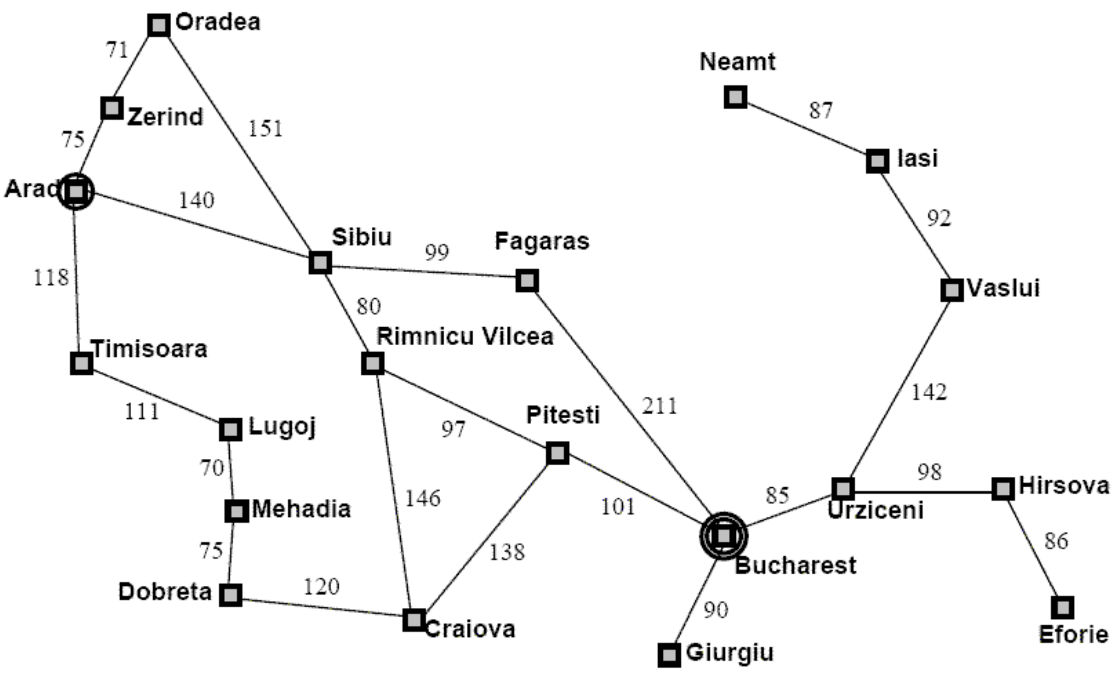
Breadth First Search

Search C455

Document last modified: 03/12/2014 21:45:24

Overview



There are two basic *uniformed* (i.e. have no a priori knowledge of solution) approaches for examining graphs: *depth first* or *breadth first*.

Breadth first search is preferred if looking for a (solution) vertex close to another vertex with some property.

However, some graphs are infinite or extremely deep (consider chess moves). Depth first search goes deep, getting to solution fast in these graphs, while breadth first search more slowly goes one level at a time until finding a solution.

For example, suppose we want to find the path from one city to another, *Arad* to *Fagaras*, but had no prior information as to which direction to begin.

Breadth first would travel to all cities *adjacent* to Arad and continue visiting *adjacent* cities until Fagaras was reached. We can observe that breadth first will always find a path having the *fewest* edges, assuming we prevent cycles.

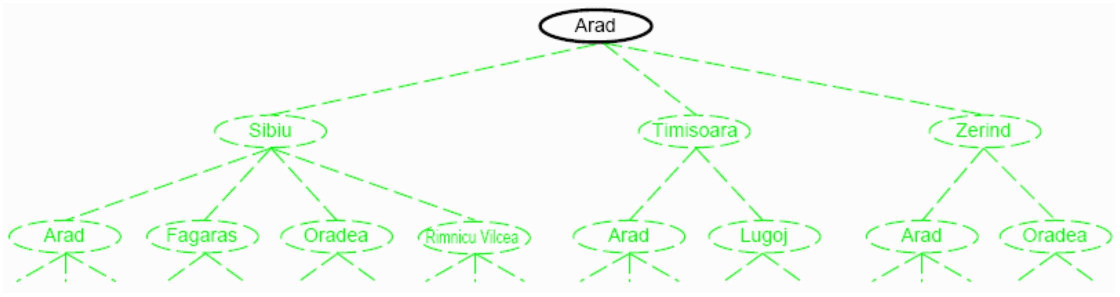
What we've just said is, for finite graphs, breadth first is *optimal* for edges in the path and *complete*, meaning it will find a path if one exists from the starting point.

