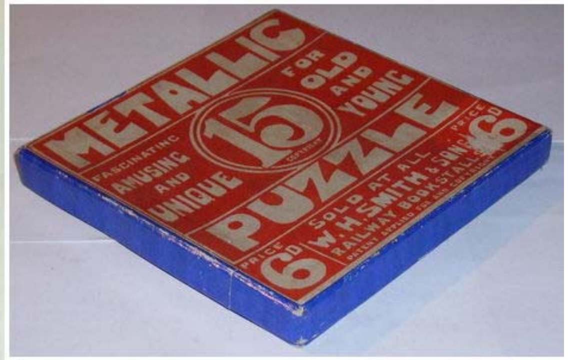
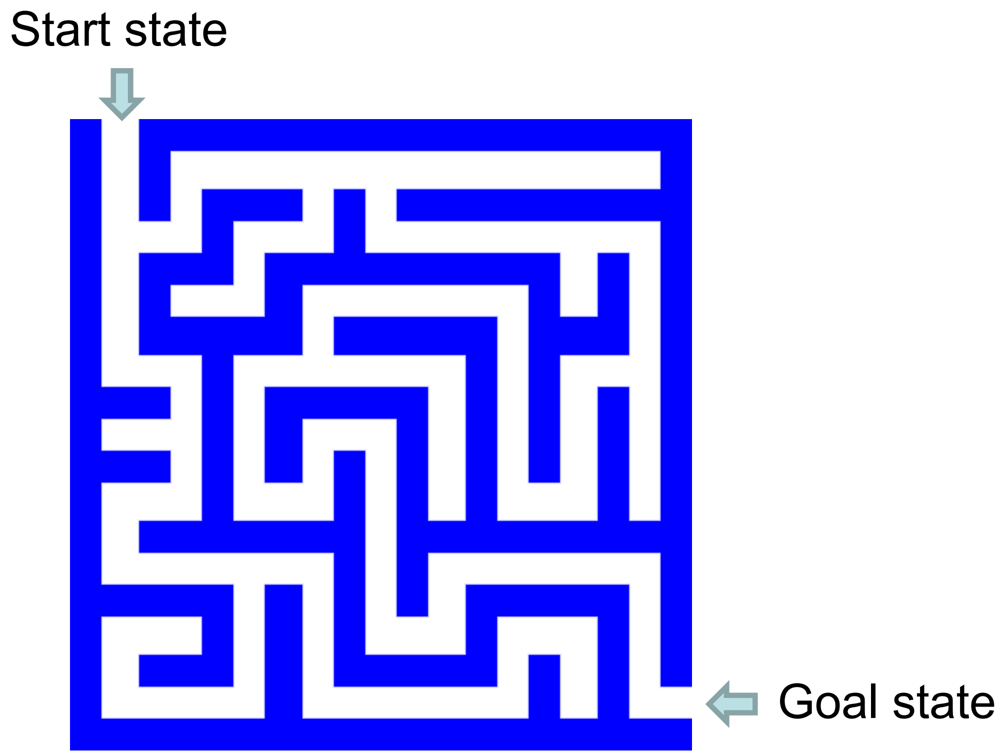
Solving problems by searching

Chapter 3



Search

- We will consider the problem of designing **goal-based agents** in **fully observable**, **deterministic**, **discrete**, **known** environments



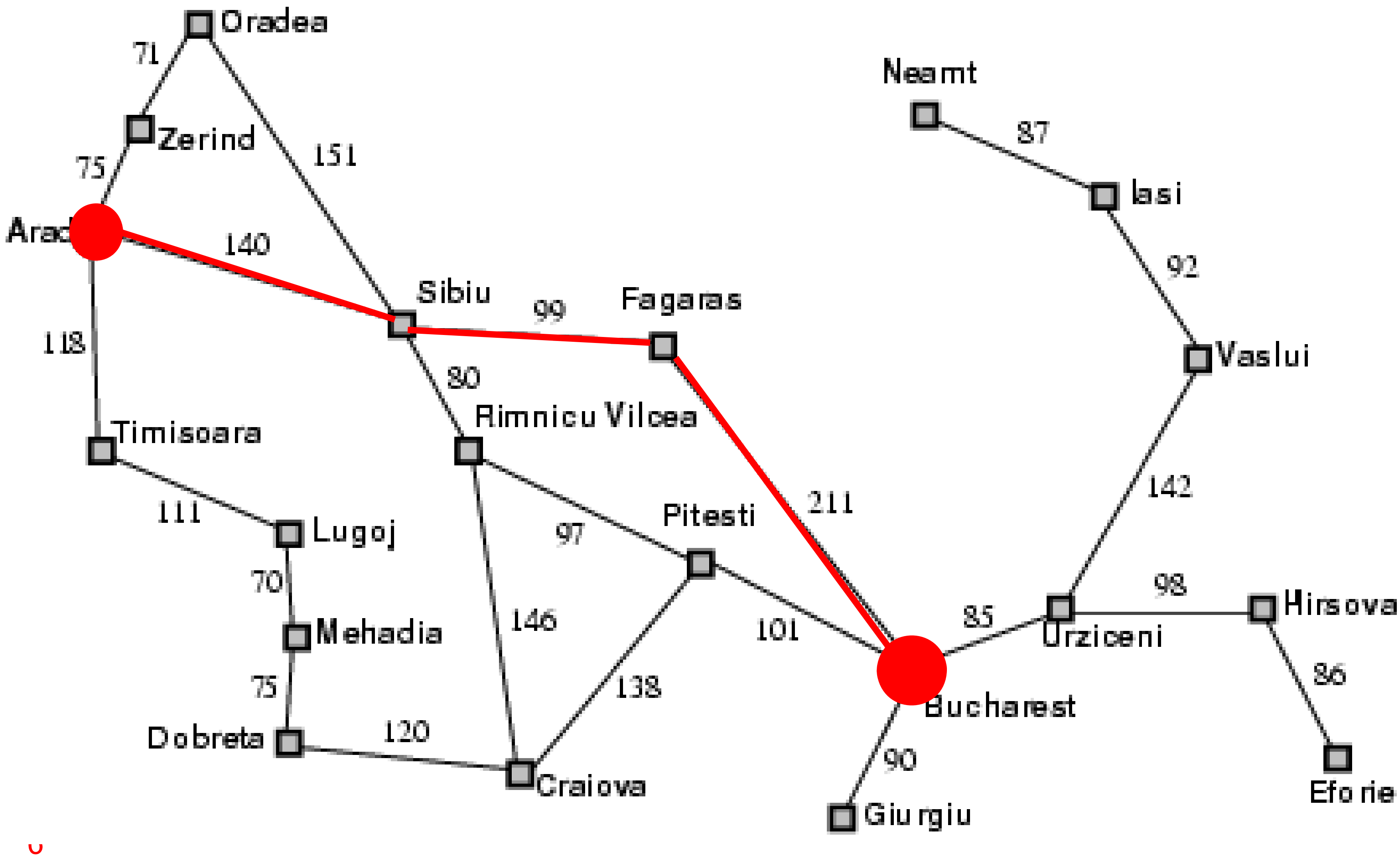
Search

- We will consider the problem of designing **goal-based agents** in **fully observable**, **deterministic**, **discrete**, **known** environments
  - The agent must find a *sequence of actions* that reaches the goal
  - The **performance measure** is defined by (a) reaching the goal and (b) how "expensive" the path to the goal is
  - We are focused on the process of finding the solution; while executing the solution, we assume that the agent can safely ignore its percepts (**open-loop system**)

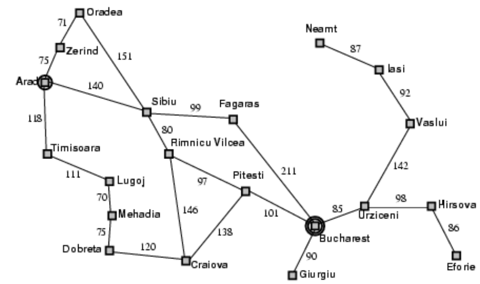
Search Problems

Search Problems Are Models

Example: Romania



Single-state problem formulation



A **problem** is defined by four items:

1. initial state e.g., "at Arad"

2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad)* = {*<Arad → Zerind, Zerind>, … *}

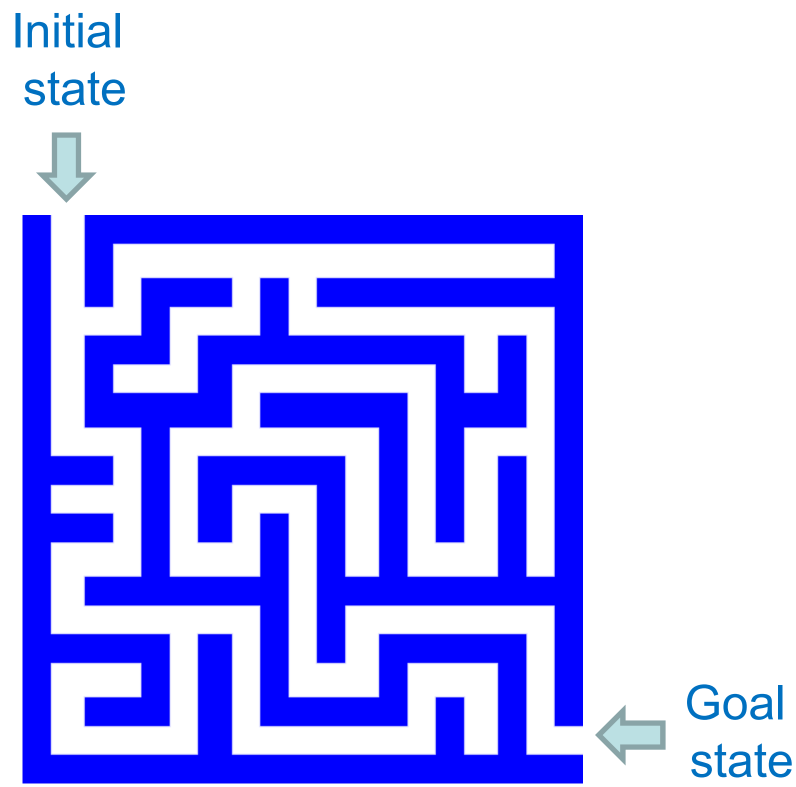
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*

4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state

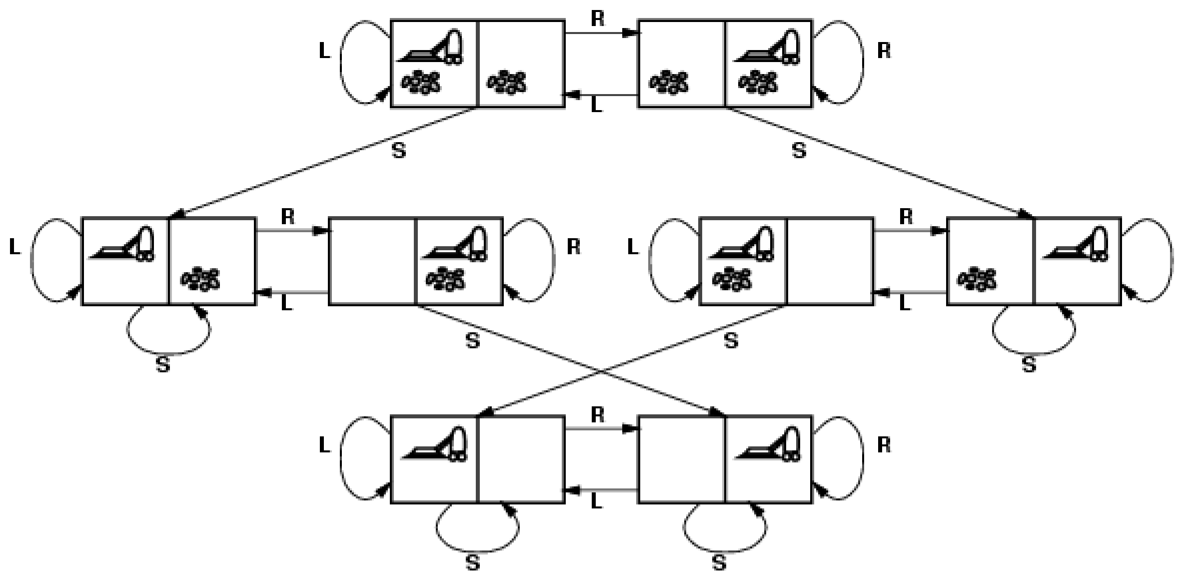
Search problem components

- **Initial state**
- **Actions**
- **Transition model**
  - What state results from performing a given action in a given state?
- **Goal state**
- **Path cost**
  - Assume that it is a sum of nonnegative *step costs*



