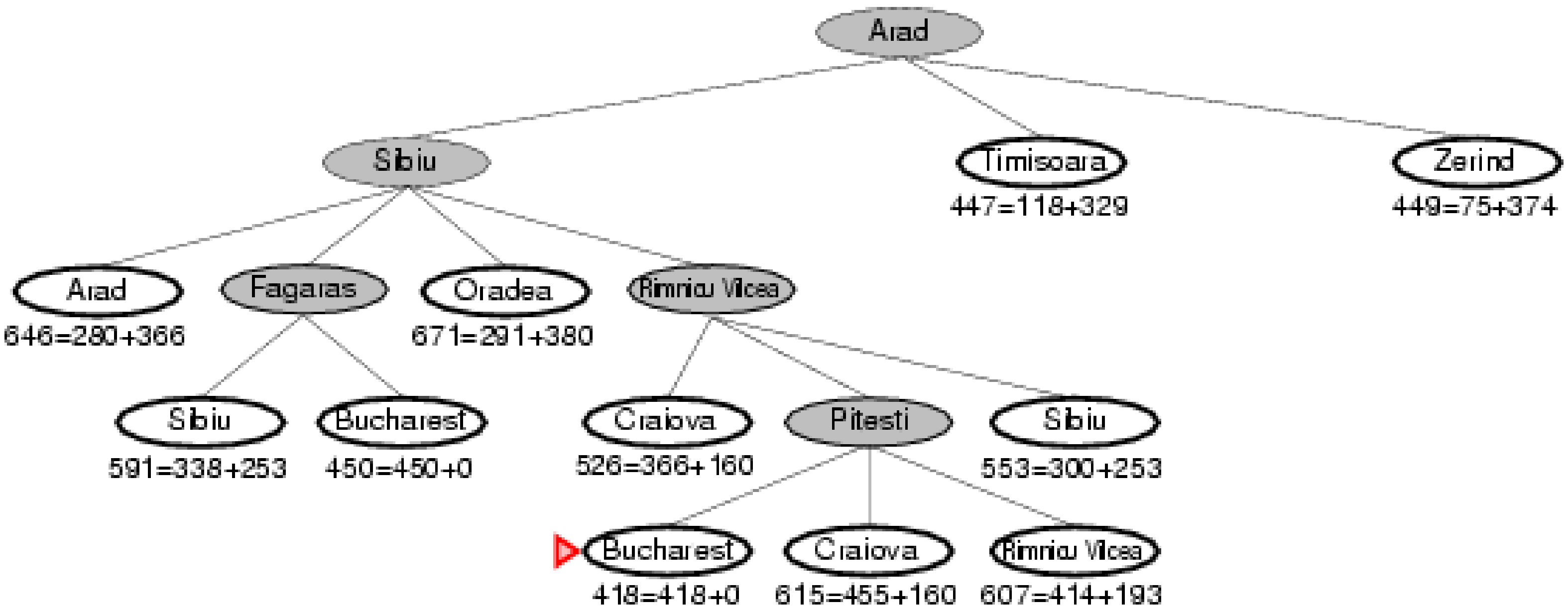
A* search example



Bucharest appears on the fringe but not selected for expansion since its cost (450) is higher than that of Pitesti (417)



A* search example



Arad-Sibiu-Rimnicu-Pitesti-Bucharest

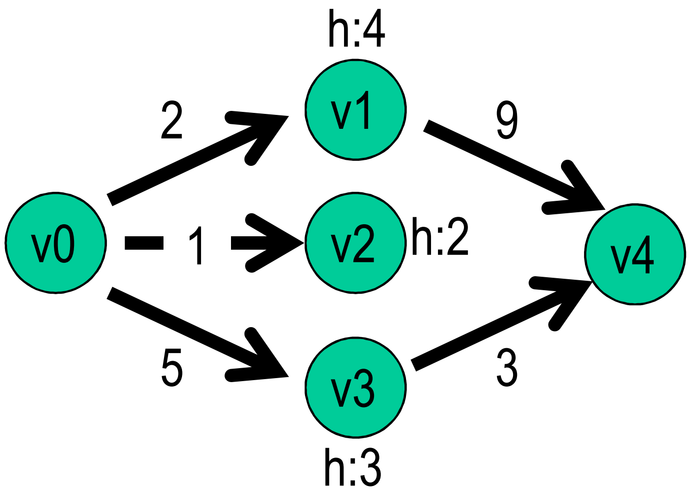
Claim: Optimal path found!