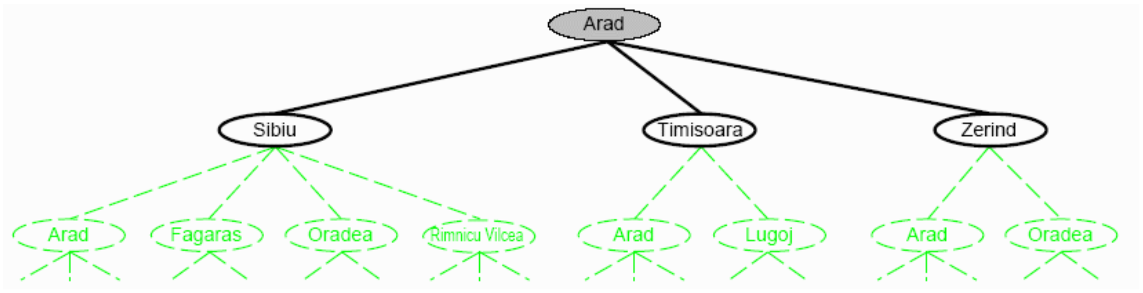
Depth first search might leave out of Arad and travel, by taking right turns first through Timisoara, Bucharest, Giurgiu and Eforie before reaching Fagaras.

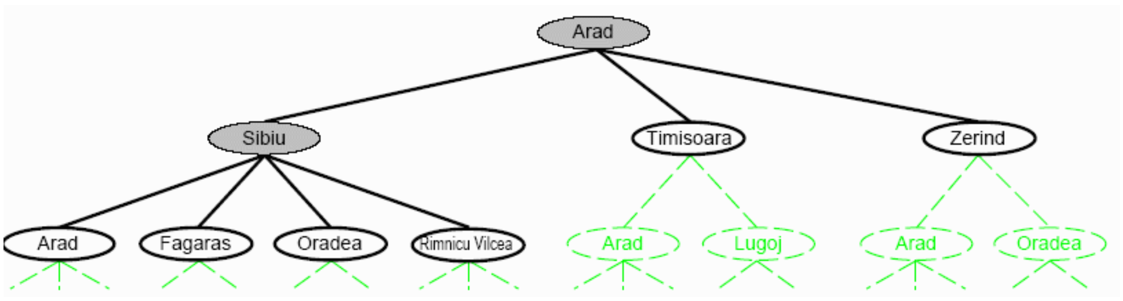
We can observe that depth first is not optimal since paths exist with fewer edges but it is *complete* for finite graphs (if cycles are avoided).

Question 22.6 - Would breadth first search, starting at Arad, reach Neamt on the way to Fagaras? What about depth first?

Breadth first search for Fagaras and beyond







For another example, imagine a chess-playing program.

You can think of each possible configuration of the chessboard as a vertex in a graph. The program would do a search in the graph, where moving a particular piece is an edge to another vertex. The program is given, say, 30 seconds to do the search.

Using depth first search would allow the program to explore many moves ahead for one particular initial move, but wouldn't allow it to explore any other moves.

Breadth first search would allow a variety of moves to be explored, up to several moves ahead to some depth of the search graph but seldom exploring an initial move to its conclusion.

Depth first search is a good way of establishing the nature of the connectivity in a graph. You can build a table out of the results of a search showing which vertices are reachable from which other vertices. Depth first search is also used in determining the *strongly connected components* of a directed graph. These are all the sub-graphs in which vertices are mutually reachable.