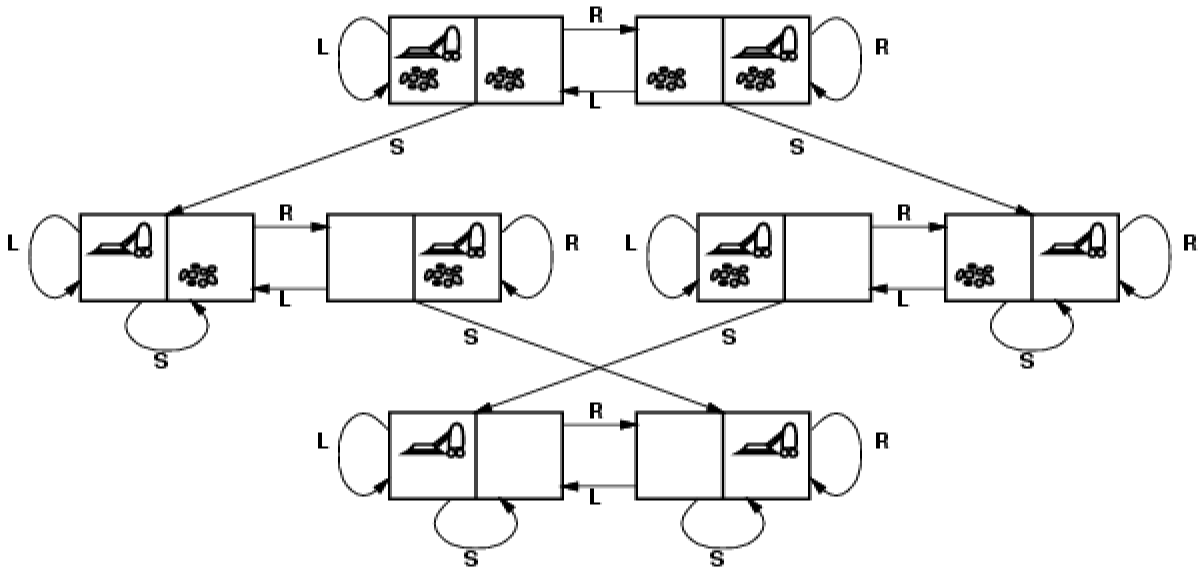
- The **optimal solution** is the sequence of actions that gives the *lowest* path cost for reaching the goal

Vacuum world state space graph



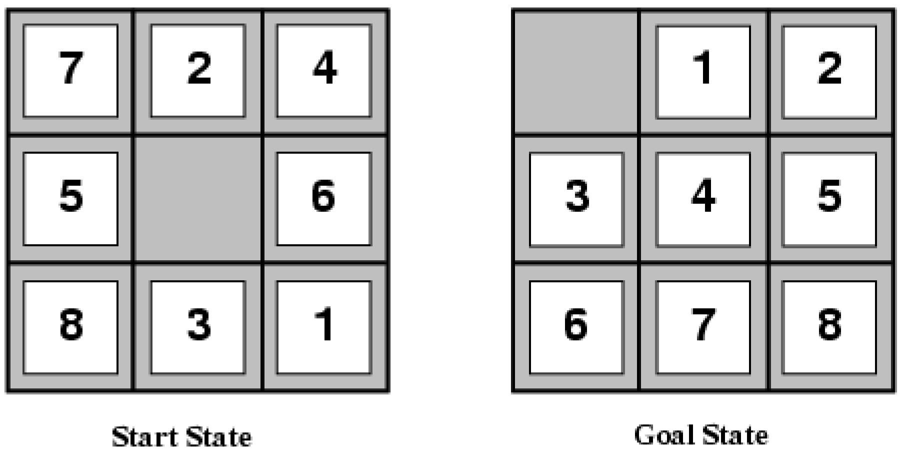
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



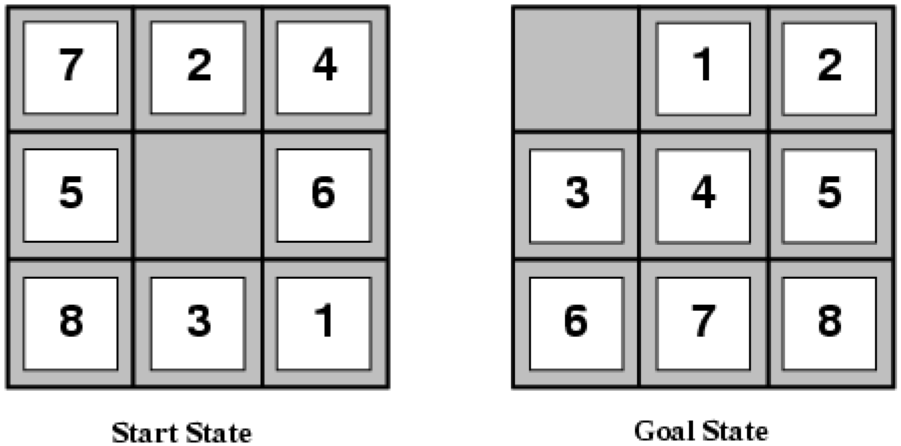
- states? dirt and robot location
- actions? *Left*, *Right*, *Clean*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



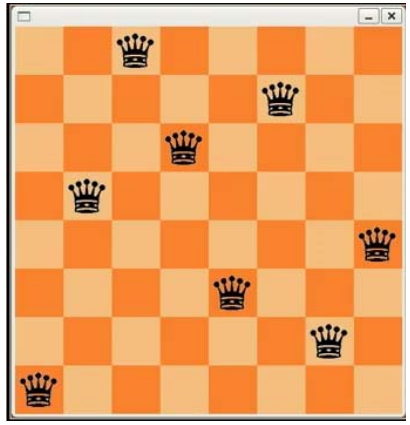
- states?
- actions?
- goal test?
- path cost?

Example: The 8-puzzle



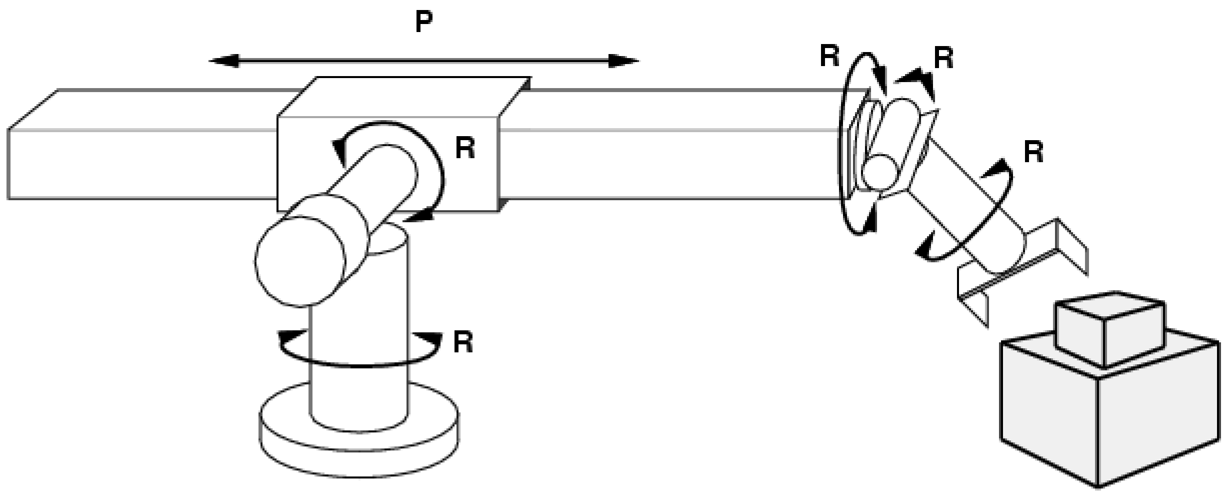
- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

Example: 8 queen



- states?: Any arrangement of 0 to 8 queens on the board is a state
- Initial state: No queen on the board
- Actions?: Add a queen to any empty square
- goal test?: 8 queens are on the board, none attacked

Example: Robot motion planning



- **States**
  – Real-valued joint parameters (angles, displacements)
- **Actions**
  – Continuous motions of robot joints
- **Goal state**
  – Configuration in which object is grasped
- **Path cost**
  – Time to execute, smoothness of path, etc.

Search

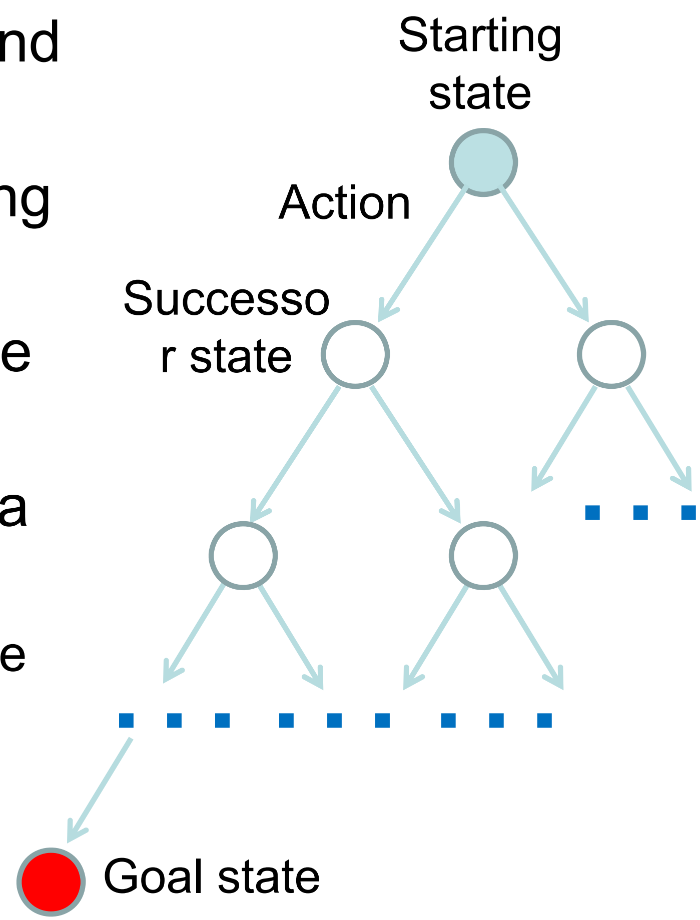
- Given:
  - **Initial state**
  - **Actions**
  - **Transition model**
  - **Goal state**
  - **Path cost**
- How do we find the optimal solution?
  - How about building the state space and then using Dijkstra's shortest path algorithm?
    - Complexity of Dijkstra's is $O(E + V \log V)$, where V is the size of the state space
    - The state space may be huge!

