L06.01
Graph, Breadth-First-Search (BFS)

50.004 Introduction to Algorithms
Ioannis Panageas (ioannis@sutd.edu.sg)
CLRS Ch: 22.1-22.2
ISTD, SUTD
Slides based on Dr. Simon LUI

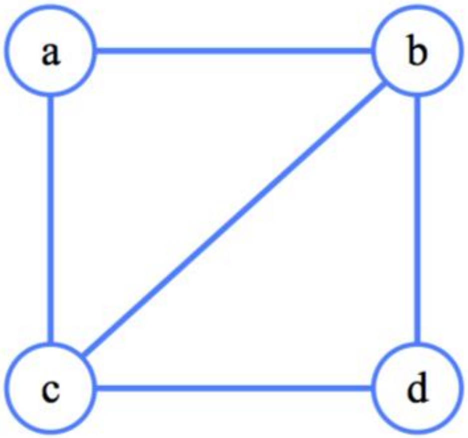
Overview

- What is graph
- 4 ways to represent a graph
- How to construct a tree from graph
  - BFS (today)
  - DFS (next lecture)

1. GRAPH

Graphs

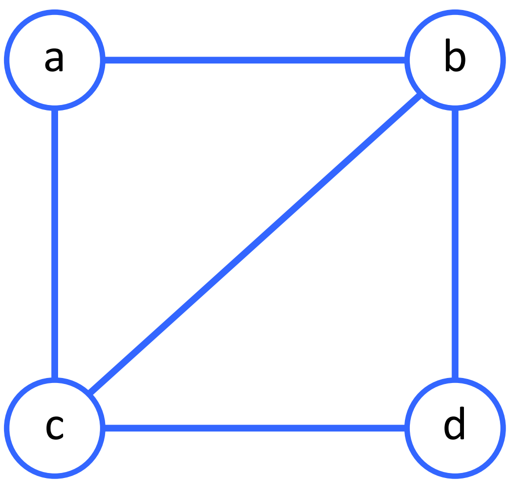
- $G=(V,E)$
- V a set of n vertices
- $E \subseteq V \times V$ a set of m edges (pairs of vertices)
- Two Flavors:
  - order of vertices on the edges matters: ***directed graphs***
  - ignore order: ***undirected graphs***



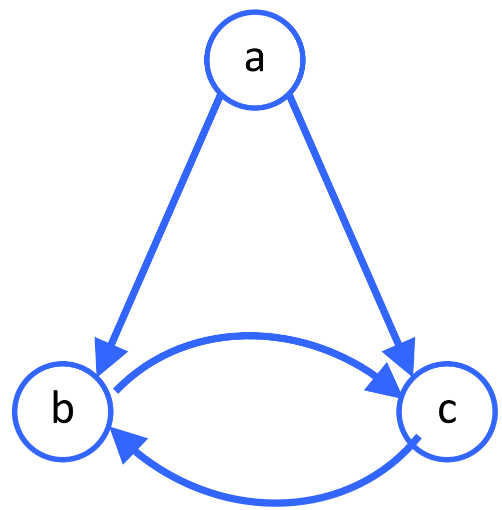
- Undirected
- $V=\{a,b,c,d\}$
- $E=\{\{a,b\}, \{a,c\}, \{b,c\}, \{b,d\}, \{c,d\}\}$

Examples

- Undirected
- V={a,b,c,d}
- E={{a,b}, {a,c}, {b,c}, {b,d}, {c,d}}



- Directed
- V = {a,b,c}
- E = {(a,c), (a,b) (b,c), (c,b)}

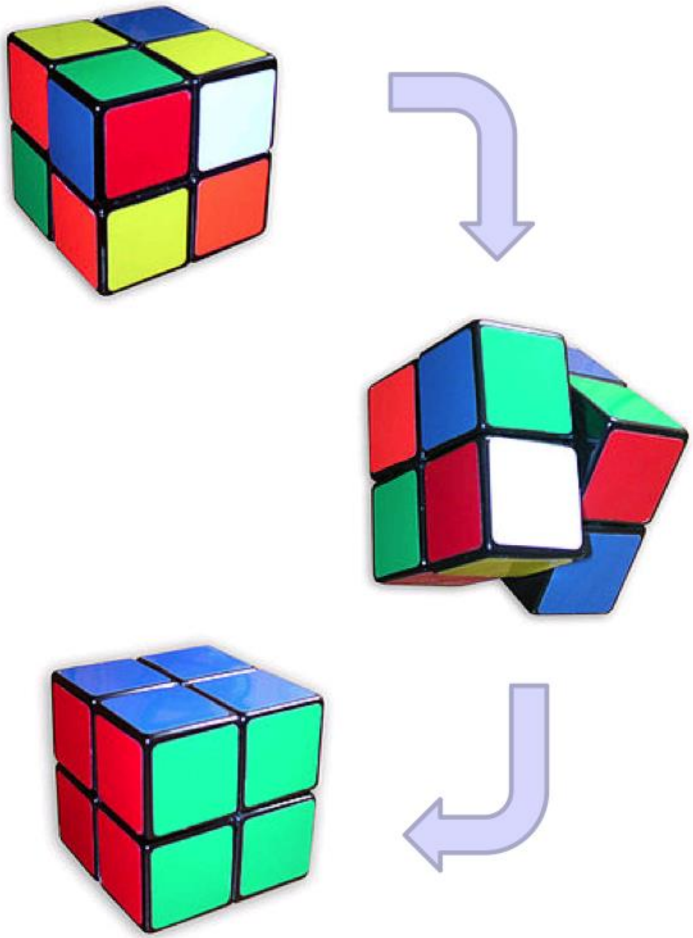


Instances/Applications

- Social Network
  - friend finder
- Computer Networks
  - internet routing
  - connectivity
- Game states
  - rubik's cube, chess
- Music
  - Auto accompaniment, chord progression

Let’s look at one example – Pocket Cube

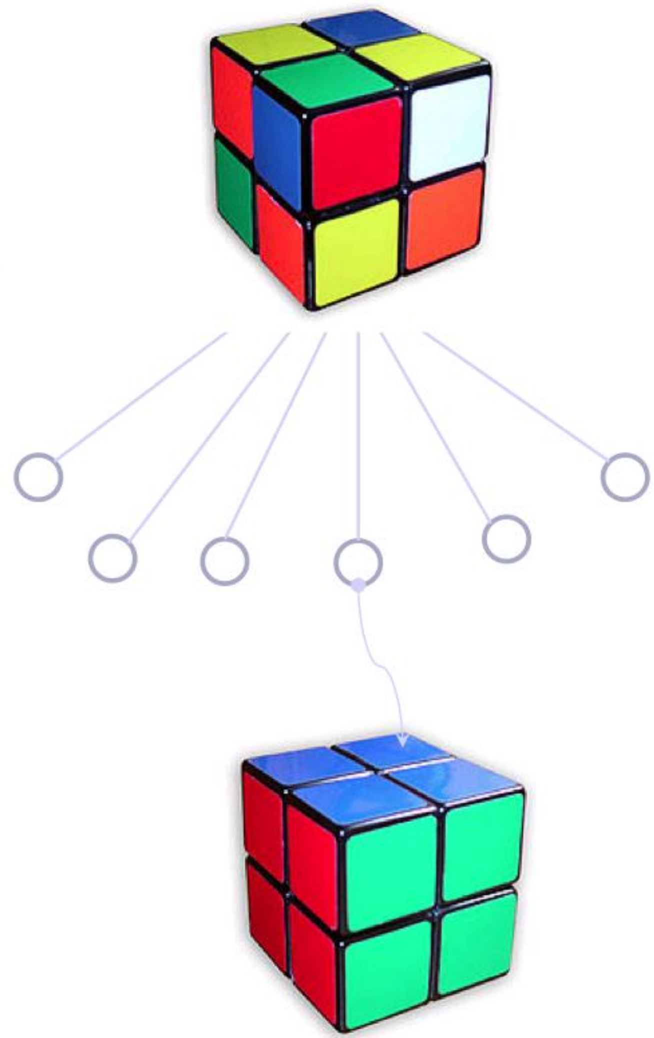
Pocket Cube



- $2 \times 2 \times 2$ Rubik's cube
- Start with a given configuration
- Moves are quarter turns of any face
- "Solve" by making each side one color