Breadth First Search

Basic idea of this algorithm is, starting at some source vertex, determine shortest path to all reachable vertices, which forms a tree.

Input: Graph G = (V, E), either directed or undirected, and source vertex $s \in V$.

Output:

v.d = distance (smallest # of *edges*) from s to v, for all $v \in V$.

v.Π = *u* such that (u, v) is last edge on shortest path $s \rightsquigarrow v$;

Π is the *predecessor* subgraph of G produced by BFS formally defined as $G_\Pi=(V_\Pi, E_\Pi)$ where:

- u is v's predecessor
- set of vertices $V_\Pi = \{ v \in V : v.\Pi \neq \text{NIL}\}$ **|** $\{s\}$
- set of edges $E_\Pi = \{(v.\Pi, v) : v \in V_\Pi - \{s\} \}$ forms a tree.

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple and consider all edges equal.

Idea: Send a wave out from some root vertex, s.

- First hits all vertices 1 edge from s (those vertices directly connected to s).
- From there, hits all vertices 2 edges from s.
- Etc.

  Use FIFO queue Q to maintain wave front.

  - $v \in Q$ if and only if wave has hit *v* but has not come out of v yet.
- Mark each vertex when visited to avoid cycles.

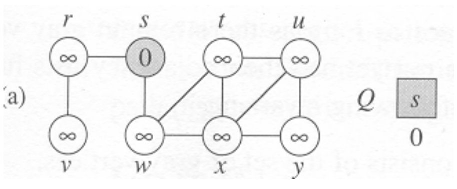
Example - Construct tree from undirected graph by constructing Π list during BFS.

V = {r, s, t, u, v, w, x, y}

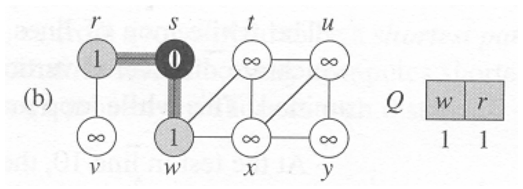
E = {(r,v), (r,s), (s,w), (w,x), (w,t), (t,x), (t,u), (x,u), (x,y), (u,y)}

Q = FIFO of vertices discovered but not visited

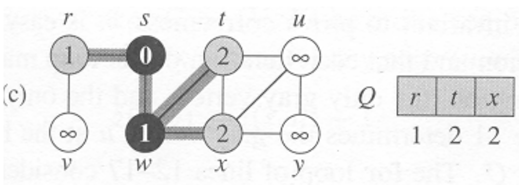
- WHITE - undiscovered
- GRAY - discovered and in queue waiting to be processed
- BLACK - discovered and visited



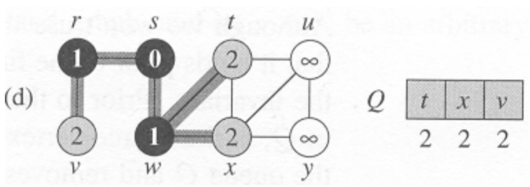
	r	s	t	u	v	w	x	y
Π	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL
d	∞	0	∞	∞	∞	∞	∞	∞



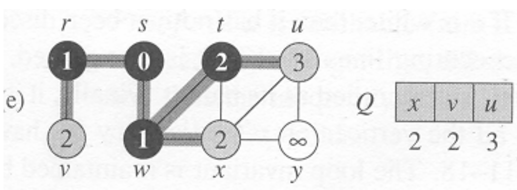
	r	s	t	u	v	w	x	y
Π	s	NIL	NIL	NIL	NIL	s	NIL	NIL
d	1	0	∞	∞	∞	1	∞	∞



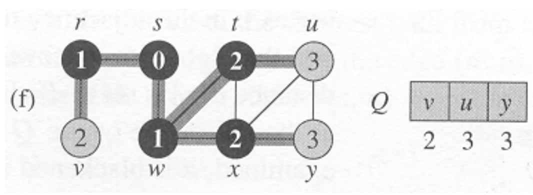
	r	s	t	u	v	w	x	y
Π	s	NIL	w	NIL	NIL	s	w	NIL
d	1	0	2	∞	∞	1	2	∞