Search: Basic idea

- Let's begin at the start state and **expand** it by making a list of all possible successor states
- Maintain a **frontier** or a list of unexpanded states
- At each step, pick a state from the frontier to expand
- Keep going until you reach a goal state
- Try to expand as few states as possible

Search tree

- “What if” tree of sequences of actions and outcomes
- The root node corresponds to the starting state
- The children of a node correspond to the **successor states** of that node’s state
- A path through the tree corresponds to a sequence of actions
  - A solution is a path ending in the goal state
- Nodes vs. states
  - A state is a representation of the world, while a node is a data structure that is part of the search tree
    - Node has to keep pointer to parent, path cost, possibly other info

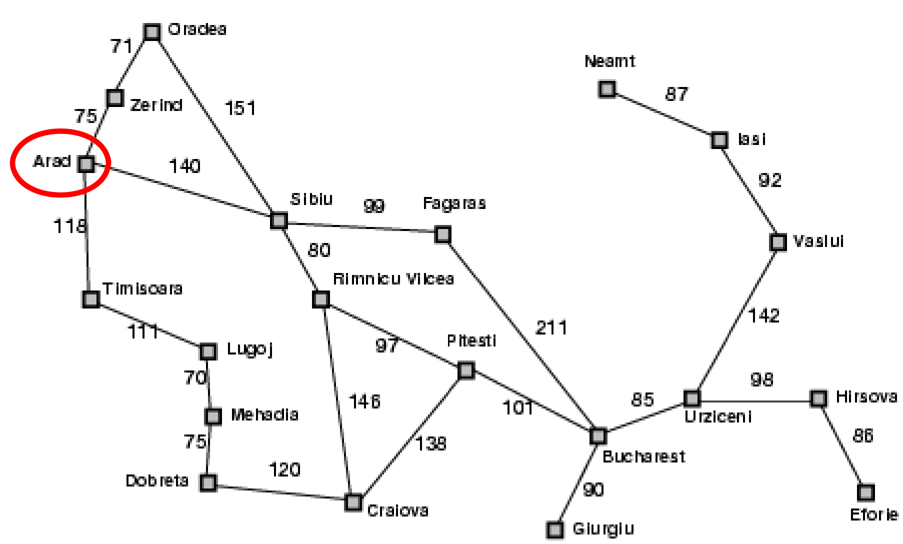


Tree Search Algorithm Outline

- Initialize the **frontier** using the **starting state**
- While the frontier is not empty
  - Choose a frontier node according to **search strategy** and take it off the frontier
  - If the node contains the **goal state**, return solution
  - Else **expand** the node and add its children to the frontier

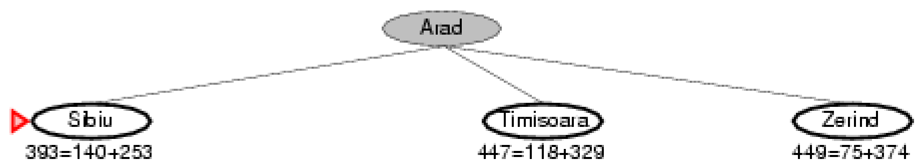
Tree search example

Start: Arad
Goal: Bucharest

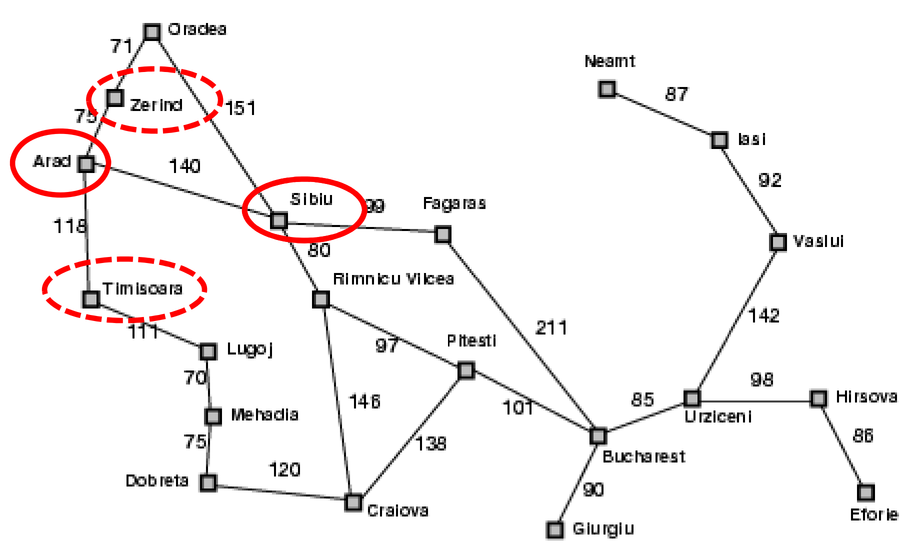


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Tree search example



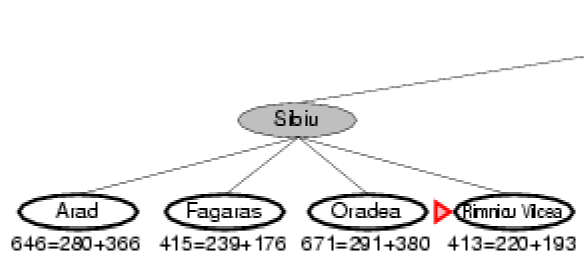
Start: Arad
Goal: Bucharest

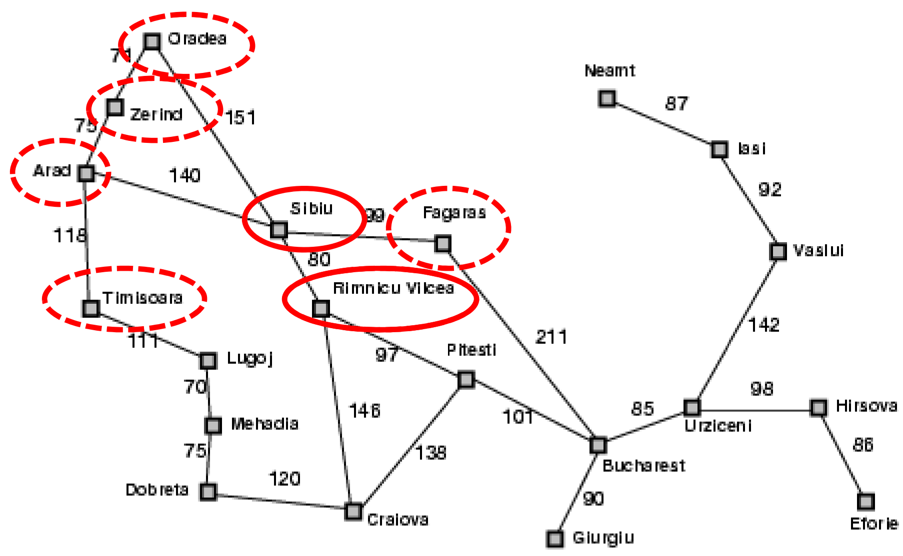


Straight–line distance to Bucharest

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

Tree search example

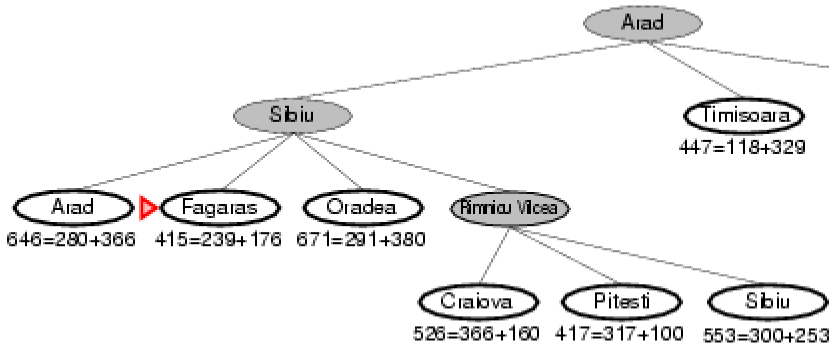




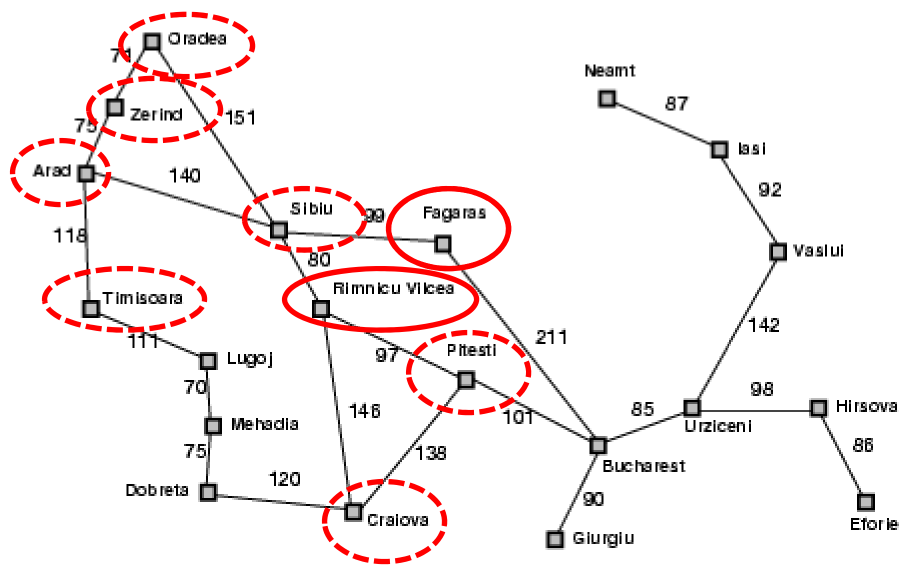
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Start: Arad
Goal: Bucharest