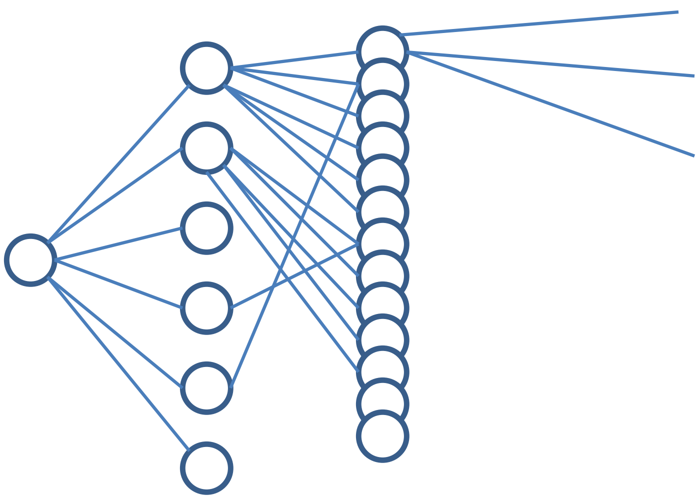
Configuration Graph

- Imagine a graph that has:
  - One vertex for each state of cube
  - One edge for each move from a vertex
    - 6 faces to twist
    - 3 nontrivial ways to twist (1/4, 2/4, 3/4)
    - So, 18 edges out of each state
- Solve cube by finding a path (of moves) from initial state (vertex) to “solved” state



State exploration

- One start vertex
- 6 others reachable by one 90° turn
- From those, 27 others by another
- And so on



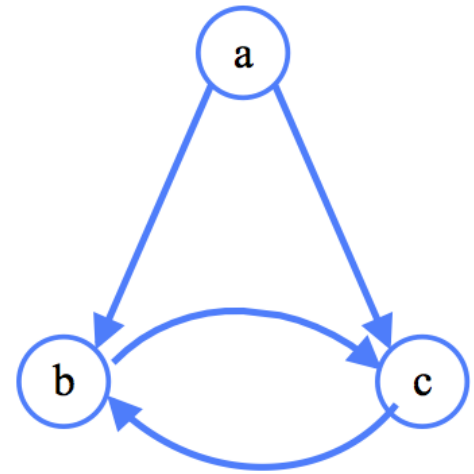
distance	90°	90° and 180°
0	1	1
1	6	9
2	27	54
3	120	321
4	534	1847
5	2,256	9,992
6	8,969	50,136
7	33,058	227,526
8	114,149	870,072
9	360,508	1,887,748
10	930,588	623,800
11	1,350,852	2,644
12	782,536	
13	90,280	
14	276	

Can be solved in at most 11 or 14 steps

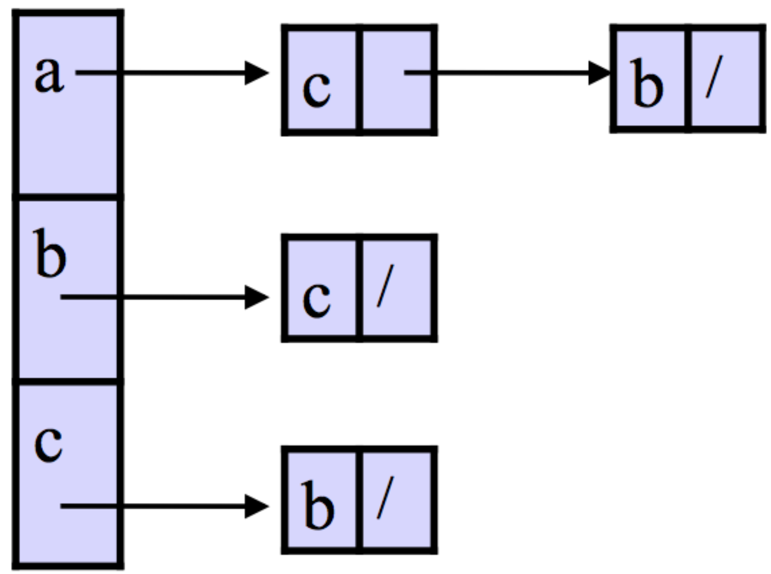
Four Representation of graph

- To solve graph problems, must examine graph
- So need to represent in computer
- Four representations with pros/cons
  1. Adjacency lists (of neighbors of each vertex)
  2. Incidence lists (of edges from each vertex)
  3. Adjacency matrix (of which pairs are adjacent)
  4. Implicit representation (as neighbor function)

Adjacency List

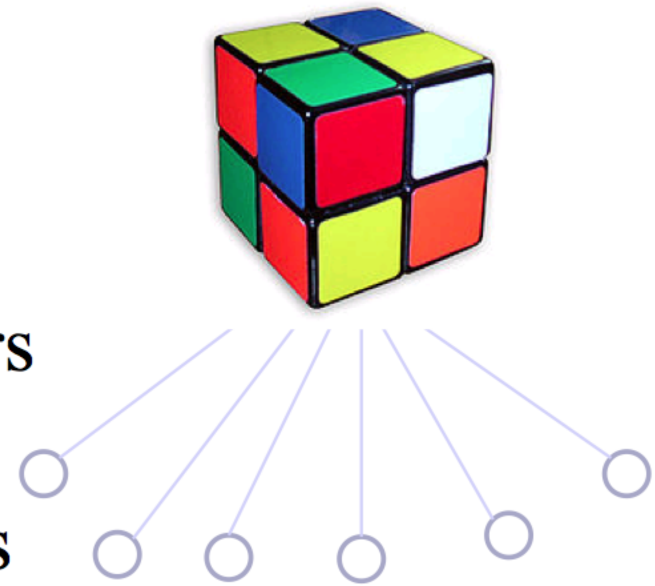


- For each vertex v, list its neighbors (vertices to which it is connected by an edge)
  - Array A of $|V|$ linked lists
  - For $v \in V$, list $A[v]$ stores neighbors $\{u \mid (v,u) \in E\}$
- Directed graph only stores outgoing neighbors
- Undirected graph stores edge in two places



Implicit representation

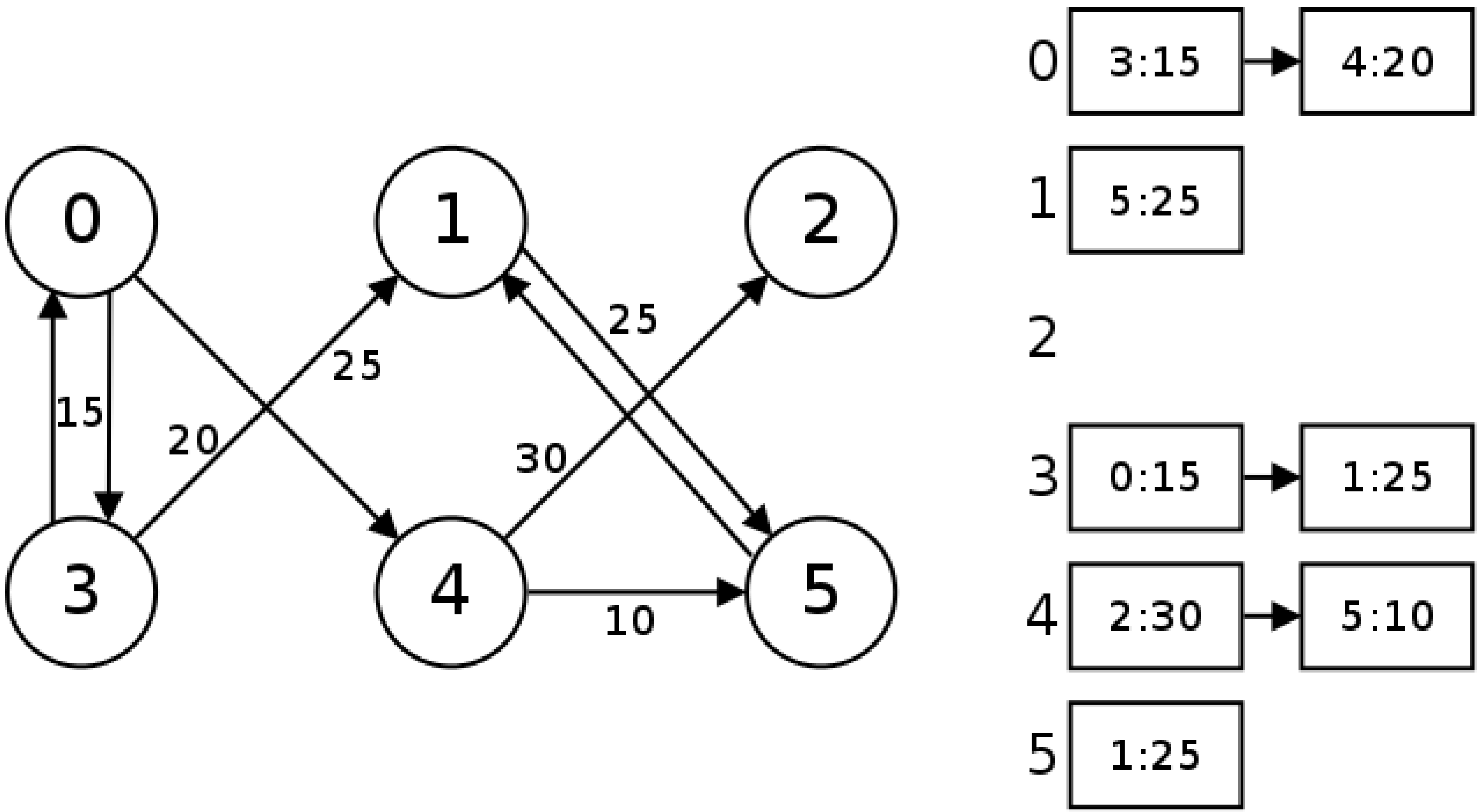
- Don't store graph at all
- Implement function Adj(u) that returns list of neighbors or edges of u
- Requires no space, use it as you need it
- And may be very efficient