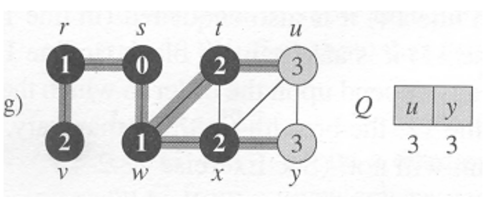
	r	s	t	u	v	w	x	y
Π	s	NIL	w	NIL	r	s	w	NIL
d	1	0	2	∞	2	1	2	∞



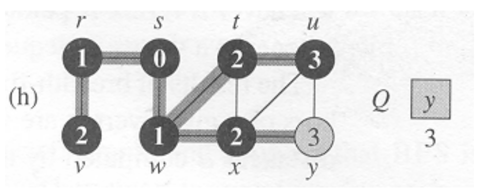
	r	s	t	u	v	w	x	y
Π	s	NIL	w	t	r	s	w	NIL
d	1	0	2	3	2	1	2	∞



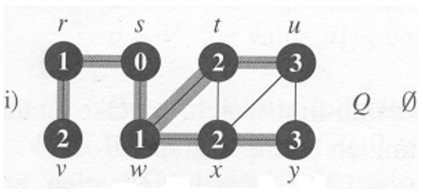
	r	s	t	u	v	w	x	y
Π	s	NIL	w	t	r	s	w	x
d	1	0	2	3	2	1	2	3



	r	s	t	u	v	w	x	y
Π	s	NIL	w	t	r	s	w	x
d	1	0	2	3	2	1	2	3



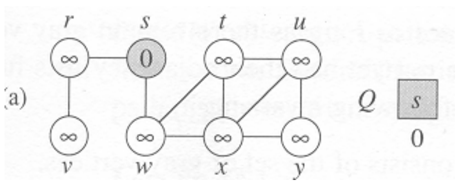
	r	s	t	u	v	w	x	y
Π	s	NIL	w	t	r	s	w	x
d	1	0	2	3	2	1	2	3



	r	s	t	u	v	w	x	y
Π	s	NIL	w	t	r	s	w	x
d	1	0	2	3	2	1	2	3

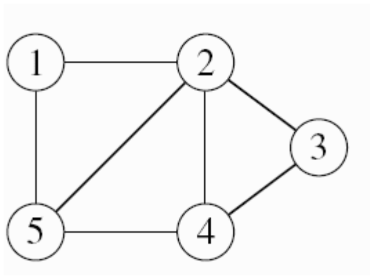
Question 22.7 After executing BFS,

1. What does x.d contain?
2. What does x.Π contain?
3. What is the maximum distance to source *s*?
4. Trace the path starting at y.Π to the source *s*.
5. Draw the tree rooted at *s*. Result is a spanning tree.



	r	s	t	u	v	w	x	y
Π	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL
d	∞	0	∞	∞	∞	∞	∞	∞

```
BFS (G, s)
        -- post: every vertex that is reachable from s
        -- is discovered and the BFS tree is created  in Π
                                                         Times      Cost
1       for each vertex u ∈ G.V - {s} do                 |V|        O(V)
2          u.color ← WHITE                               |V|-1
3          u.d ← ∞                                       |V|-1
4          u.Π ← NIL                                     |V|-1
        -- Initialize the source vertex and Queue
5       s.color ← GRAY                                   1
6       s.d ← 0                                          1
7       s.Π ← NIL                                        1
8       Q ← Ø                                            1
9       Enqueue (Q, s)                                   1
        -- Perform breadth first search
10      while Q ≠ Ø do                                   |V|+1      O(V)
11         u ← Dequeue (Q)                               |V|
12         for each v ∈ G.Adj[ u ] do                    |V|+|E|+1  O(V+E)
13            if v.color = WHITE then                    |V|+|E|
14               v.color ← GRAY                          |V|
15               v.d ← u.d + 1                           |V|
16               v.Π ← u                                 |V|
17                Enqueue (Q, v)                         |V|
18         u.color ← BLACK                               |V|
```