Tree search example

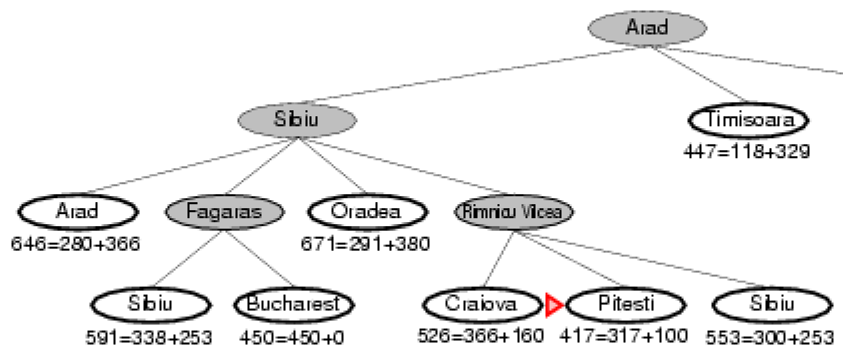


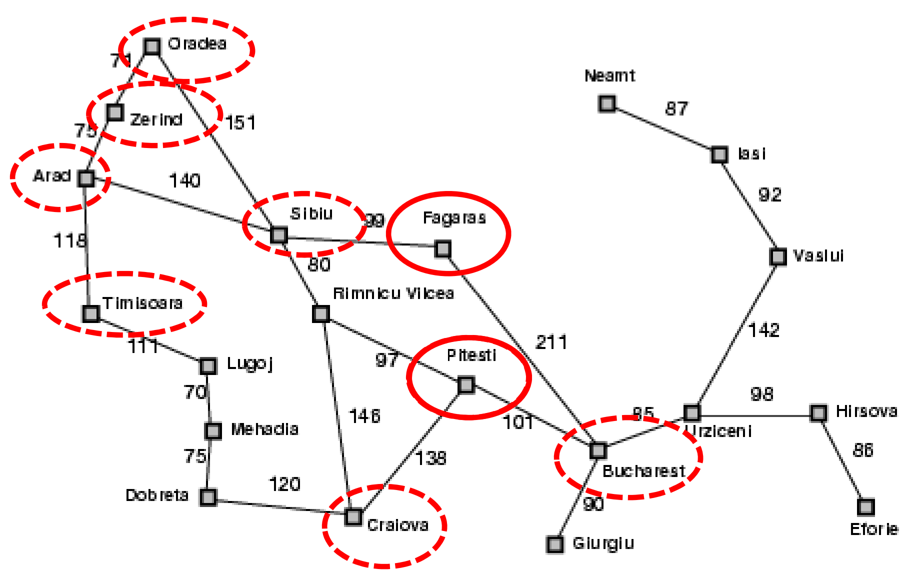
Start: Arad
Goal: Bucharest



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

Tree search example



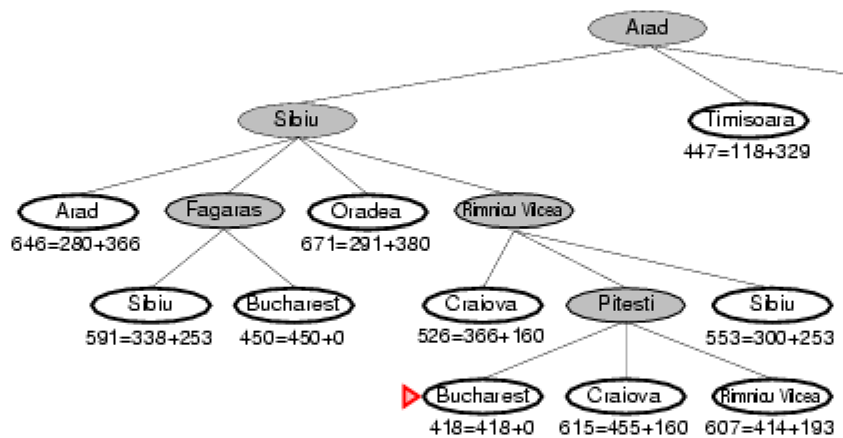


Straight–line distance to Bucharest

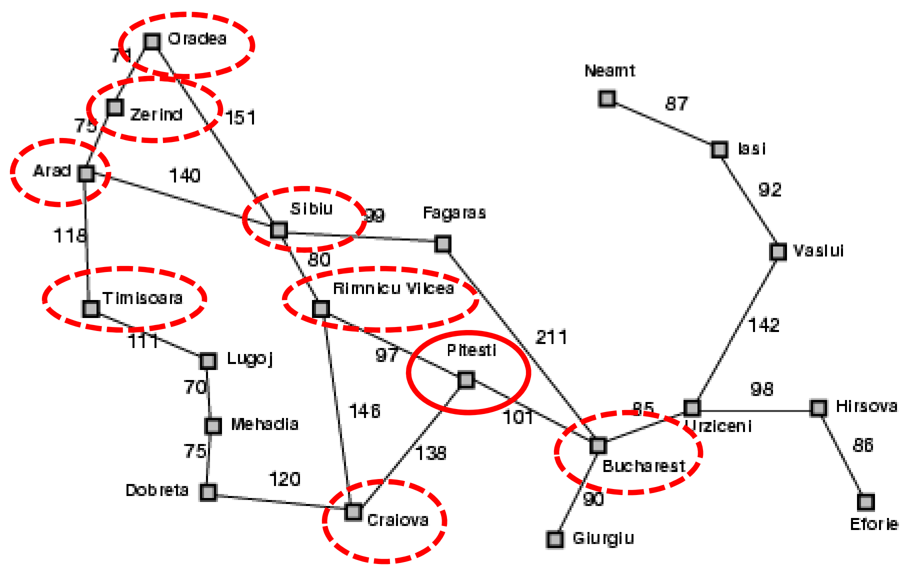
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Start: Arad
Goal: Bucharest

Tree search example



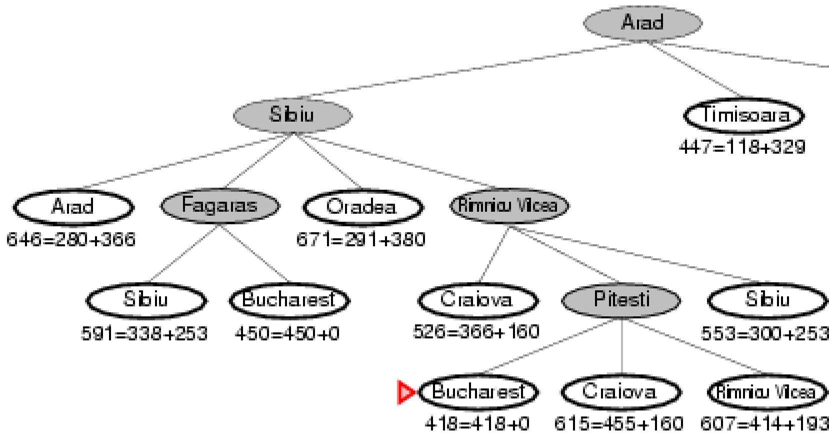
Start: Arad
Goal: Bucharest



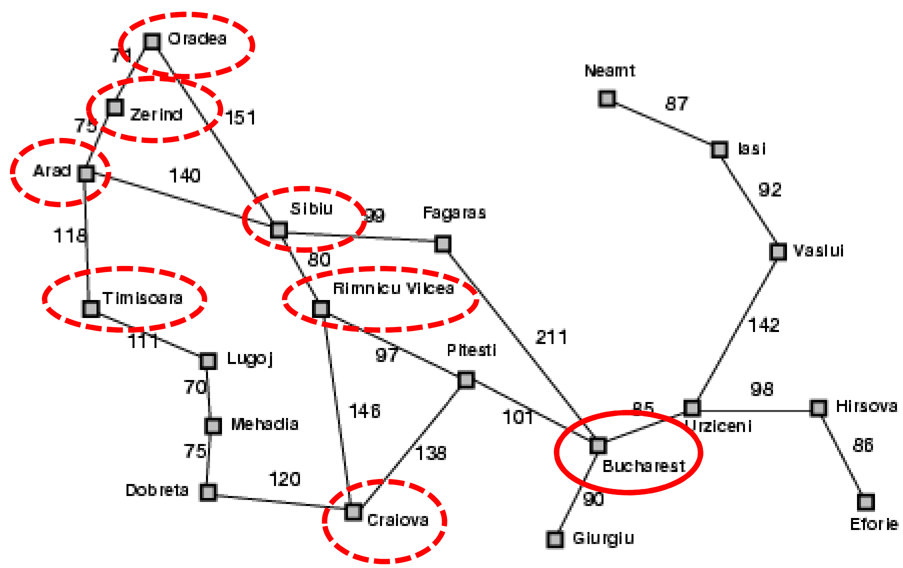
Straight–line distance
to Bucharest

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

Tree search example



Start: Arad
Goal: Bucharest



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

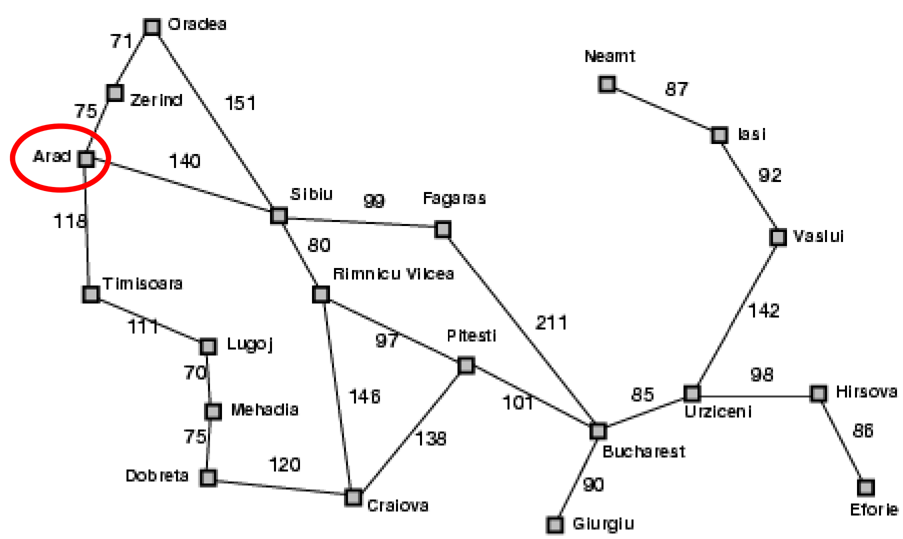
Handling repeated states

- Initialize the **frontier** using the **starting state**
- While the frontier is not empty
  - Choose a frontier node according to **search strategy** and take it off the frontier
  - If the node contains the **goal state**, return solution
  - Else **expand** the node and add its children to the frontier

- To handle repeated states:
  - Every time you expand a node, add that state to the **explored set**; do not put explored states on the frontier again
  - Every time you add a node to the frontier, check whether it already exists in the frontier with a higher path cost, and if yes, replace that node with the new one

Search without repeated states

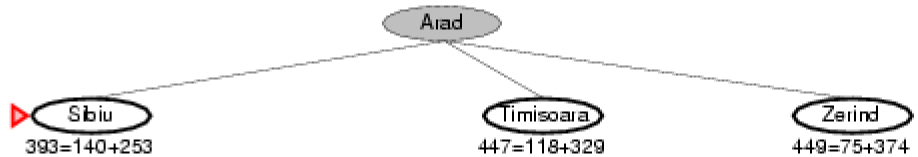
Start: Arad
Goal: Bucharest



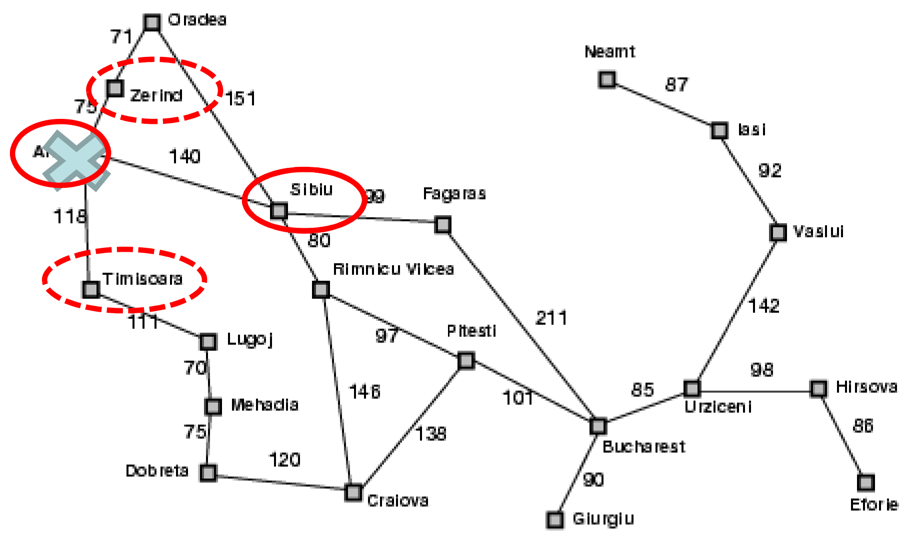
Straight–line distance
to Bucharest

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

Search without repeated states

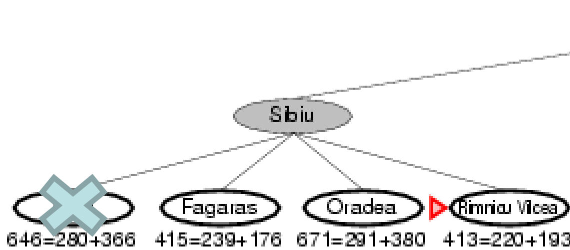


Start: Arad
Goal: Bucharest

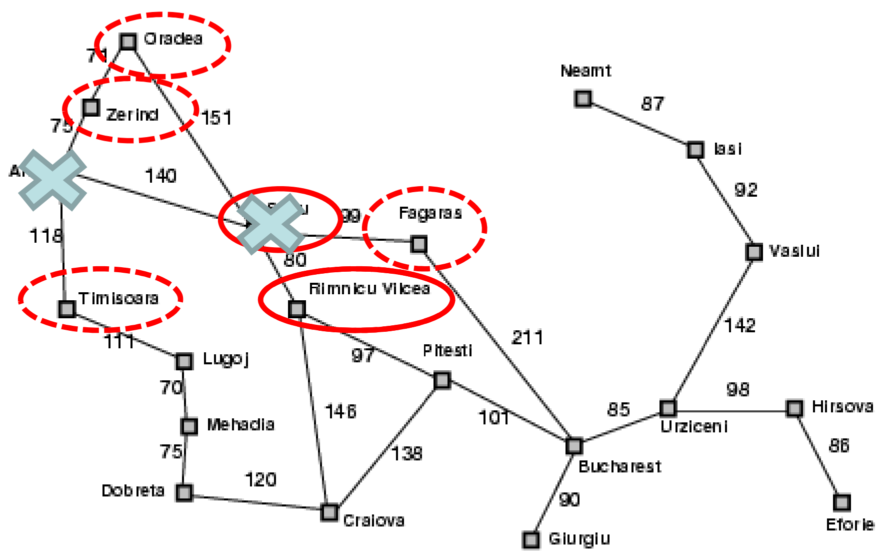


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Search without repeated states