Pseudo-Code A*

```
forall v : g(v) = infinity; g(v_start) = 0              //init g()
forall v : f(v) = infinity; f(v_start) = h(v_start)     //init f()
add(v_start, OPEN)                                      //put start in priority queue OPEN
while OPEN != {}                                        //as long as OPEN is not empty
  v_visit = min-f (OPEN)                                //extract the node with smallest f() from OPEN
  if v_visit == v_goal then done                        //found the solution
    add(v_visit, CLOSED)                                //mark visited node as CLOSED
    generate set S of all successors of v_visit         //expand the fringe
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) + w(v_visit, v_next)   // add edge weight to get new g()
        if tmp_g(v_next) < g(v_next)                    //found a better path to v_next
          path-parent = v_visit                   //the currently best path to v_next is coming from v_visit
          g(v_next) = tmp_g(v_next)                     //update g()
          f(v_next) = g(v_next) + h(v_next)             //update f()
          if v_next not in OPEN then add(v_next, OPEN)    //add v_next to the fringe
return fail                                             //no path from start to goal
```

Pseudo-Code A*

find path from v0 to v4

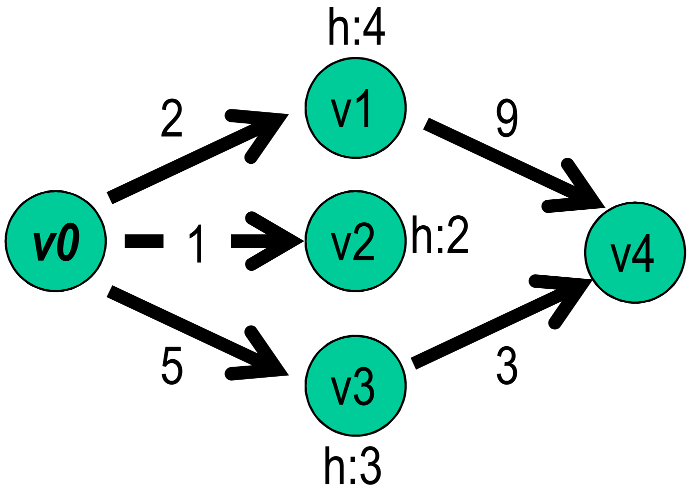


	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	inf	inf	NIL
v2	2	inf	inf	NIL
v3	3	inf	inf	NIL
v4	0	inf	inf	NIL