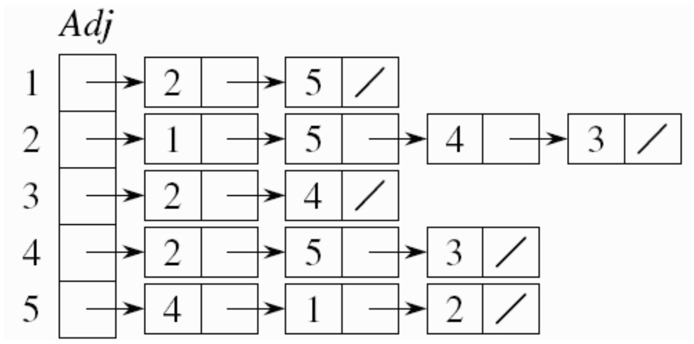


	1	2	3	4	5
Π	NIL	NIL	NIL	NIL	NIL
d	∞	∞	0	∞	∞
color	WHITE	WHITE	GRAY	WHITE	WHITE

Q = 3

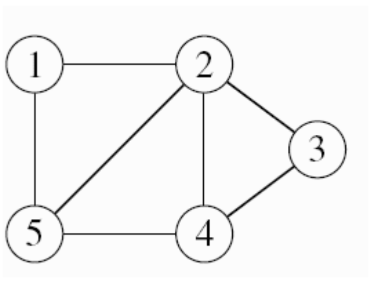
Question 22.8

- Execute the BFS on source vertex 3, visiting adjacent vertices at line 12 in numerical order.
- Count the number of times Line 12 is executed. Did it agree with |V|+|E|+1?
- Draw the resulting BFS tree rooted at vertex 3.



```
BFS (G, s)
                                                 Times      Cost
1       for each vertex u ∈ G.V - {s} do         |V|        O(V)
2          u.color ← WHITE                       |V|-1
3          u.d ← ∞                               |V|-1
4          u.Π ← NIL                             |V|-1
        -- Initialize the source vertex and Queue
5       s.color ← GRAY                           1
6       s.d ← 0                                  1
7       s.Π ← NIL                                1
8       Q ← Ø                                    1
9       Enqueue (Q, s)                           1
        -- Perform breadth first search
10      while Q ≠ Ø do                           |V|+1      O(V)
11         u ← Dequeue (Q)                       |V|
12      for each v ∈ G.Adj[u] do              |V|+|E|+1 O(V+E)
13         if v.color = WHITE then            |V|+|E|
14            v.color ← GRAY                  |V|
15            v.d ← u.d + 1                   |V|
16            v.Π ← u                         |V|
17             Enqueue (Q, v)                 |V|
18      u.color ← BLACK                       |V|
```

Loop-invariant - while loop of Lines 10-18



Line 10, the queue Q consists of the set of GRAY vertices (i.e. those that have been discovered but not visited)

loop invariant:

Π - all discovered vertices (except the source) are assigned their predecessor

d - all discovered vertices are assigned the distance from the source

initialization: prior to first iteration, Q = {s}, s.color = GRAY, s.d = 0 and s.Π = NIL

Π - the source predecessor is assigned NIL

d - the distance from the source to the source is assigned 0

```
BFS (G, s)
6 s.d ← 0
7 s.Π ← NIL
```

maintenance:

Line 11 removes one GRAY vertex *u* from Q which will be colored BLACK (i.e. discovered and visited).

u ← Dequeue (*Q*)

Lines 12-17, any WHITE (i.e. undiscovered) vertex *v* reachable from *u* is colored GRAY and inserted at the tail of Q.