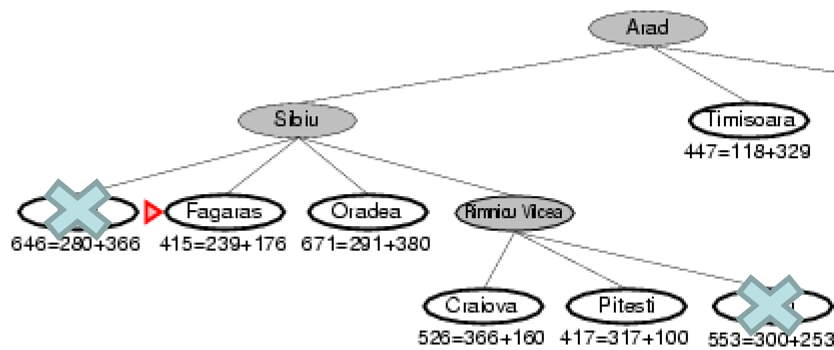
Start: Arad
Goal: Bucharest



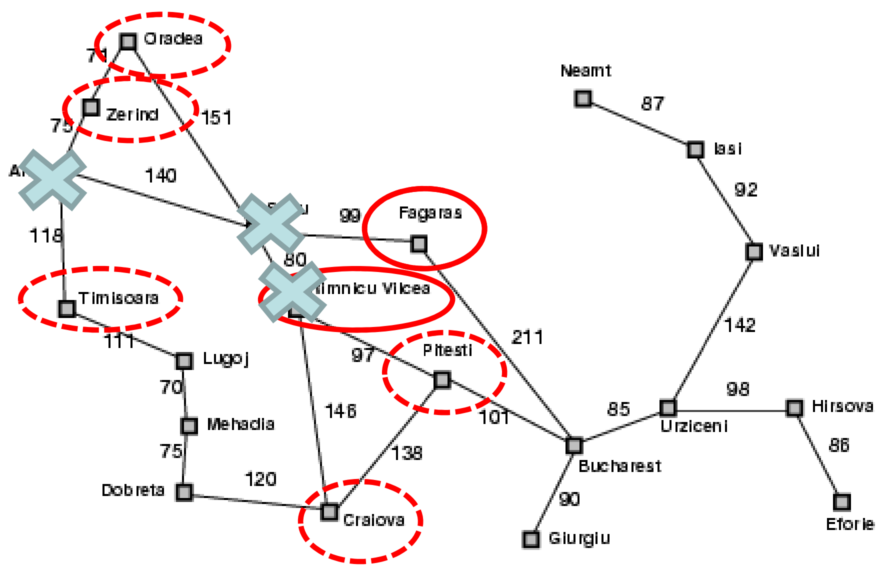
Straight–line distance to Bucharest

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

Search without repeated states



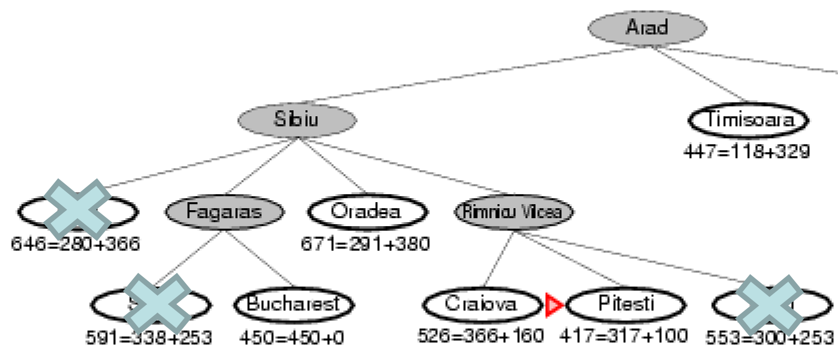
Start: Arad
Goal: Bucharest

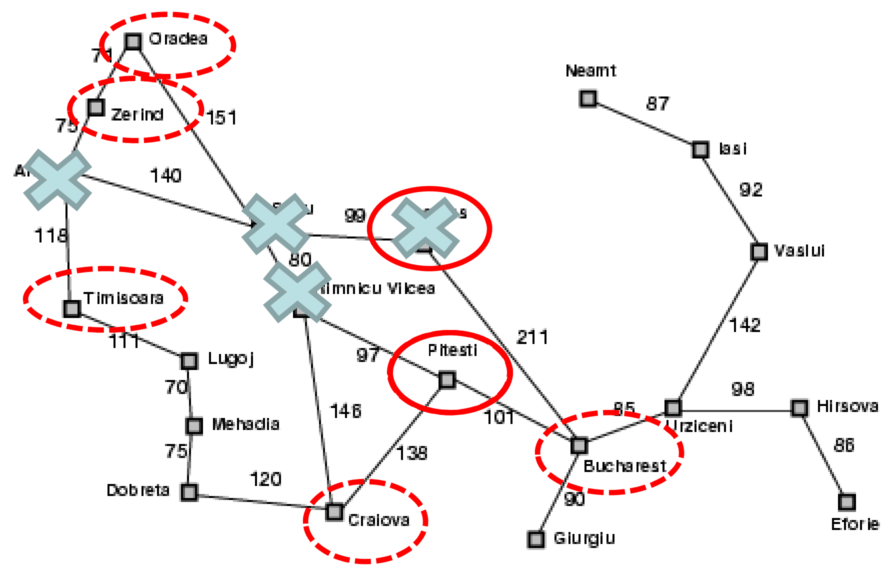


Straight–line distance
to Bucharest

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

Search without repeated states

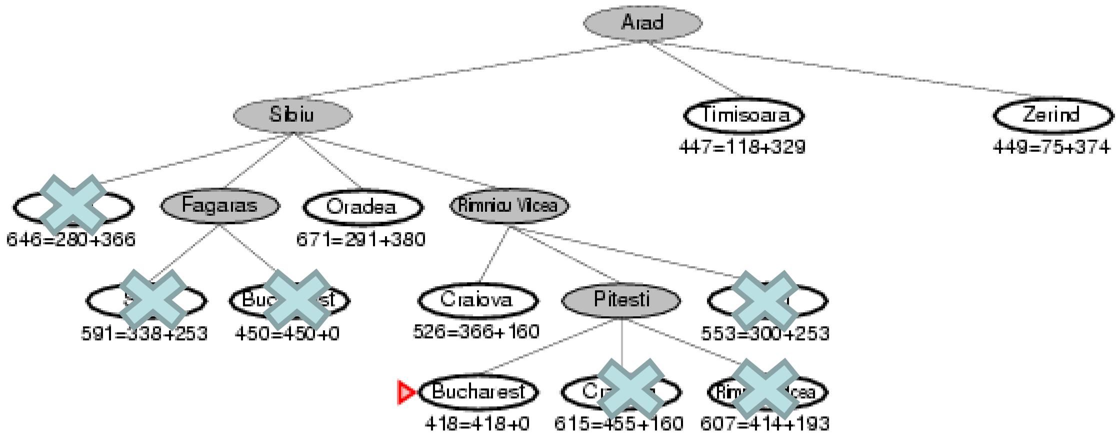




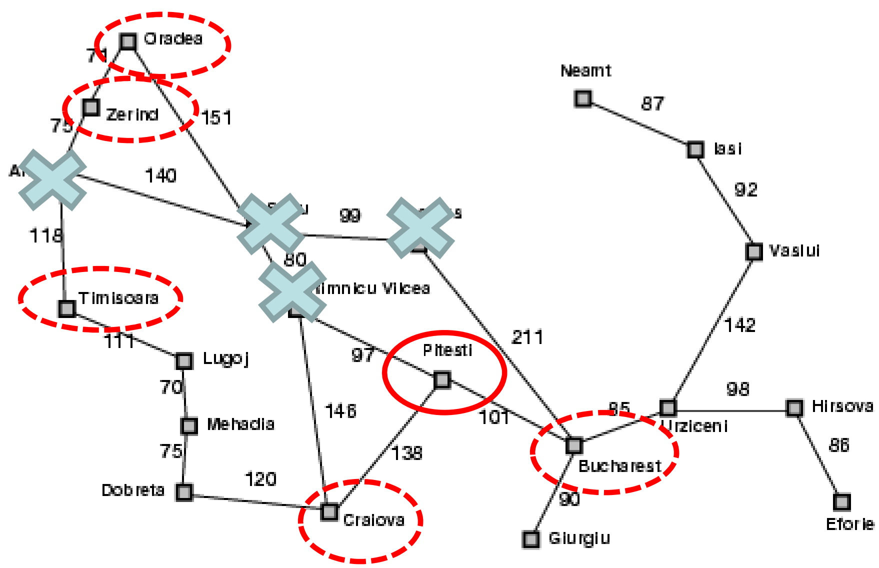
Start: Arad
Goal: Bucharest

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

Search without repeated states



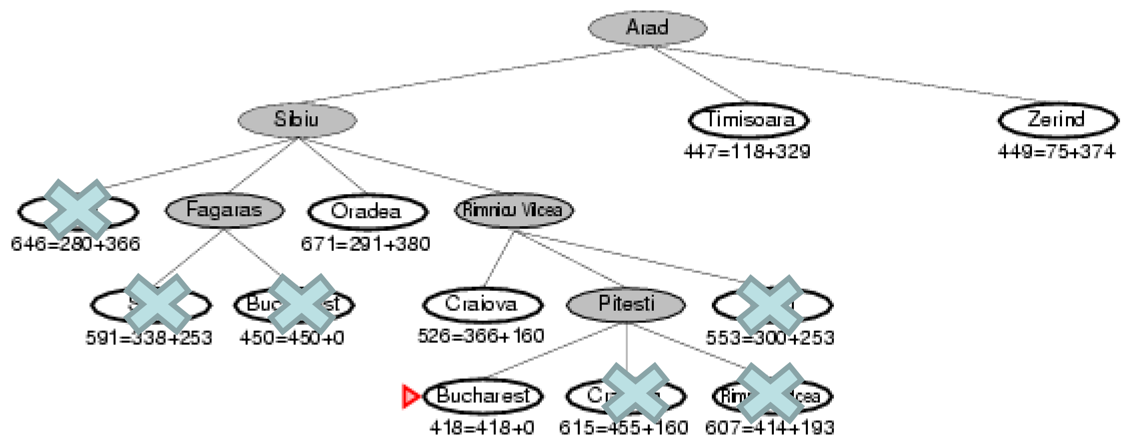
Start: Arad
Goal: Bucharest



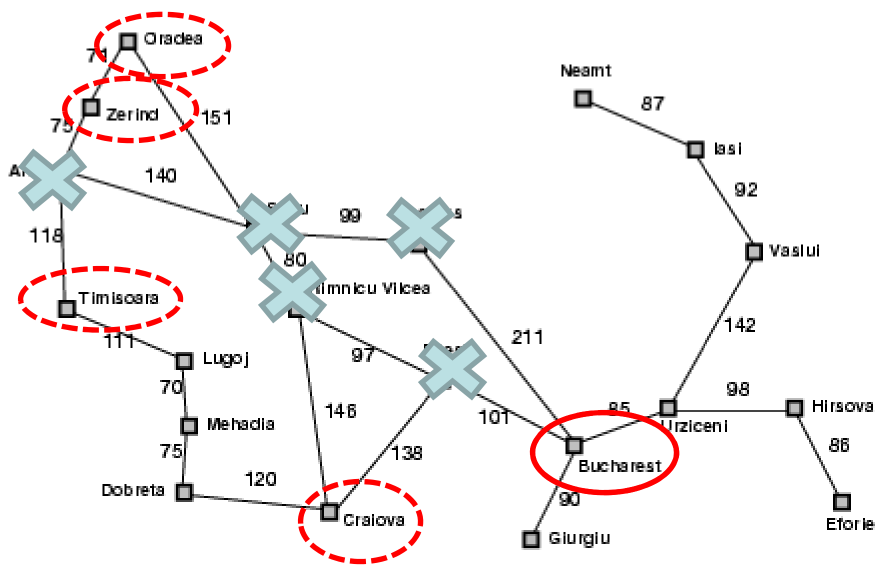
Straight–line distance to Bucharest

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

Search without repeated states



Start: Arad
Goal: Bucharest

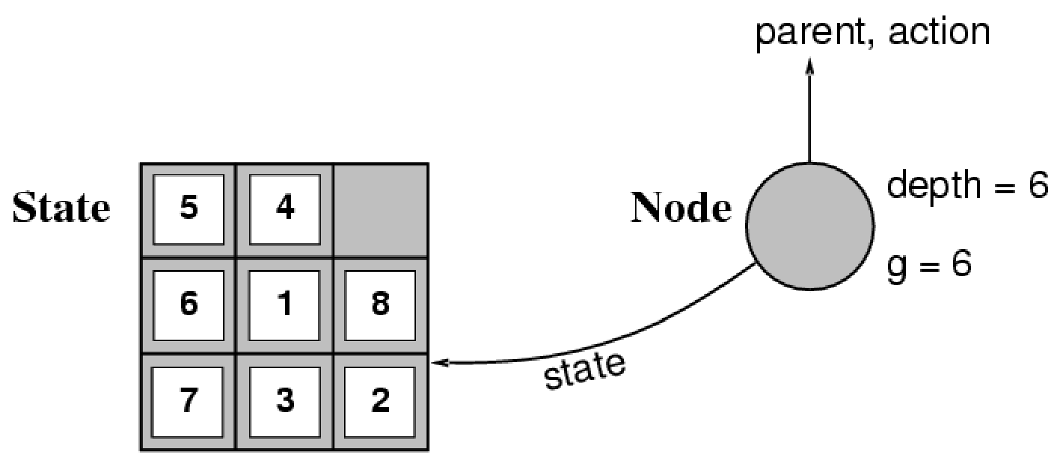


Straight–line distance to Bucharest

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

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth



Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

Uninformed search strategies

- Uninformed (blind) search strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Graph & BFS