Incidence List

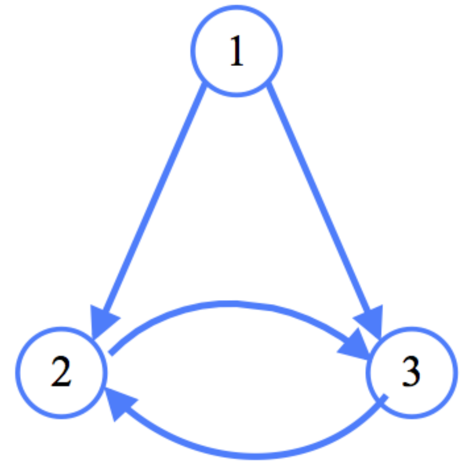
- For each vertex v, list its edges
  - Array A of $|V|$ linked lists
  - For $v \in V$, list $A[v]$ stores edges $\{e \mid e=(v,u) \in E\}$
  - Directed graph only stores outgoing edges
  - Undirected graph stores edge in two places
- In python, $A[v]$ can be hash table

Incidence list - example



Adjacency Matrix

- Assume $V=\{1, \ldots, n\}$
- $n \times n$ matrix $A=(a_{ij})$
  - $a_{ij} = 1$ if $(i,j) \in E$
  - $a_{ij} = 0$ otherwise
- (store as, e.g., array of arrays)



1	2	3	
0	1	1	1
0	0	1	2
0	1	0	3

Tradeoff: Space

- Assume vertices $\{1,\ldots,n\}$
- Adjacency lists:
  - One list node per edge
  - So space is $\Theta(n+m)$ bits
- Adjacency matrix:
  - Uses n^2 entries
  - But each entry can be just one bit
  - So $\Theta(n^2)$ bits
- Matrix better only for very dense graphs, i.e., m near n^2

Tradeoff: Time

- Add an edge
  - Both data structures are $O(1)$
- Check "is there an edge from u to v"?
  - Matrix is $O(1)$
  - Adjacency list of u must be scanned
- Visit all neighbors of u (very common)
  - adjacency list is $O(\text{neighbors})$
  - matrix is $\Theta(n)$
- Remove edge
  - like find + add

2. TREE

Tree

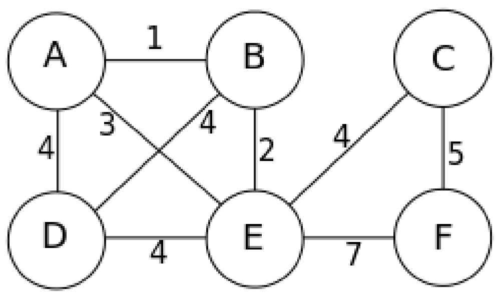
Tree

- a restricted form of a graph
- have a parent-child relationship
- Directed Acyclic Graph
- a single parent
- Trees don’t need directed edges: their parent-child relation is implicit from tree structure
- Many different trees can be constructed from a given graph

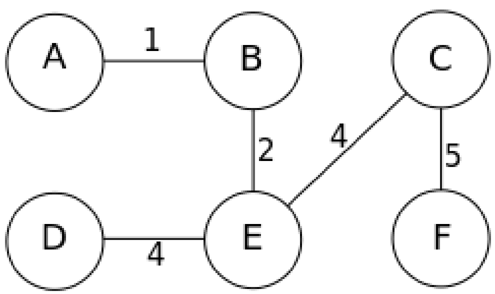
Forest

- set of trees

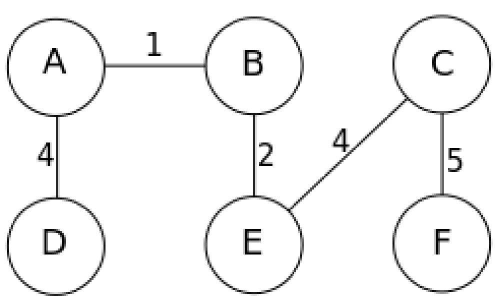
Examples of tree constructions



Graph G



Minimum spanning tree 1, constructed from G



Minimum spanning tree 2, constructed from G

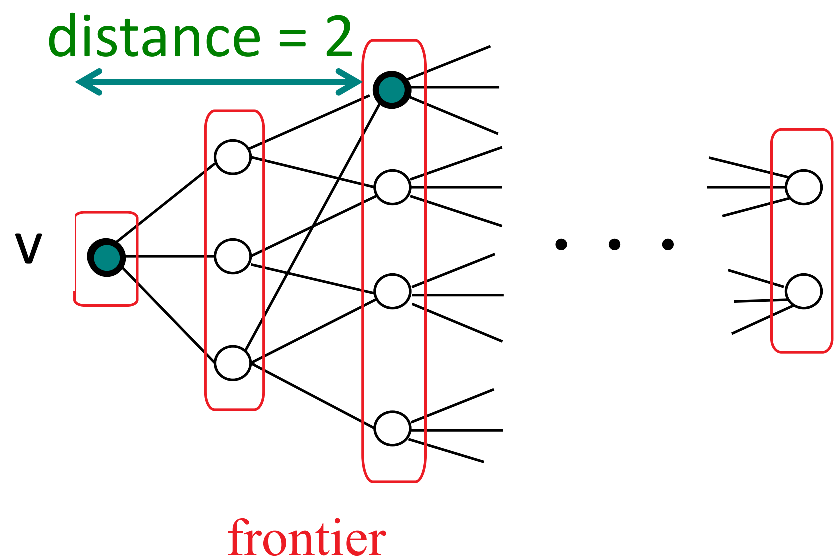
3. SEARCH A GRAPH TO GENERATE A TREE

Graph search

1. Breadth first search (BFS)
2. Depth first search (DFS)

Breadth First Search

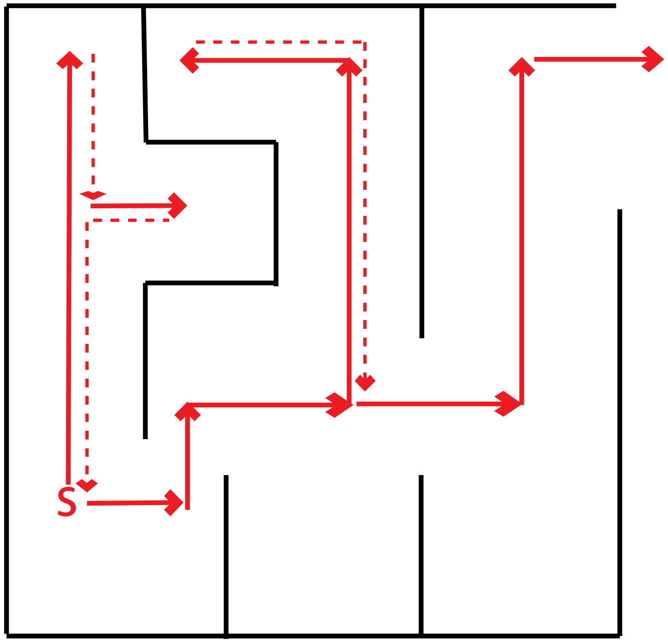
- start with vertex v
- list all its neighbors (distance 1 from v)
- then all their neighbors (distance 2 from v)
- etc.