Q consists of the set of GRAY vertices (i.e. those that have been discovered but not visited)

v.d ← u.d+1 The distance from source to a discovered (GRAY) vertex is assigned one greater than the predecessor

v.Π ← *u* The predecessor is assigned to the discovered (GRAY) vertex

Line 18 colors the current vertex BLACK as visited

```
BFS (G, s)
10 while Q ≠ ∅ do
11    u ← Dequeue (Q)
12    for each v ∈ G.Adj[u] do
13       if v.color = WHITE then
14          v.color ← GRAY
15          v.d ← u.d + 1
16          v.Π ← u
17           Enqueue (Q, v)
18    u.color ← BLACK
```

termination:

loop terminates when Q **=** ∅, no GRAY (discovered but not visited) vertices remain.

all vertices reachable from a GRAY vertex have been visited (colored BLACK).

by loop maintenance, all reachable vertices distance and predecessor values have been correctly assigned.

Analysis - BFS is O(V+E) due to line 12.

- V+E because each V vertex is examined but only the E edges connecting to adjacent vertices:

  for each *v* ∈ *G.Adj*[*u*] **do**, not all the edges of G, just adjacent.

- O(V) because every vertex is enqueued at most once when WHITE, Line 17.
- O(E) because every vertex is dequeued at most once and we examine (u, v) only when u is dequeued.

  Therefore, every edge examined at most once if directed, at most twice if undirected.

Note the analysis assumes an adjacency list representation.

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 1 & 0 & 0 & 1 \end{array}$$

Question 22.9

a. Is every vertex visited for an undirected graph? a directed graph? See the graph at right.

b. Give an example that contradicts the statement "every edge is examined in a directed graph".

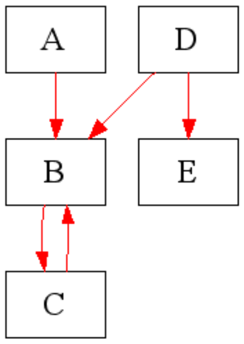
c. What is the purpose of the queue? Think about what happens in Lines 12-17.

d. What would be the running time for an adjacency *matrix* representation? Hint: Consider how edges are represented at right.

e. Run BFS on the graph at right starting at D.

f. Draw the resulting breadth-first tree(s).

g. We stated that BFS is O(V+E). From Question e, is it also Ω(V+E)?



```
BFS (G, s)
                                                  Times       Cost
1     for each vertex u ∈ G.V - {s} do            |V|         O(V)
2        u.color ← WHITE                          |V|-1
3        u.d ← ∞                                  |V|-1
4        u.Π ← NIL                                |V|-1
      -- Initialize the source vertex and Queue
5     s.color ← GRAY                              1
6     s.d ← 0                                     1
7     s.Π ← NIL                                   1
8     Q ← Ø                                       1
9     Enqueue (Q, s)                              1
      -- Perform breadth first search
10    while Q ≠ Ø do                              |V|+1       O(V)
11       u ← Dequeue (Q)                          |V|
12       for each v ∈ G.Adj[u] do                 |V|+|E|+1   O(V+E)
13          if v.color = WHITE then               |V|+|E|
14             v.color ← GRAY                     |V|
15             v.d ← u.d + 1                      |V|
16             v.Π ← u                            |V|
17              Enqueue (Q, v)                    |V|
18       u.color ← BLACK                          |V|
```

Shortest paths

- BFS finds the shortest path from source vertex to all other *reachable* vertices in graph. The text gives a proof of correctness for the BFS algorithm.

Question 22.10 - For the graph below.

a. Suppose we reran BFS with a different source vertex, would the results be the same?

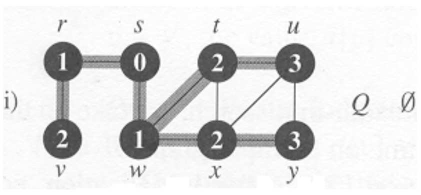
b. What is the distance to *s* from *y?*

c. What is the shortest path to *s* from y?

d. Give an undirected graph that has the shortest path length |V| (i.e. the path includes every vertex).

e. Give an algorithm to print the vertices in the shortest path from any connected vertex to the source.

f. Give an algorithm to print the vertices in the shortest path from the source to any connected vertex.



	r	s	t	u	v	w	x	y
Π	s	NIL	w	t	r	s	w	x
d	1	0	2	3	2	1	2	3