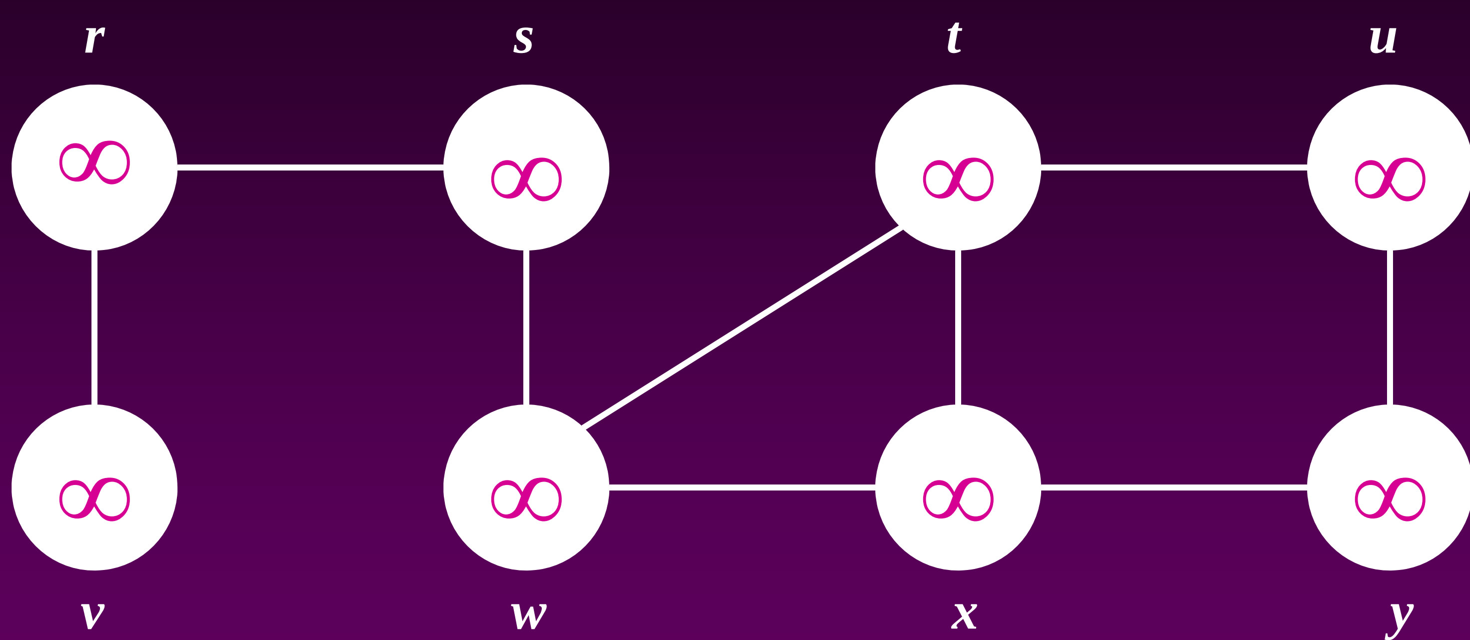
Breadth-First Search: The Code

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
   for each vertex u ∈ V-{s}
   {
      color[u]=WHITE;
       prev[u]=NIL;
       d[u]=inf;
   }
   color[s]=GRAY;
  d[s]=0; prev[s]=NIL;
  Q=empty;
  ENQUEUE(Q,s);
```

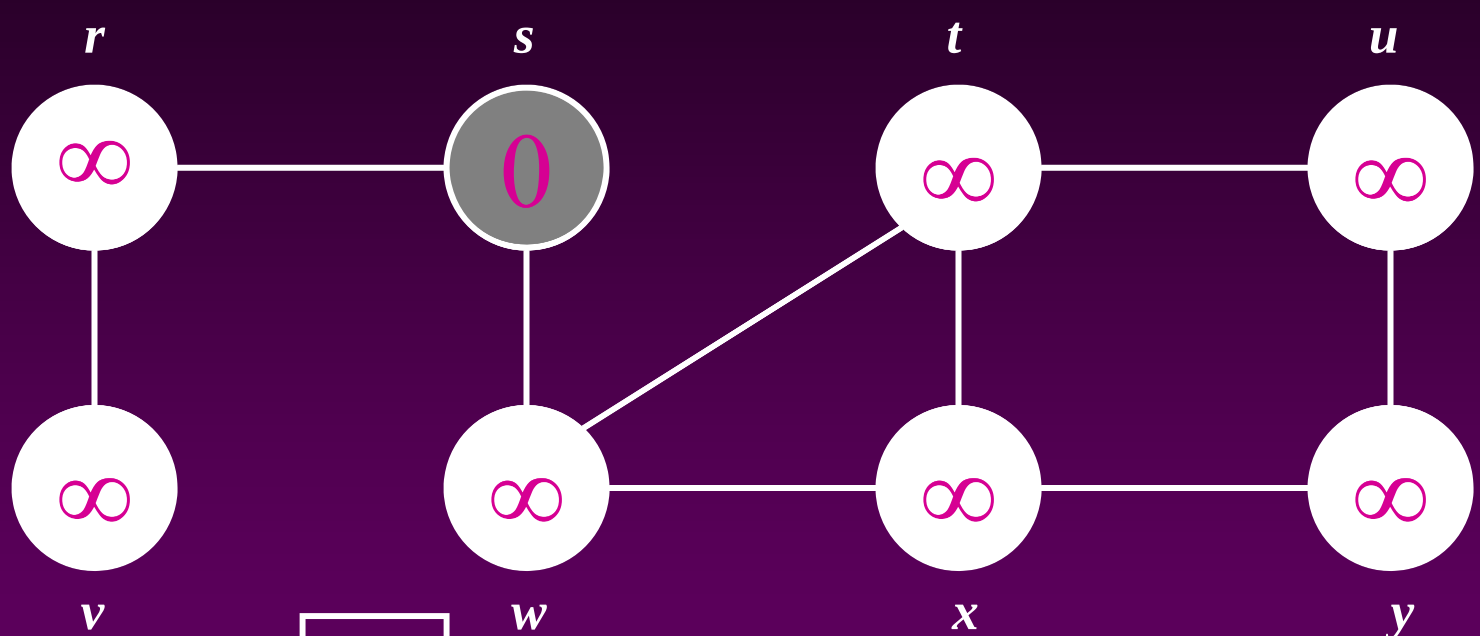
```
While(Q not empty)
{
  u = DEQUEUE(Q);
  for each v ∈ adj[u]{
    if (color[v] ==
 WHITE){
        color[v] = GREY;
        d[v] = d[u] + 1;
        prev[v] = u;
        Enqueue(Q, v);
    }
  }
  color[u] = BLACK;
 }
}
```

Breadth-First Search: Example



Vertex	r	s	t	u	v	w	x	y
color	W	W	W	W	W	W	W	W
d	∞	∞	∞	∞	∞	∞	∞	∞
prev	nil	nil	nil	nil	nil	nil	nil	nil

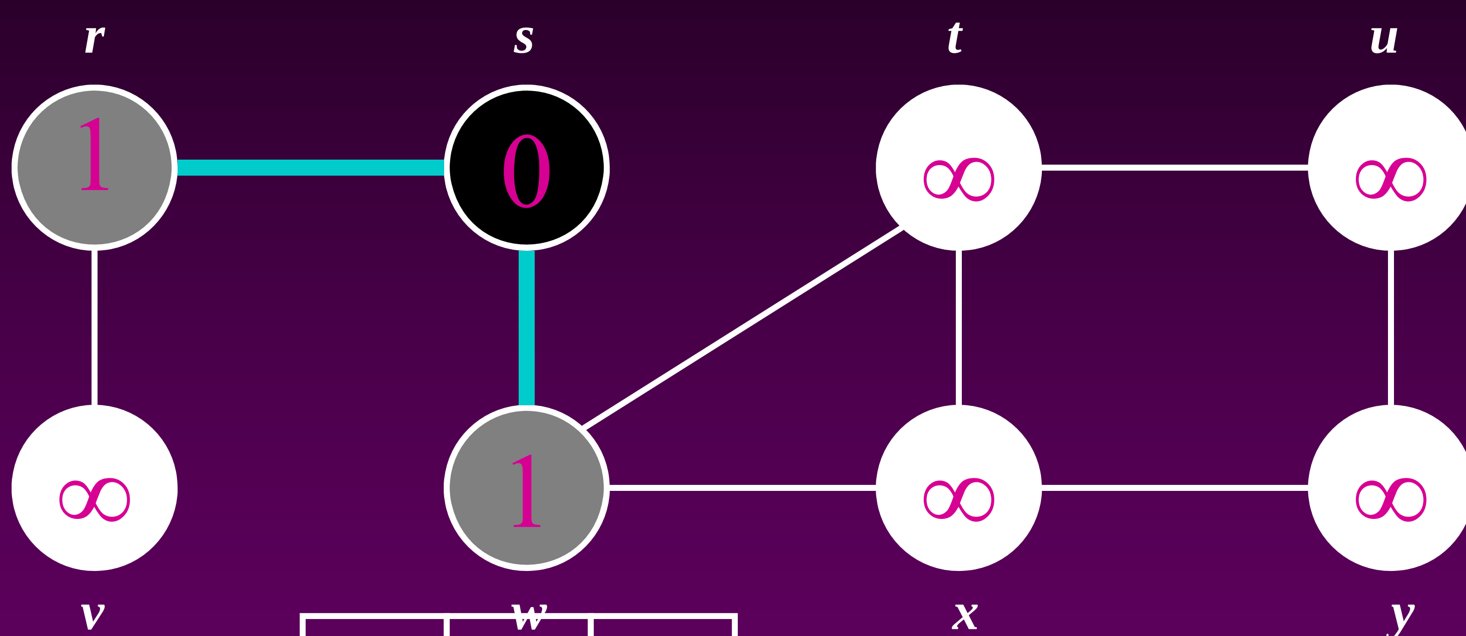
Breadth-First Search: Example



Q: s

vertex	r	s	t	u	v	w	x	y
Color	W	G	W	W	W	W	W	W
d	∞	0	∞	∞	∞	∞	∞	∞
prev	nil	nil	nil	nil	nil	nil	nil	nil

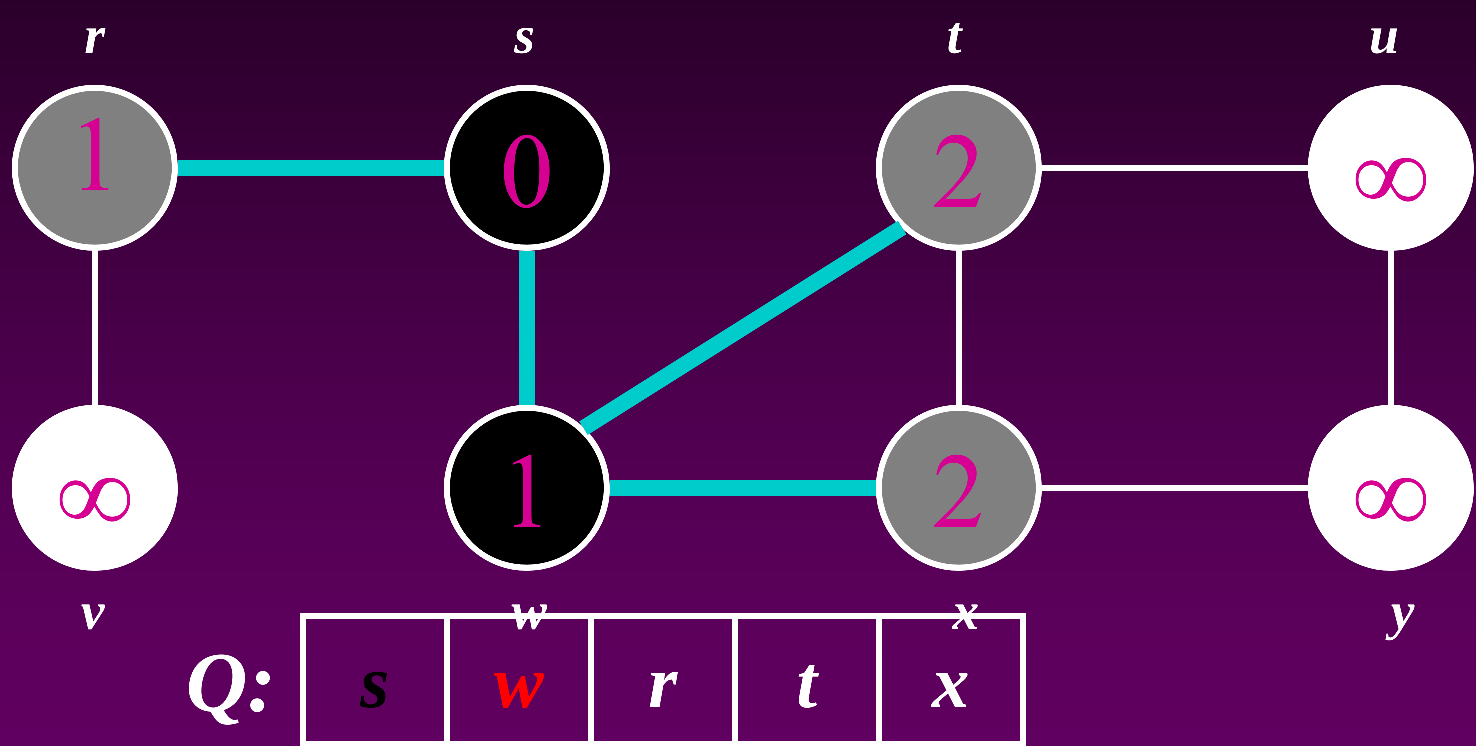
Breadth-First Search: Example



Q: | s | w | r |

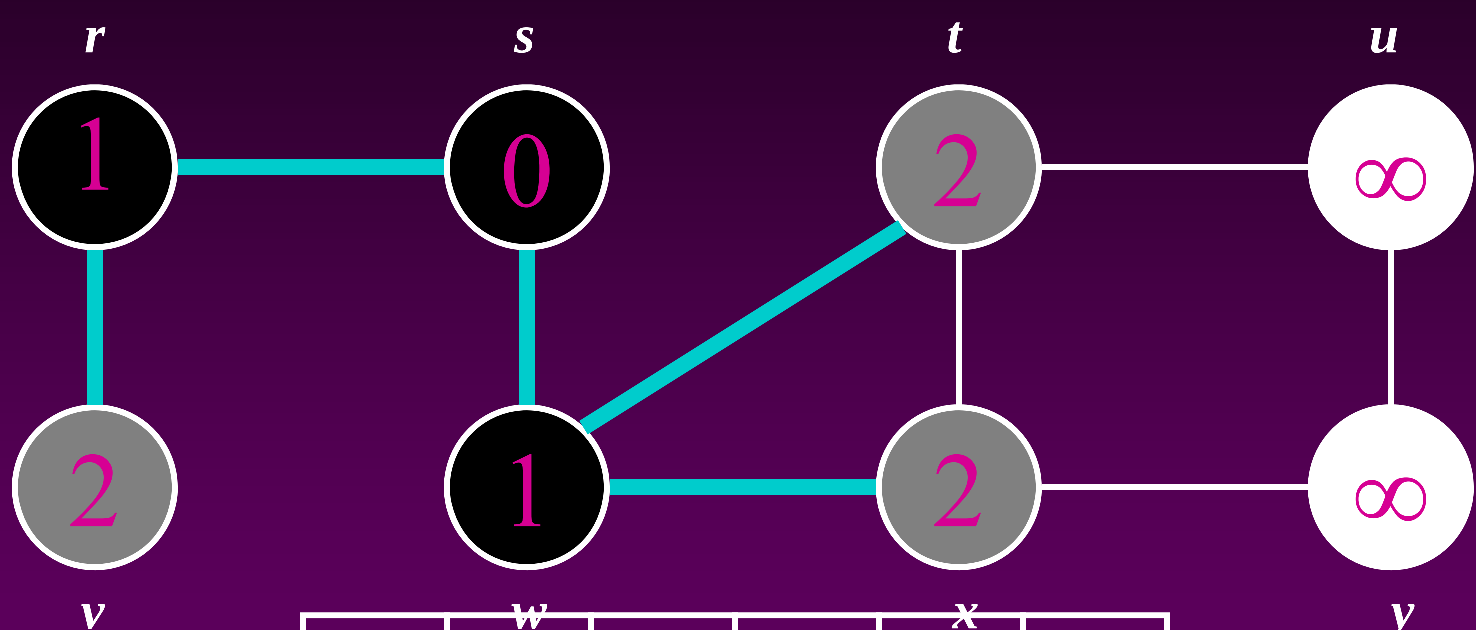
vertex	r	s	t	u	v	w	x	y
Color	G	B	W	W	W	G	W	W