OPEN	v0			
CLOSED				

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
  v_visit = min-f (OPEN)
  if v_visit == v_goal then done
    add(v_visit, CLOSED)
    generate set S
          of all successors of v_visit
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) +
                    w(v_visit, v_next)
        if tmp_g(v_next) < g(v_next)
          path-parent = v_visit
          g(v_next) = tmp_g(v_next)
          f(v_next) = g(v_next) + h(v_next)
          if v_next not in OPEN then
                    add(v_next, OPEN)
return fail
```

Pseudo-Code A*

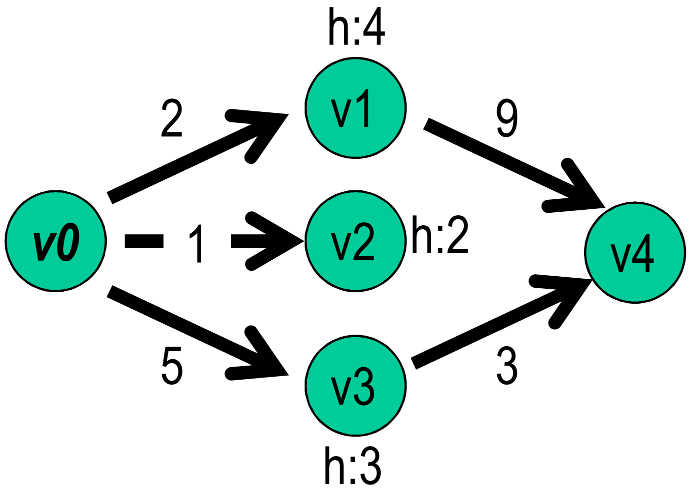


	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	6	v0
v2	2	1	3	v0
v3	3	5	8	v0
v4	0	inf	inf	NIL

OPEN	v0		
v_visit	v0		
CLOSED	v0		
S	v1	v2	v3
tmp_g	2	1	5

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
  v_visit = min-f (OPEN)   v0
  if v_visit == v_goal then done
    add(v_visit, CLOSED)  {v0}
    generate set S               {v1,v2,v3}
          of all successors of v_visit
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) +
                        w(v_visit, v_next)
        if tmp_g(v_next) < g(v_next)  =inf
          path-parent = v_visit
          g(v_next) = tmp_g(v_next)
          f(v_next) = g(v_next) + h(v_next)
          if v_next not in OPEN then
                        add(v_next, OPEN)
return fail
```

Pseudo-Code A*

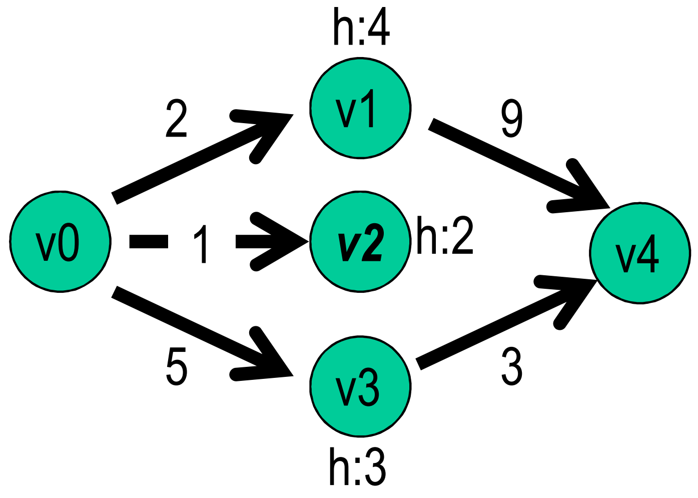


	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	6	v0
v2	2	1	3	v0
v3	3	5	8	v0
v4	0	inf	inf	NIL