- algorithm starting at v:
  - define frontier F
  - initially F={v}
  - repeat: F’= all new neighbors of vertices in F, F=F’
  - until all vertices are found

Depth First Search

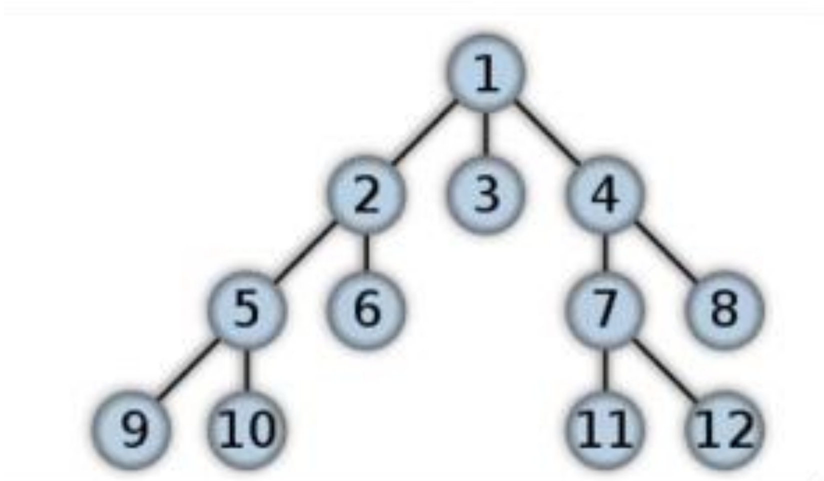
- Like exploring a maze
- From current vertex, move to another
- Until you get stuck
- Then backtrack till you find a new place to explore



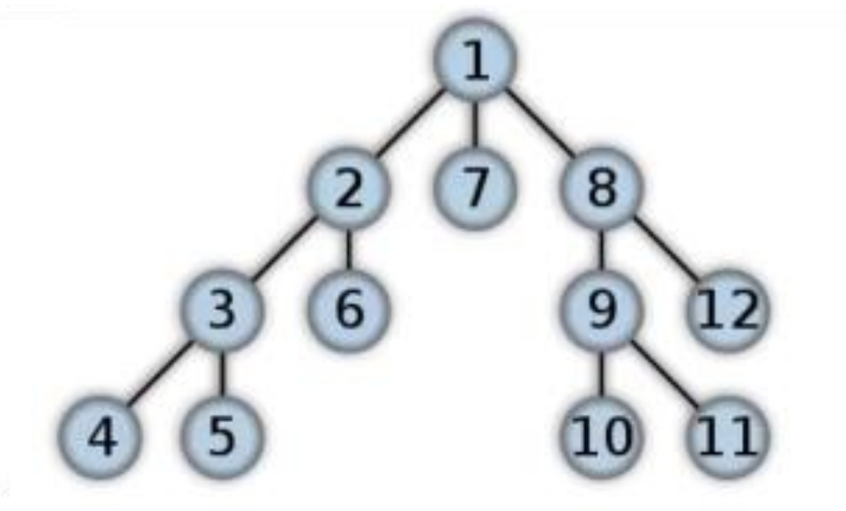
Example: traversal order of BFS vs DFS

(P.S. this is just a node traversal order, not constructing tree from graph)

BFS



DFS



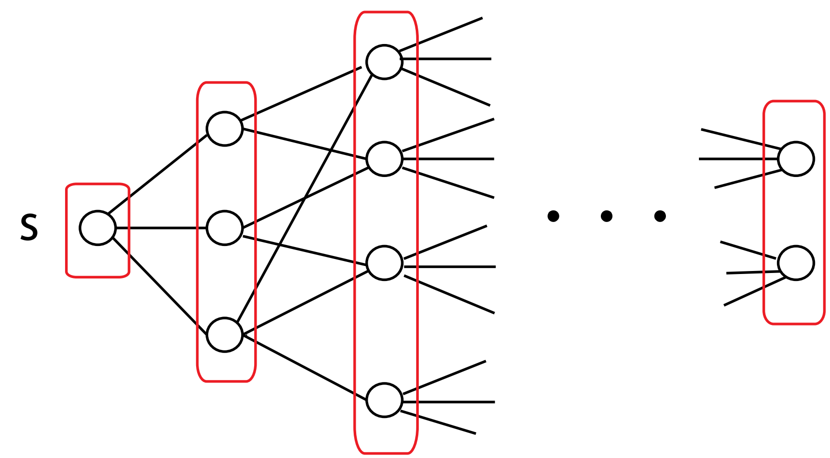
Problem: Cycles

- What happens if unknowingly revisit a vertex?
- BFS: get wrong notion of distance
- DFS: go in circles
- Solution: mark vertices
  - if you've seen it before, ignore

The BFS search algorithm (Array + adj list implementation)

```
BFS(s,Adj):
  level={s:0}
  parent={s:None}
  i=1
  frontier=[s]
  while frontier:
    next=[]
    for u in frontier
      for v in Adj[u]
        if v not in level
          level[v]=i
          parent[v]=u
          next.append(v)
    frontier=next
    i+=1
```

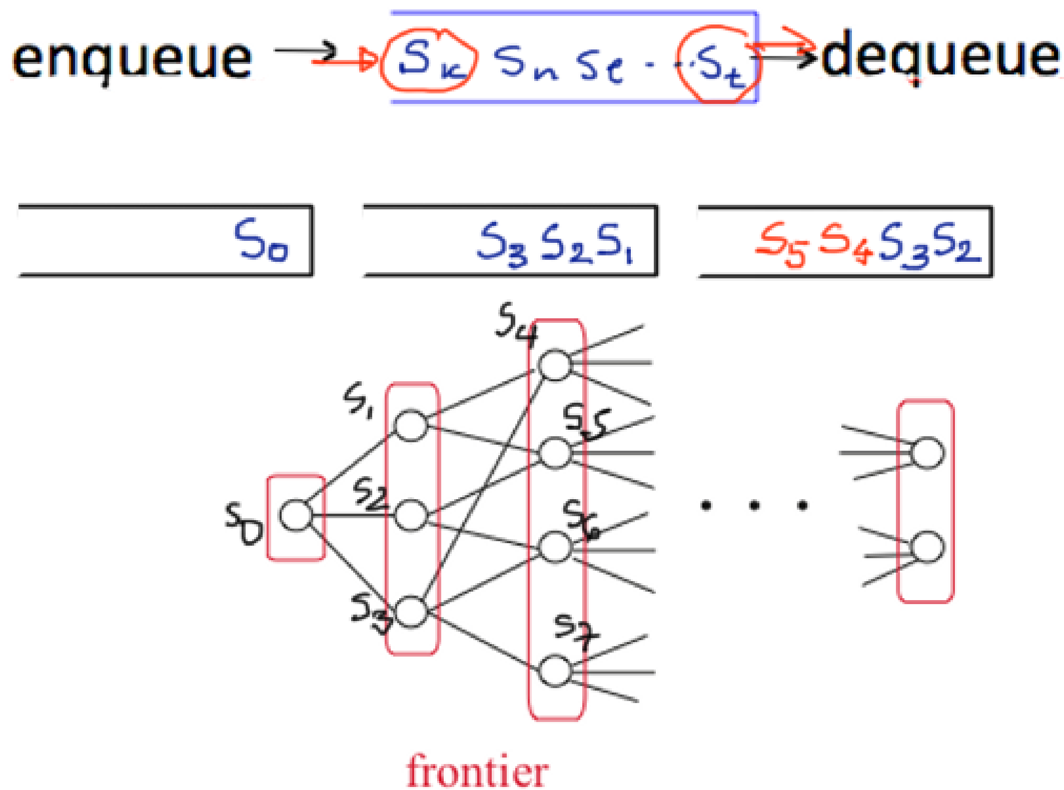
level: stores the nodes seen so far and their distance from the root
frontier: the last level of reachable nodes



frontier

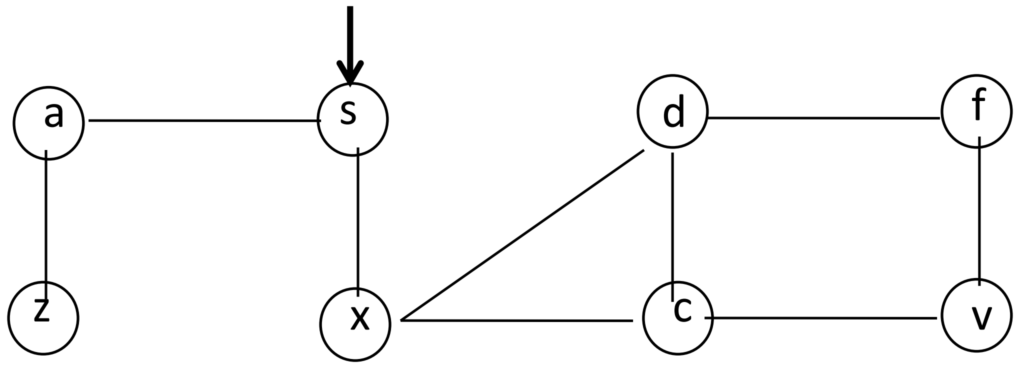
The BFS search <u>algorithm</u> (Queue + adj list implementation)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```