d	1	0	∞	∞	∞	1	∞	∞
prev	s	nil	nil	nil	nil	s	nil	nil

Breadth-First Search: Example



vertex	r	s	t	u	V	w	X	y
Color	G	B	**G**	W	W	**B**	**G**	W
d	1	0	**2**	∞	∞	**1**	**2**	∞
prev	s	nil	**w**	nil	nil	**s**	**w**	nil

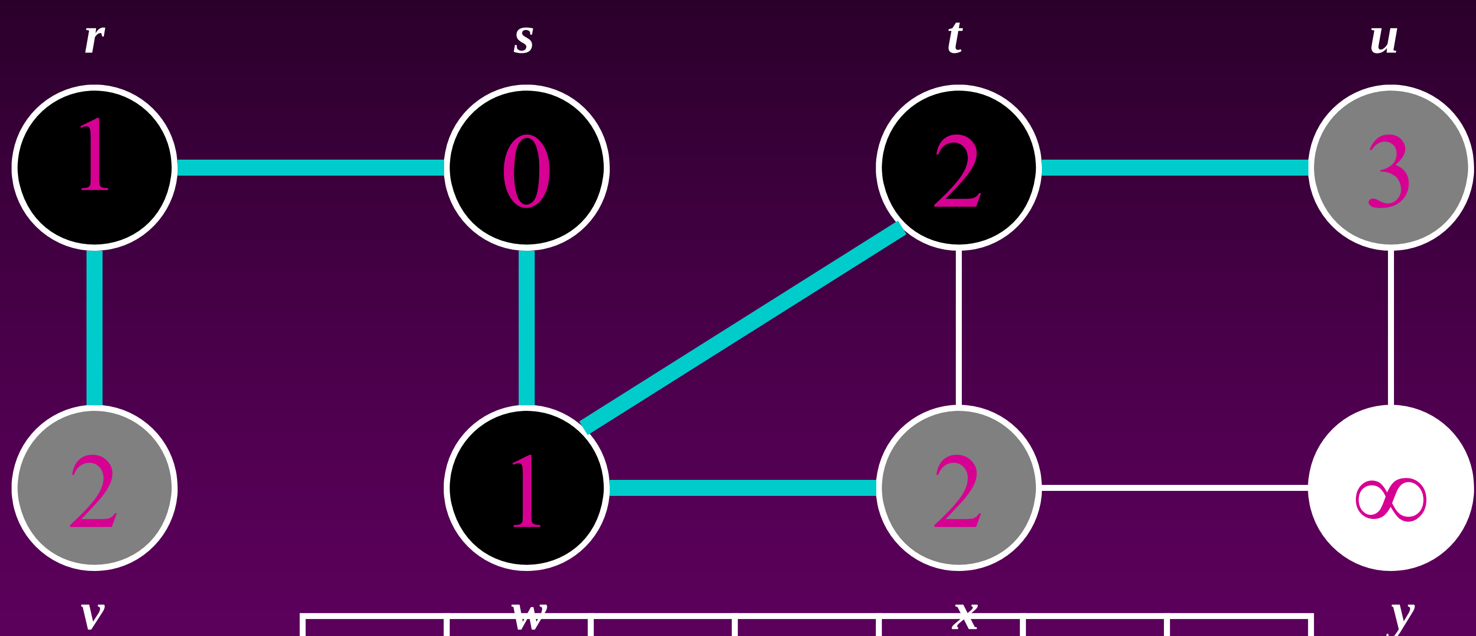
Breadth-First Search: Example



Q: | s | w | r | t | x | v |

vertex	r	s	t	u	v	w	x	y
Color	**B**	B	G	W	**G**	B	G	W
d	**1**	0	2	∞	**2**	1	2	∞
prev	**s**	nil	w	nil	**r**	s	w	nil

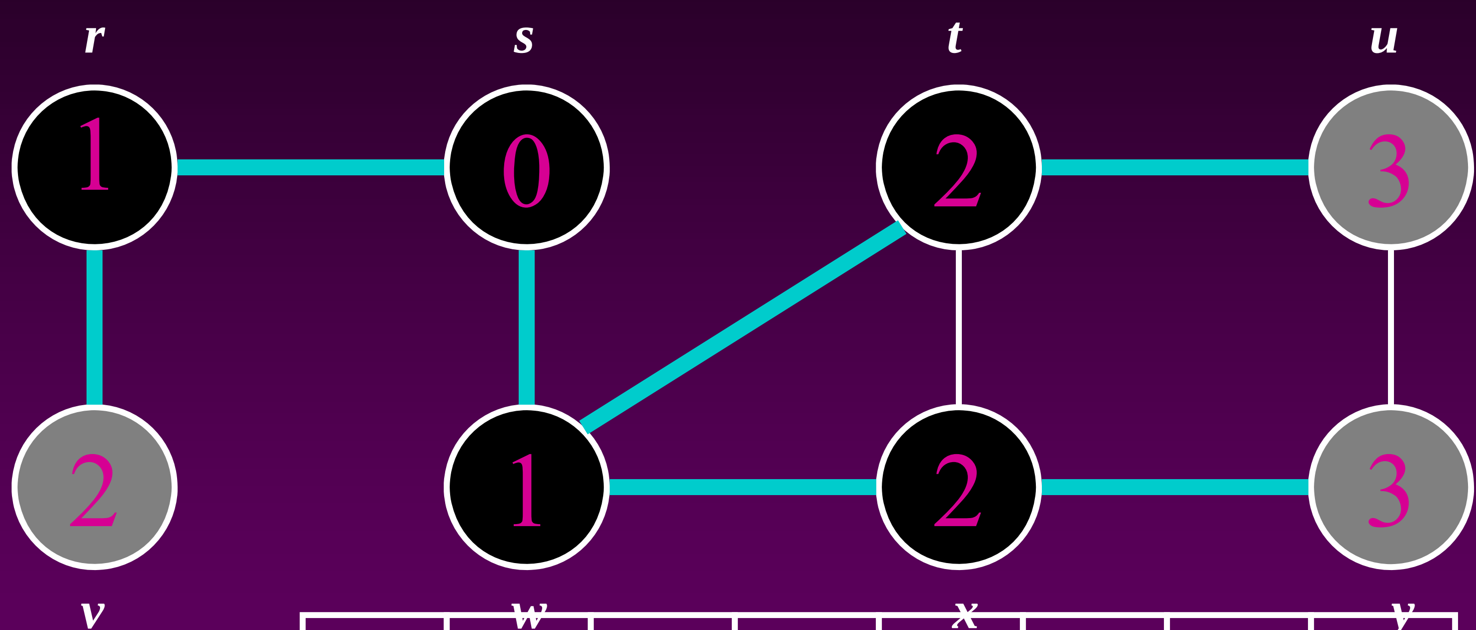
Breadth-First Search: Example



Q:	s	w	r	t	x	v	u

vertex	r	s	t	u	v	w	x	y
Color	B	B	**B**	**G**	G	B	G	W
d	1	0	**2**	**3**	2	1	2	∞
prev	s	nil	**w**	**t**	r	s	w	nil

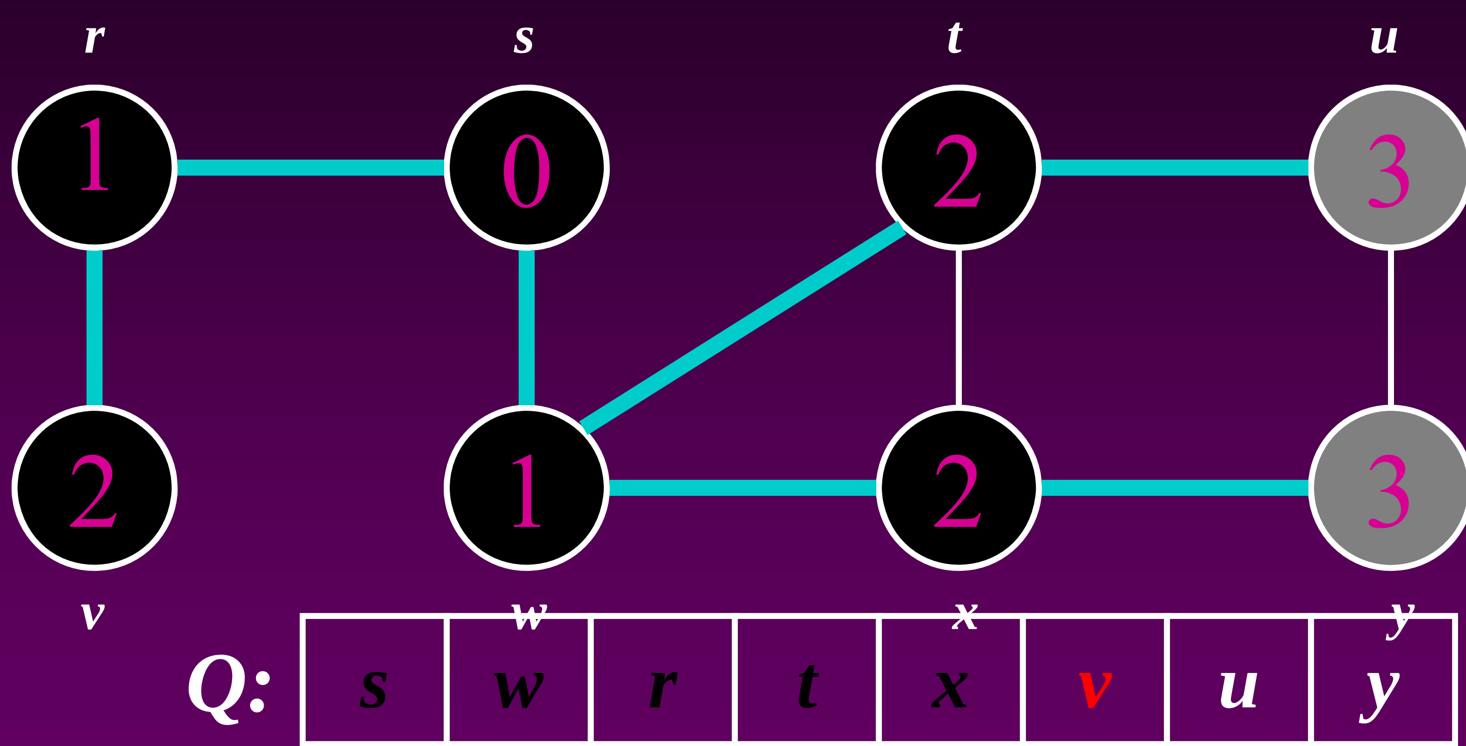
Breadth-First Search: Example



Q:	s	w	r	t	x	v	u	y

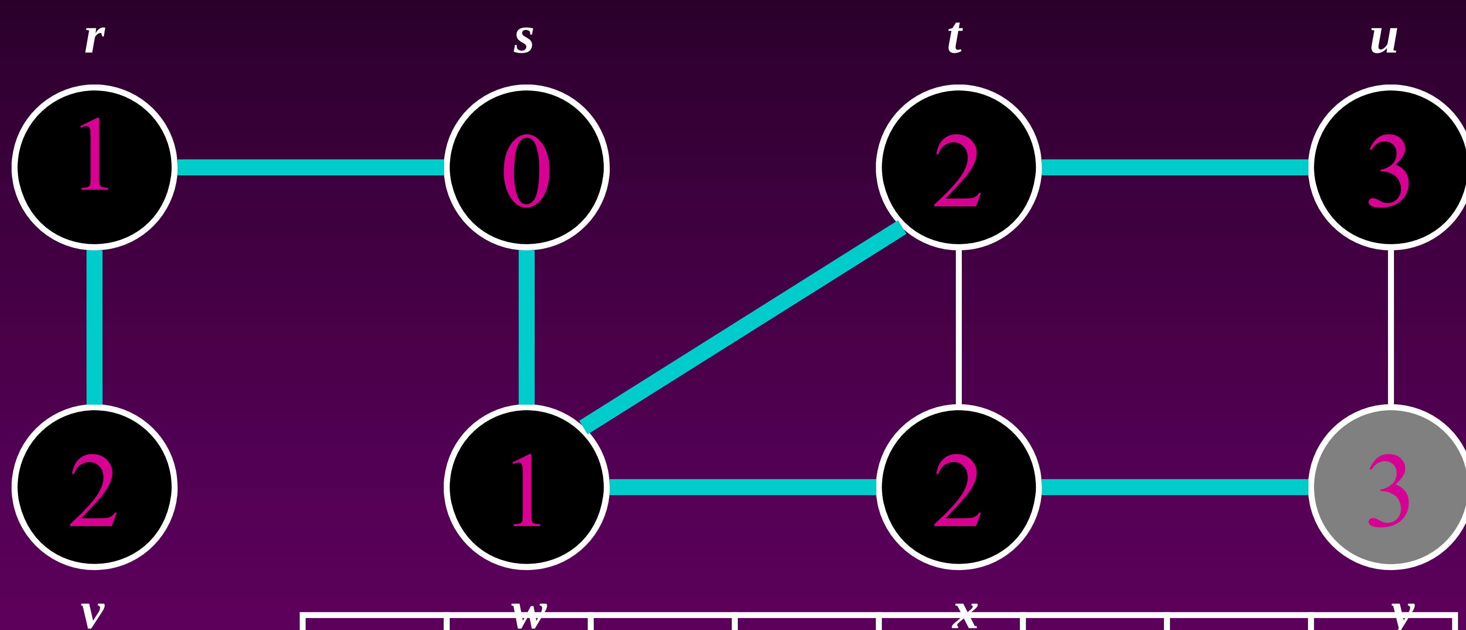
vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	G	B	B	G
d	1	0	2	3	2	1	2	3
prev	s	nil	w	t	r	s	w	x

Breadth-First Search: Example



vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	**B**	B	B	G
d	1	0	2	3	**2**	1	2	3
prev	s	nil	w	t	**r**	s	w	x

Breadth-First Search: Example