OPEN	v1	v2	v3
v_visit	v0		
CLOSED	v0		
S			
tmp_g	2	1	5

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
  v_visit = min-f (OPEN)
  if v_visit == v_goal then done
    add(v_visit, CLOSED)
    generate set S    {v1,v2,v3}
          of all successors of v_visit
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) +
                   w(v_visit, v_next)
        if tmp_g(v_next) < g(v_next)   =inf
          path-parent = v_visit
          g(v_next) = tmp_g(v_next)
          f(v_next) = g(v_next) + h(v_next)
          if v_next not in OPEN then
{v1,v2,v3}    add(v_next, OPEN)
return fail
```

Pseudo-Code A*

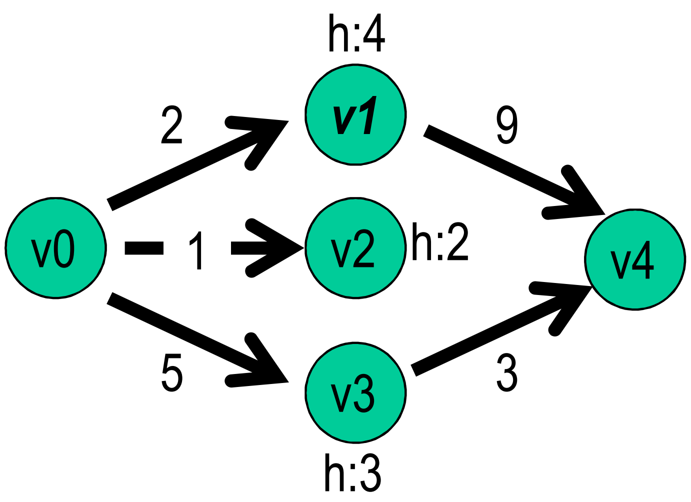


	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	6	v0
v2	2	1	**3**	v0
v3	3	5	8	v0
v4	0	inf	inf	NIL

OPEN	v1	**v2**	v3
v_visit	v2		
CLOSED	v0	v2	
S			
tmp_g			

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
   v_visit = min-f (OPEN)     v2
   if v_visit == v_goal then done
      add(v_visit, CLOSED)
      generate set S            S={}
               of all successors of v_visit
      forall v_next in S
         if v_next not in CLOSED
            tmp_g(v_next) = g(v_visit) +
                              w(v_visit, v_next)
            if tmp_g(v_next) < g(v_next)
               path-parent = v_visit
               g(v_next) = tmp_g(v_next)
               f(v_next) = g(v_next) + h(v_next)
               if v_next not in OPEN then
                              add(v_next, OPEN)
return fail
```

Pseudo-Code A*

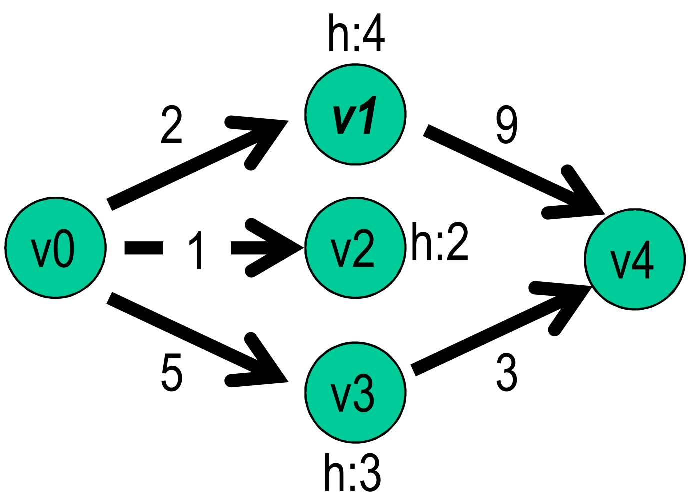


	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	**6**	v0
v2	2	1	3	v0
v3	3	5	8	v0
v4	0	inf	inf	NIL