Replace queue by stack:
it become DFS

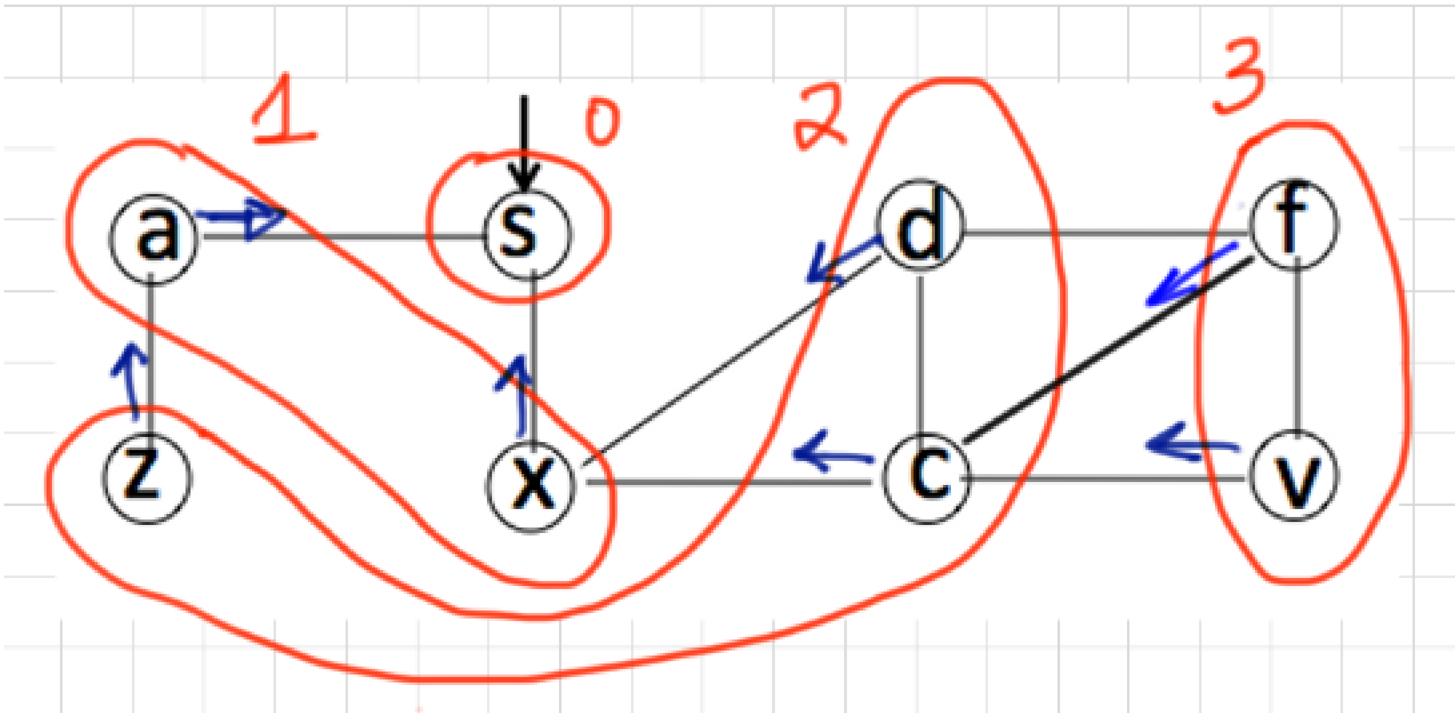
Example of BFS



Class exercise:

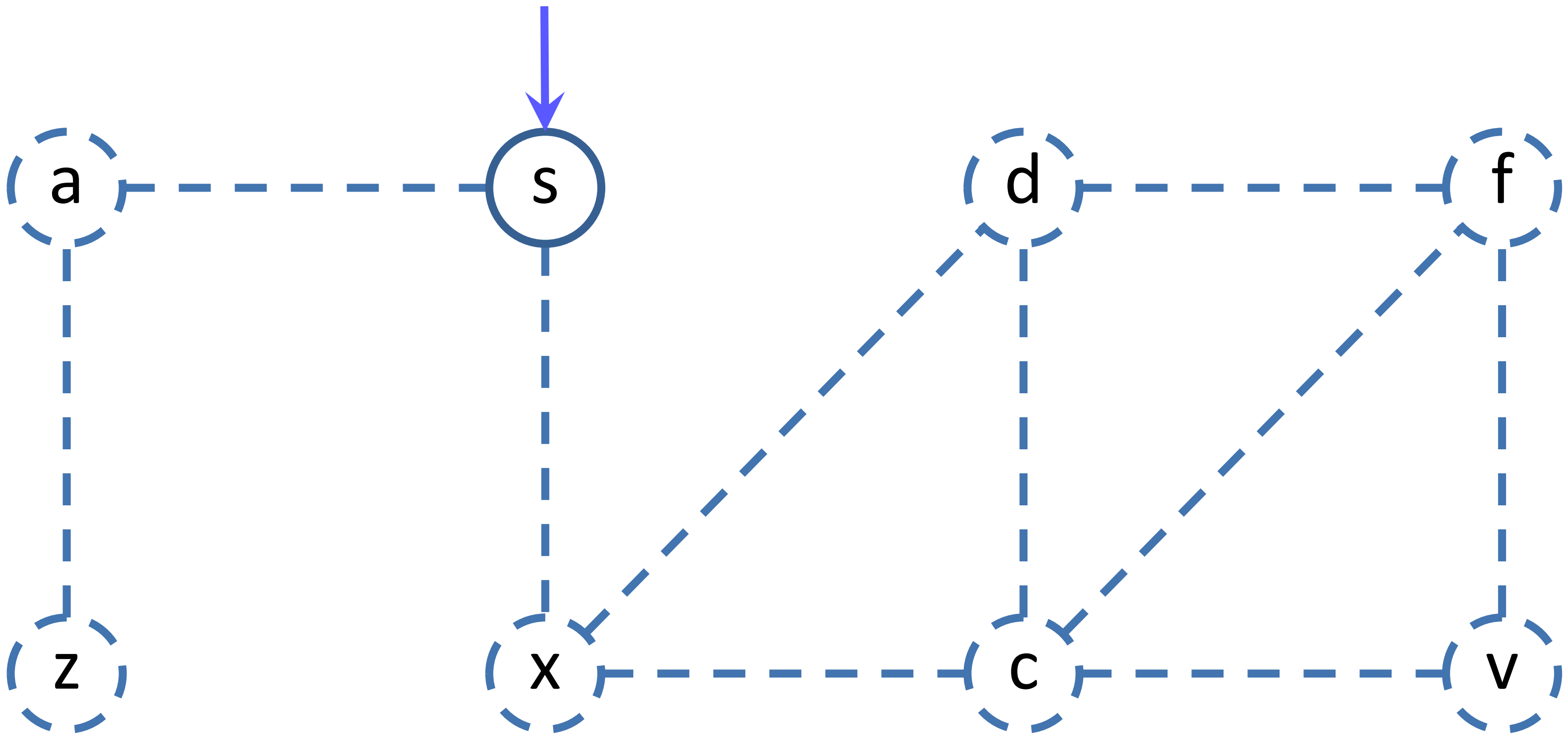
1) Build a BFS
2) Mark the level of each node (0,1,2,3)