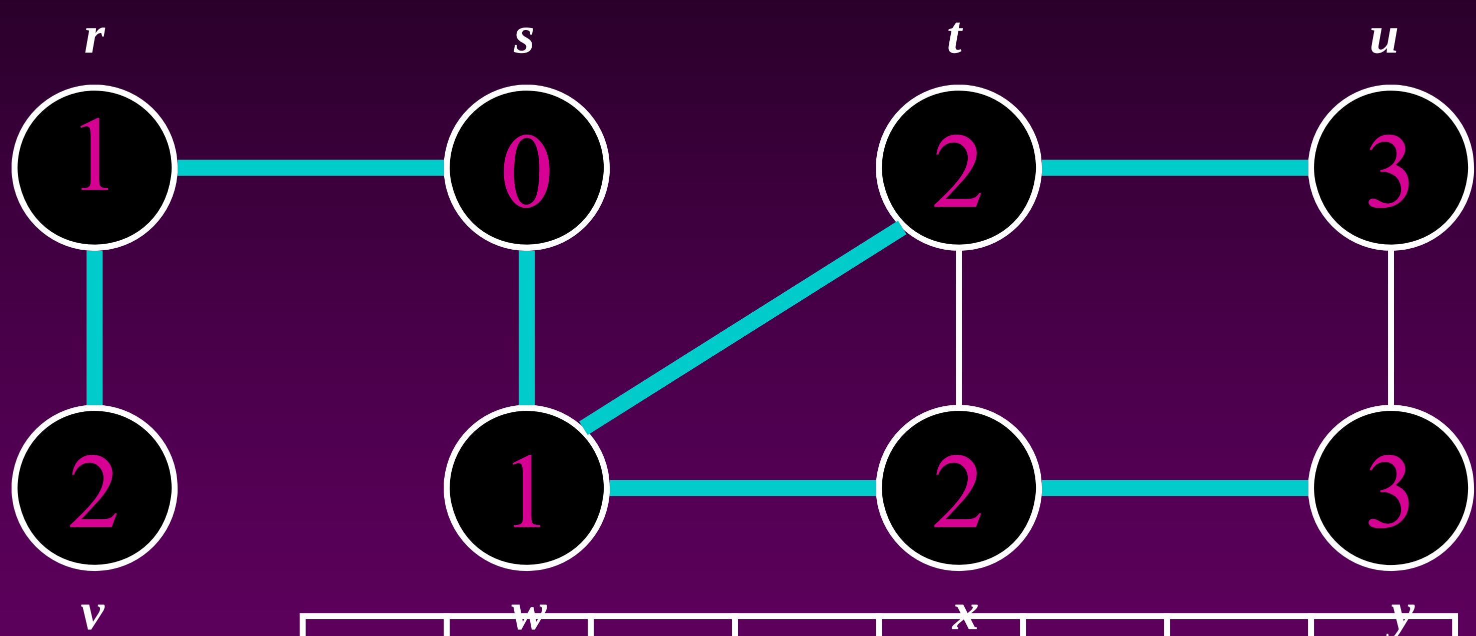


Q:

s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	**B**	B	B	B	G
d	1	0	2	**3**	2	1	2	3
prev	s	nil	w	**t**	r	s	w	x

Breadth-First Search: Example



Q: *s* *w* *r* *t* *x* *v* *u* *y*

vertex	r	s	t	u	v	w	x	y
Color	B	B	B	G	B	B	B	**B**
d	1	0	2	3	2	1	2	**3**
prev	s	nil	w	t	r	s	w	**x**

BFS: The Code (again)

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
   for each vertex u ∊ V-{s}
   {
      color[u]=WHITE;
       prev[u]=NIL;
       d[u]=inf;
   }
   color[s]=GRAY;
  d[s]=0; prev[s]=NIL;
  Q=empty;
  ENQUEUE(Q,s);
```

```
While(Q not empty)
{
  u = DEQUEUE(Q);
  for each v ∊ adj[u]{
    if (color[v] ==
 WHITE){
          color[v] = GREY;
          d[v] = d[u] + 1;
          prev[v] = u;
          Enqueue(Q, v);
    }
  }
  color[u] = BLACK;
}
}
```

Breadth-First Search: Print Path

```
Data: color[V], prev[V],d[V]

Print-Path(G, s, v)
{
   if(v==s)
       print(s)
    else if(prev[v]==NIL)
       print(No path);
   else{
       Print-Path(G,s,prev[v]);
       print(v);
   }
}
```

Thank You