OPEN	**v1**	v3	
v_visit	v1		
CLOSED	v0	v2	v1
S	v4		
tmp_g	11		

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
  v_visit = min-f (OPEN)     v1
  if v_visit == v_goal then done
    add(v_visit, CLOSED)
    generate set S           S={v4}
          of all successors of v_visit
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) +
          2+9=11       w(v_visit, v_next)
        if tmp_g(v_next) < g(v_next)
          path-parent = v_visit
          g(v_next) = tmp_g(v_next)
          f(v_next) = g(v_next) + h(v_next)
          if v_next not in OPEN then
                     add(v_next, OPEN)
return fail
```

Pseudo-Code A*

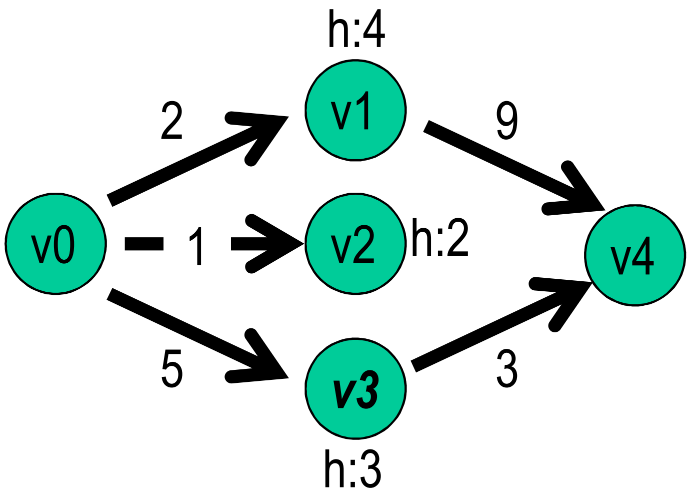


	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	**6**	v0
v2	2	1	3	v0
v3	3	5	8	v0
v4	0	11	11	v1

OPEN	v3	v4	
v_visit	v1		
CLOSED	v0	v2	v1
S	v4		
tmp_g	11		

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
  v_visit = min-f (OPEN)
  if v_visit == v_goal then done
    add(v_visit, CLOSED)
    generate set S
          of all successors of v_visit
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) +
                     w(v_visit, v_next)
        11
        if tmp_g(v_next) < g(v_next)   =inf
          path-parent = v_visit
          g(v_next) = tmp_g(v_next)
          f(v_next) = g(v_next) + h(v_next)
          if v_next not in OPEN then
                     add(v_next, OPEN)
return fail
```

Pseudo-Code A*

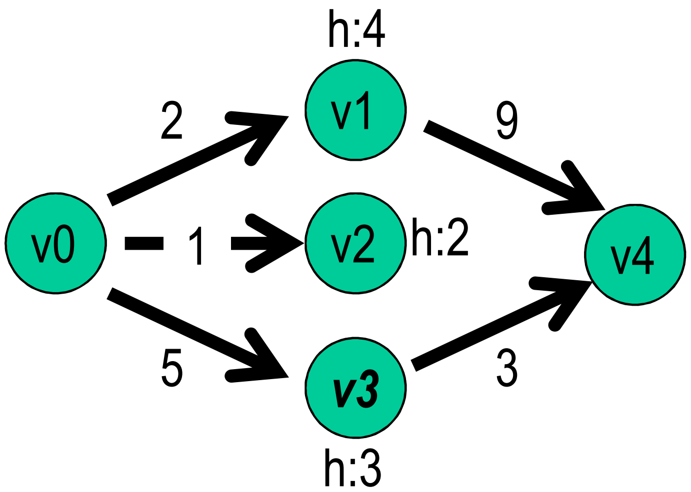



```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
   v_visit = min-f (OPEN)     v3
   if v_visit == v_goal then done
      add(v_visit, CLOSED)
      generate set S           S={v4}
               of all successors of v_visit
      forall v_next in S
         if v_next not in CLOSED
            tmp_g(v_next) = g(v_visit) +
            5+3=8        w(v_visit, v_next)
            if tmp_g(v_next) < g(v_next)
               path-parent = v_visit
               g(v_next) = tmp_g(v_next)
               f(v_next) = g(v_next) + h(v_next)
               if v_next not in OPEN then
                                 add(v_next, OPEN)
return fail
```

	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	6	v0
v2	2	1	3	v0
v3	3	5	**8**	v0
v4	0	11	11	v1