Example of BFS

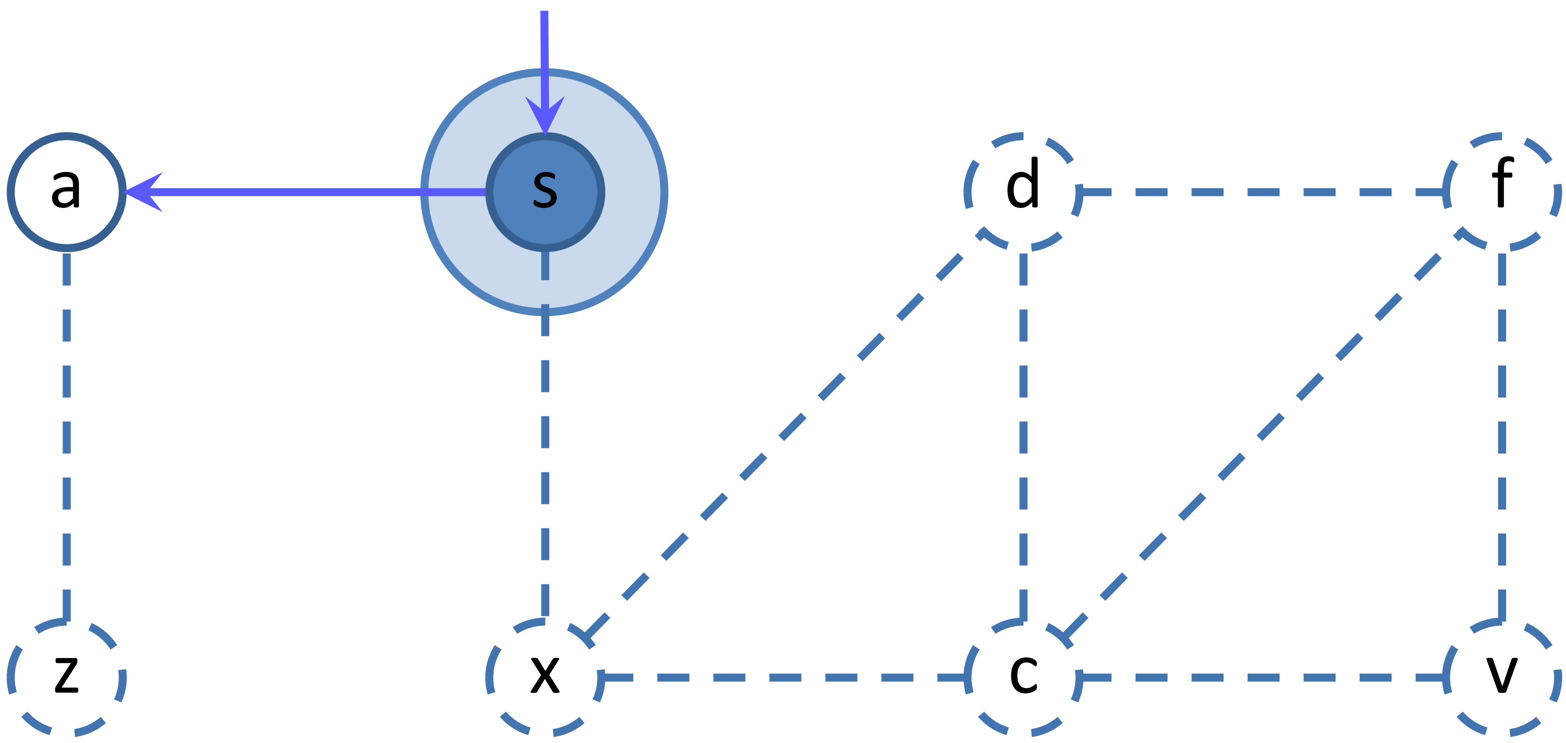


$\text{Frontier}_0 = \{s\}$, $\text{frontier}_1=\{a,x\}$, $\text{frontier}_2=\{z,d,c\}$, $\text{frontier}_3=\{f,v\}$