OPEN	**v3**	v4		
v_visit	v3			
CLOSED	v0	v2	v1	v3
S	v4			
tmp_g	8			

Pseudo-Code A*



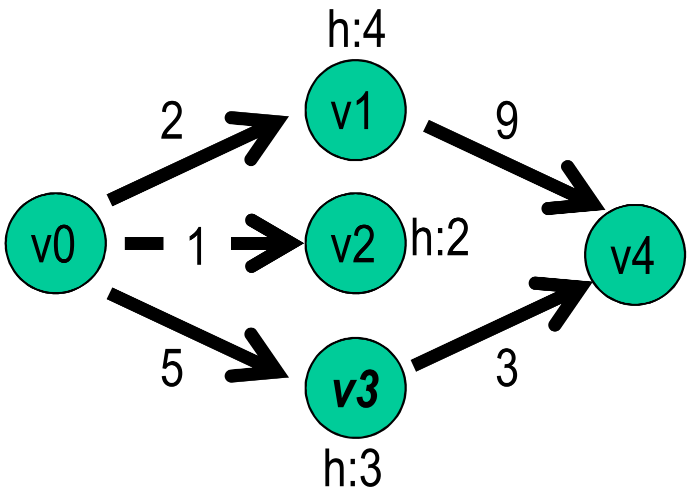
	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	6	v0
v2	2	1	3	v0
v3	3	5	**8**	v0
v4	0	11	11	v1

OPEN	v4			
v_visit	v3			
CLOSED	v0	v2	v1	v3
S	v4			
tmp_g	8			

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
  v_visit = min-f (OPEN)
  if v_visit == v_goal then done
    add(v_visit, CLOSED)
    generate set S
          of all successors of v_visit
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) +
                               w(v_visit, v_next)
        if tmp_g(v_next) < g(v_next)
          path-parent = v_visit
          g(v_next) = tmp_g(v_next)
          f(v_next) = g(v_next) + h(v_next)
          if v_next not in OPEN then
                    add(v_next, OPEN)
return fail
```

tmp_g(v4)=7 **g(v4)=11**

Pseudo-Code A*

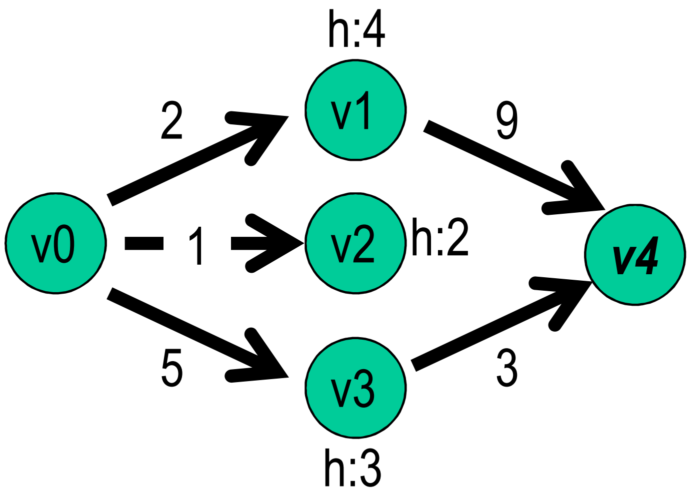


	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	6	v0
v2	2	1	3	v0
v3	3	5	8	v0
v4	0	8	8	v3

OPEN	v4			
v_visit	v3			
CLOSED	v0	v2	v1	v3
S	v4			
tmp_g	7			

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}
  v_visit = min-f (OPEN)
  if v_visit == v_goal then done
    add(v_visit, CLOSED)
    generate set S
          of all successors of v_visit
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) +
                        w(v_visit, v_next)
        if tmp_g(v_next) < g(v_next)
          path-parent = v_visit
          g(v_next) = tmp_g(v_next)
          f(v_next) = g(v_next) + h(v_next)
          if v_next not in OPEN then
                        add(v_next, OPEN)
return fail
```

Pseudo-Code A*



	h()	g()	f()	path-p
v0	-	0	0	NIL
v1	4	2	6	v0
v2	2	1	3	v0
v3	3	5	8	v0
v4	0	8	8	v3

OPEN					
v_visit	v4				
CLOSED	v0	v2	v1	v3	
S					
tmp_g					

```
forall v : g(v) = infinity; g(v_start) = 0
forall v : f(v) = infinity; f(v_start) = h(v_start)
add(v_start, OPEN)
while OPEN != {}     {v4}
  v_visit = min-f (OPEN)     v4
  if v_visit == v_goal then done
    add(v_visit, CLOSED)
    generate set S
          of all successors of v_visit
    forall v_next in S
      if v_next not in CLOSED
        tmp_g(v_next) = g(v_visit) +
                     w(v_visit, v_next)
        if tmp_g(v_next) < g(v_next)
          path-parent = v_visit
          g(v_next) = tmp_g(v_next)
          f(v_next) = g(v_next) + h(v_next)
          if v_next not in OPEN then
                     add(v_next, OPEN)
return fail
```

Properties of A*

A* generates an **optimal** solution

- if h(n) is an **admissible** heuristic
- and the search space is a **tree**

h(n) is **admissible**

- if it never overestimates the cost to reach the destination
- i.e., it is optimistically underestimating the cost

Properties of A*

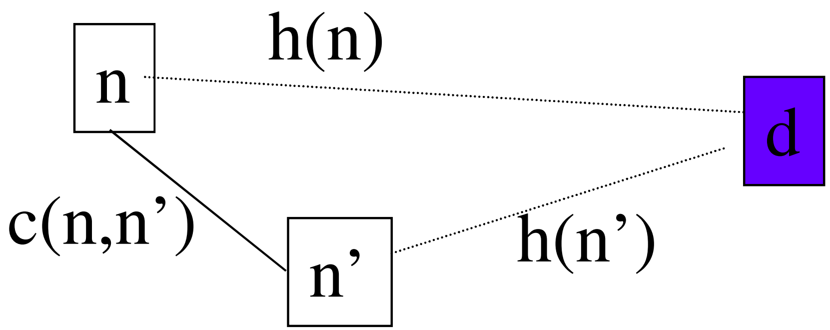
A* generates an **optimal** solution

- if h(n) is a **consistent** heuristic
- and the search space is a **graph**

h(n) is **consistent**

- if for every node n & for every successor node n' of n
- it holds that: $h(n) \leq c(n,n') + h(n')$

If h(n) is consistent then
the values of f(n) along any path
are non-decreasing
(kind of triangle inequality)



Properties of A*

Note:

- if h(n) is consistent then h(n) is admissible
- often, when h(n) is admissible, it is also consistent

Computational Cost of A*

it can also be shown that

- A* makes optimal use of the heuristics
- i.e., there is no search algorithm that
  - expands fewer nodes using the heuristic
  - and finds the optimal solution

Creating Good Heuristics

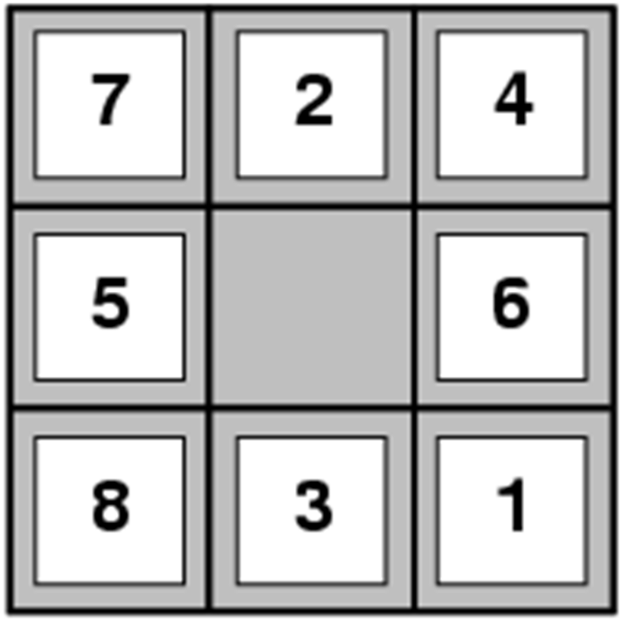
desired properties

- consistent / admissible
- as close to the optimum as possible (without overestimate)
- easy to compute