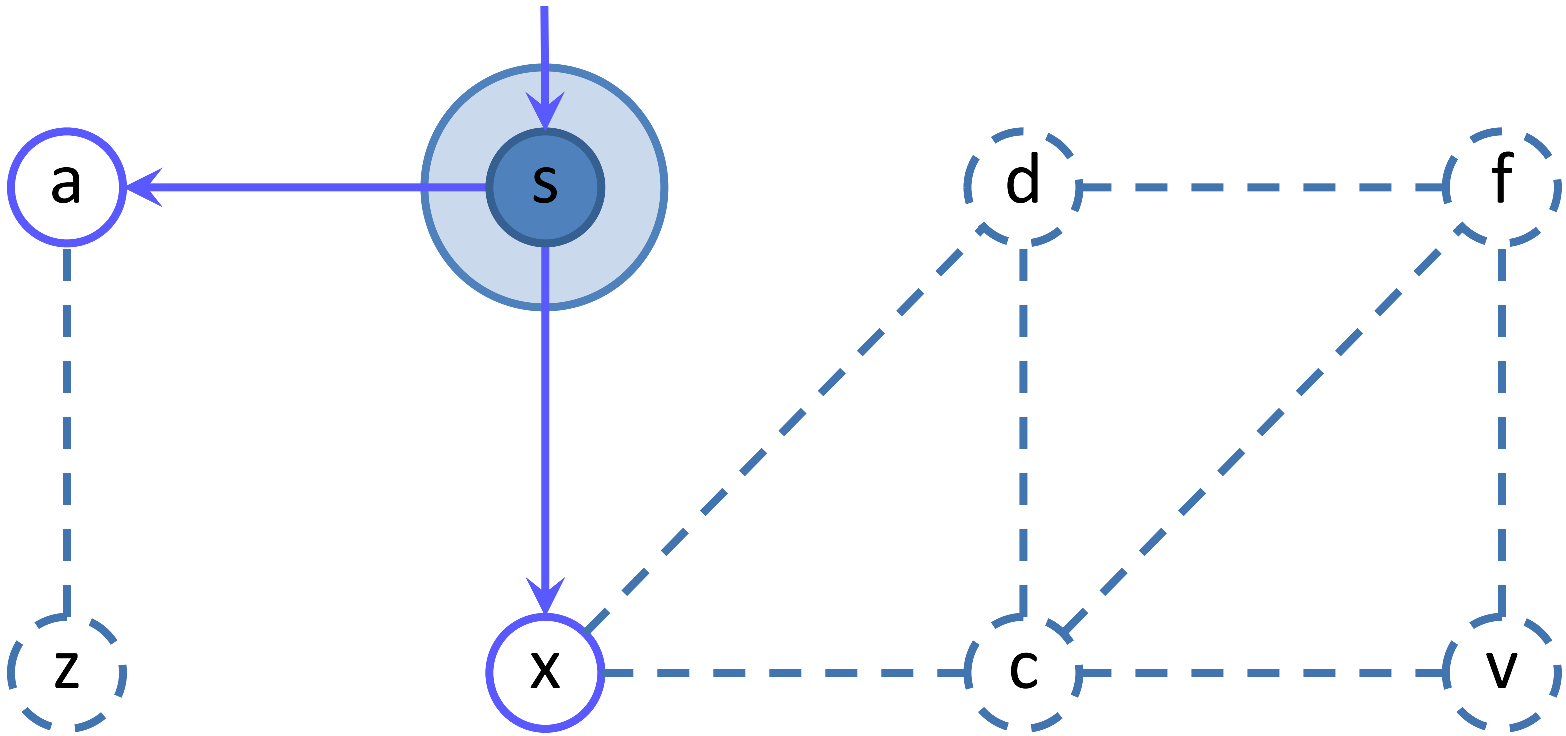
BFS: level 0



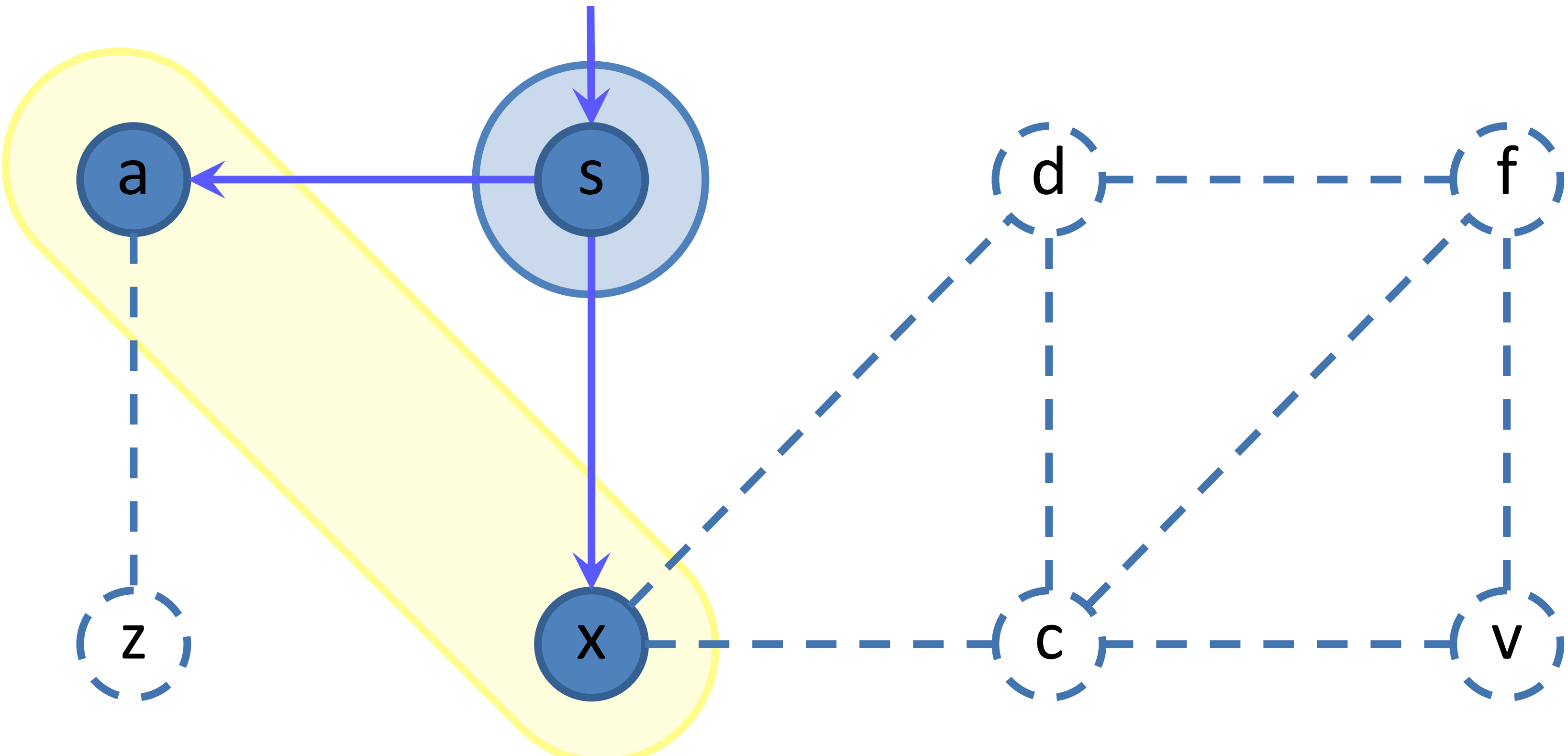
BFS: level 1



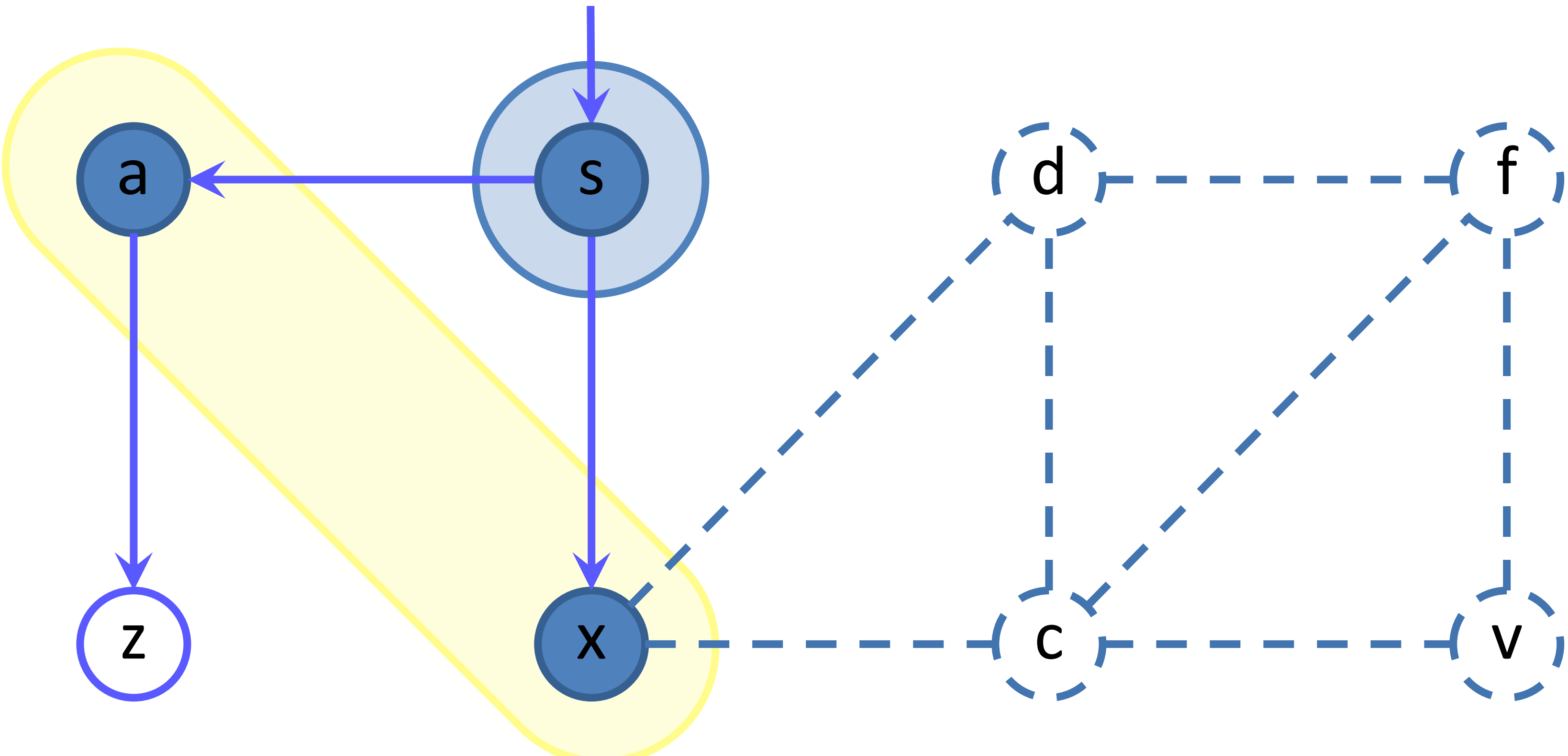
BFS: level 1