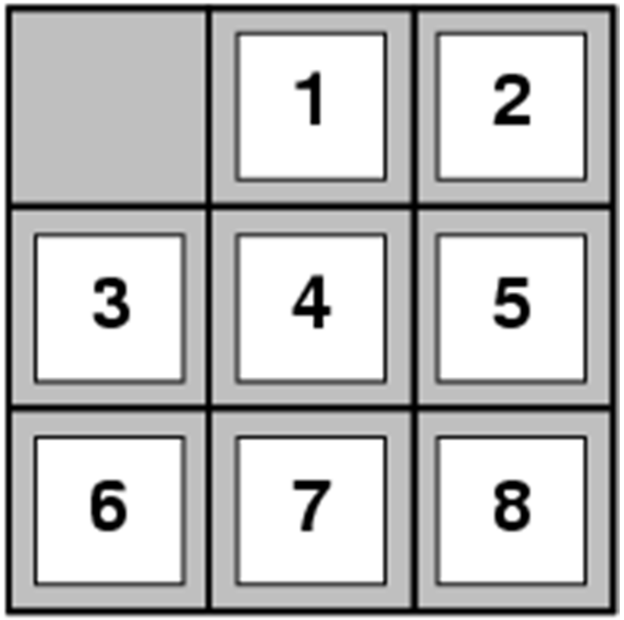
one general strategy: problem **relaxation**

- simplify problem by reducing restrictions aka constraints on actions (aka relaxed problem)
- this results in admissible heuristics

Example: 8-Puzzle



Start State



Goal State

sort the tiles into order by sliding motion of just 1 tile

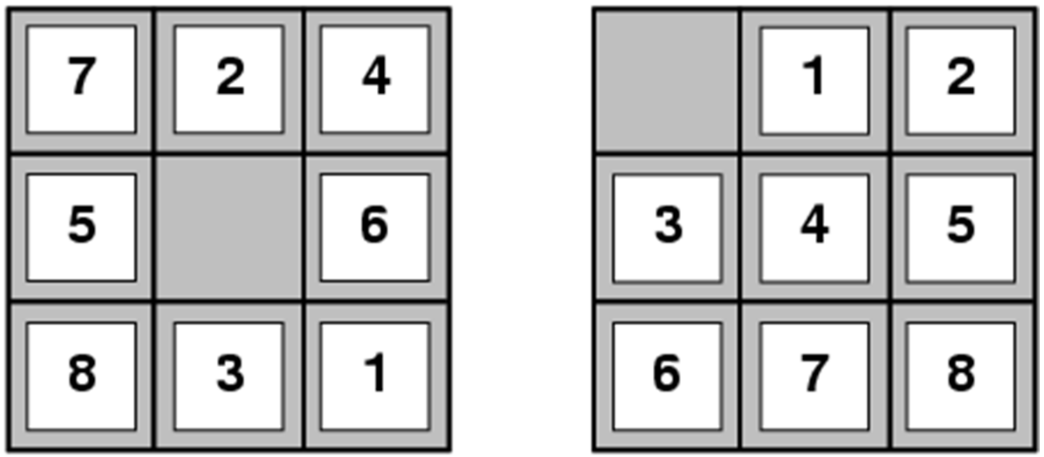
(play e.g. at *http://mypuzzle.org/sliding*)

Example: 8-Puzzle

<u>Action Constraint</u>

a tile can move from square A to square B iff

- A is horizontally or vertically adjacent to B
- and B is blank



Start State   Goal State

Admissible Heuristics in 8-Puzzle

Heuristic 1: tile A can be moved to any tile B

- H1(n) = "number of misplaced tiles in board n"

Heuristic 2: tile A can be moved to tile B if B is adjacent to A

- H2(n) = "sum of Manhattan distances of misplaced tiles to goal positions in board n"

Experimental results (Russell & Norvig, 2002):

- A* with h2 performs up to 10 times better than A* with h1
- A* with h2 performs up to 36,000 times better than a classical uninformed search algorithm (iterative deepening)