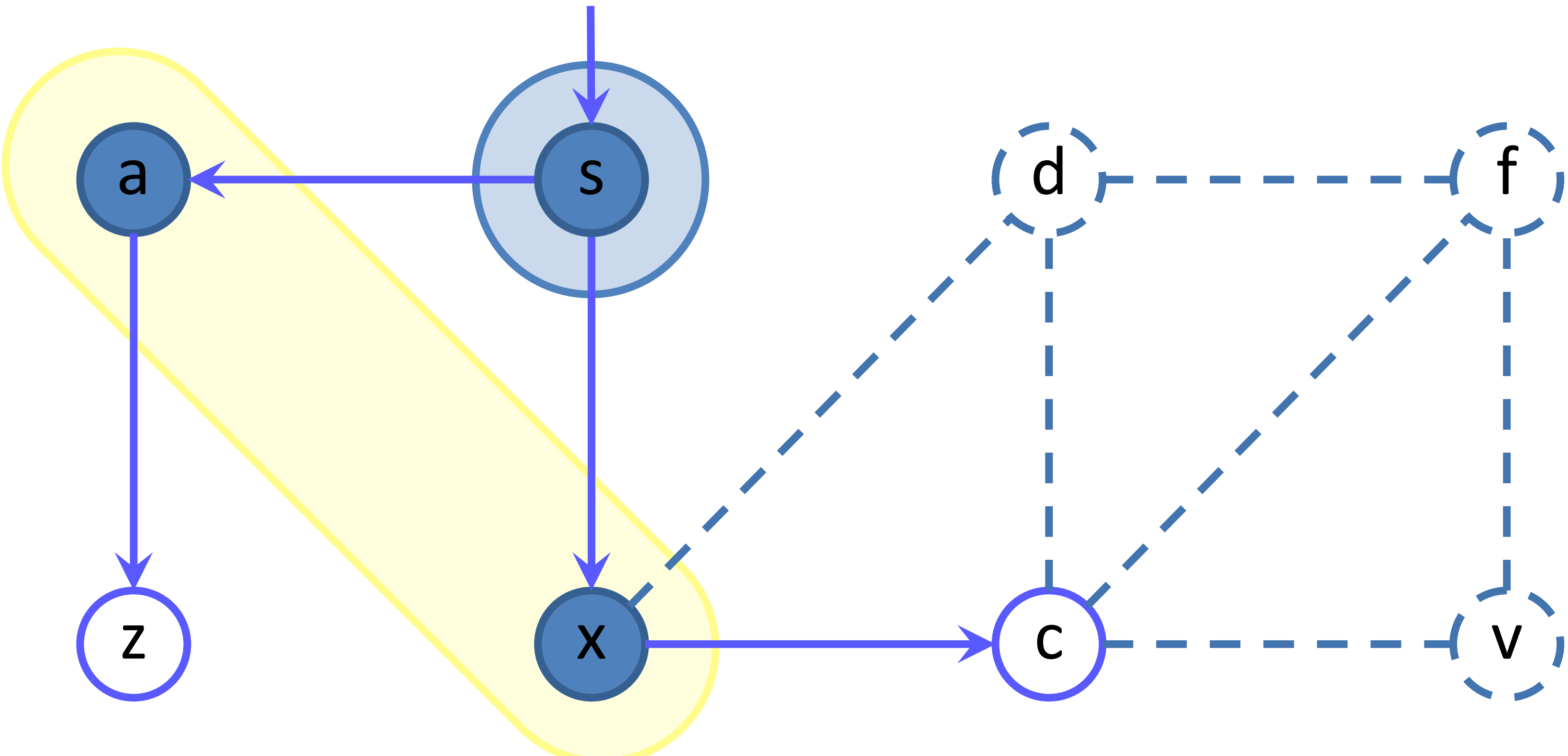
BFS: level 2



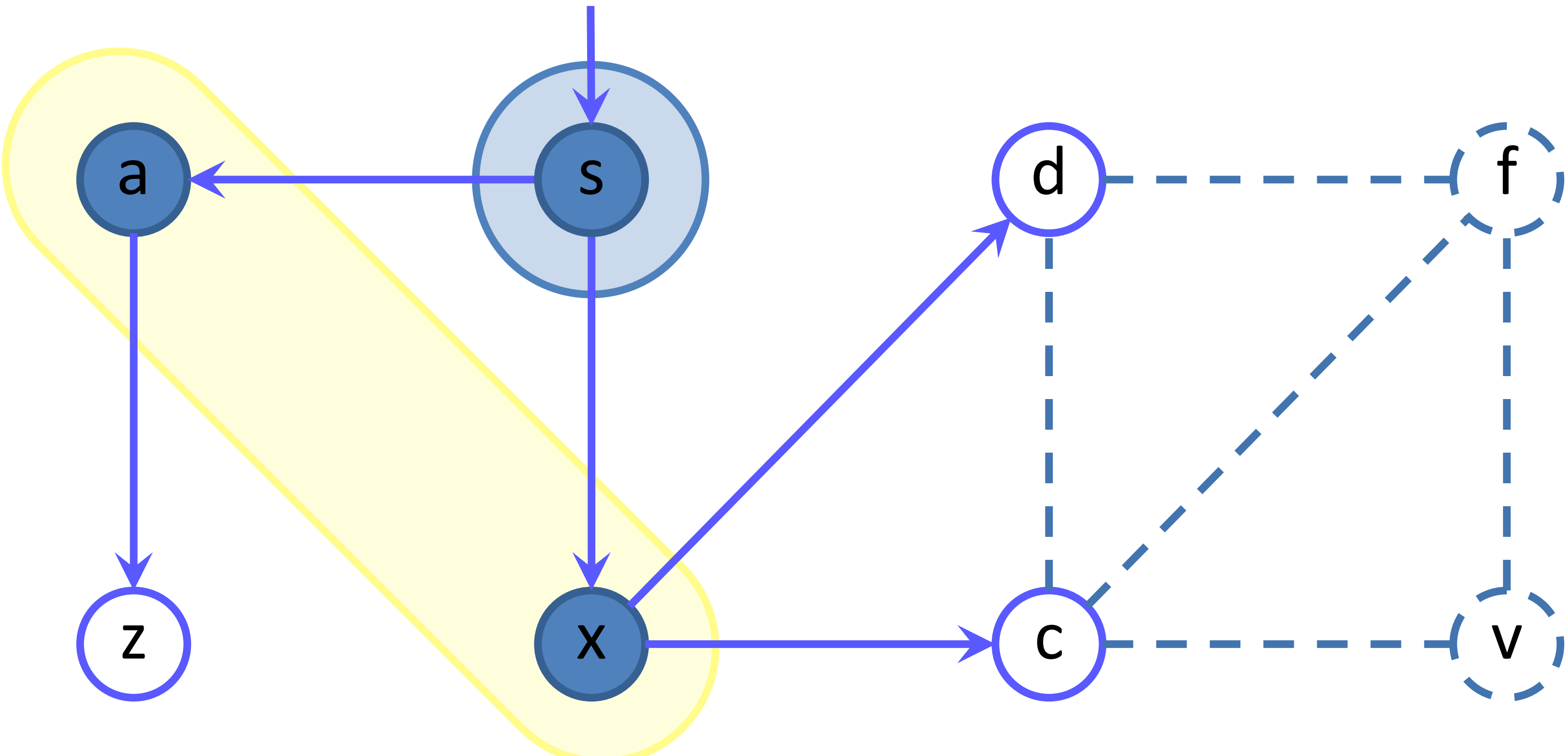
BFS: level 2



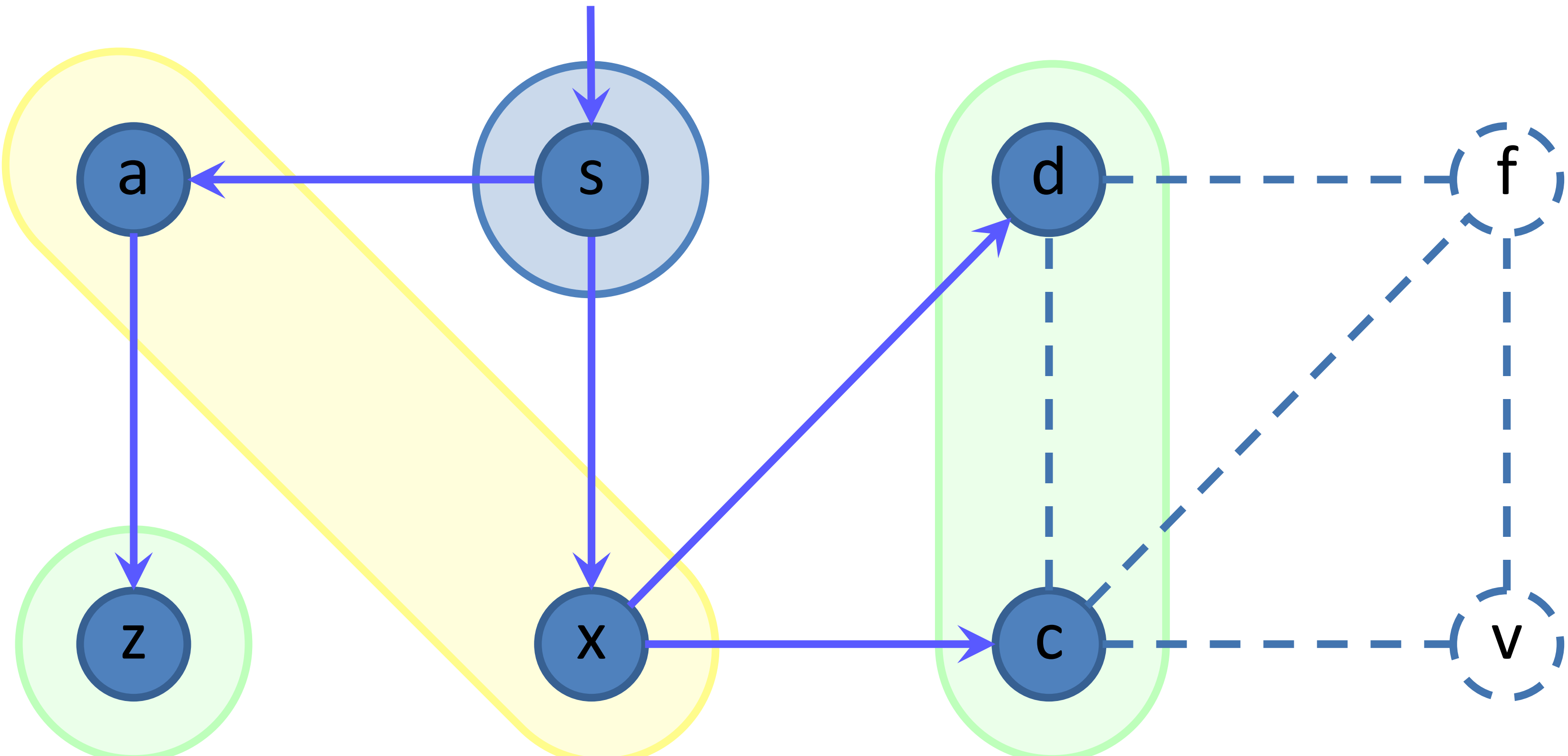
BFS: level 2



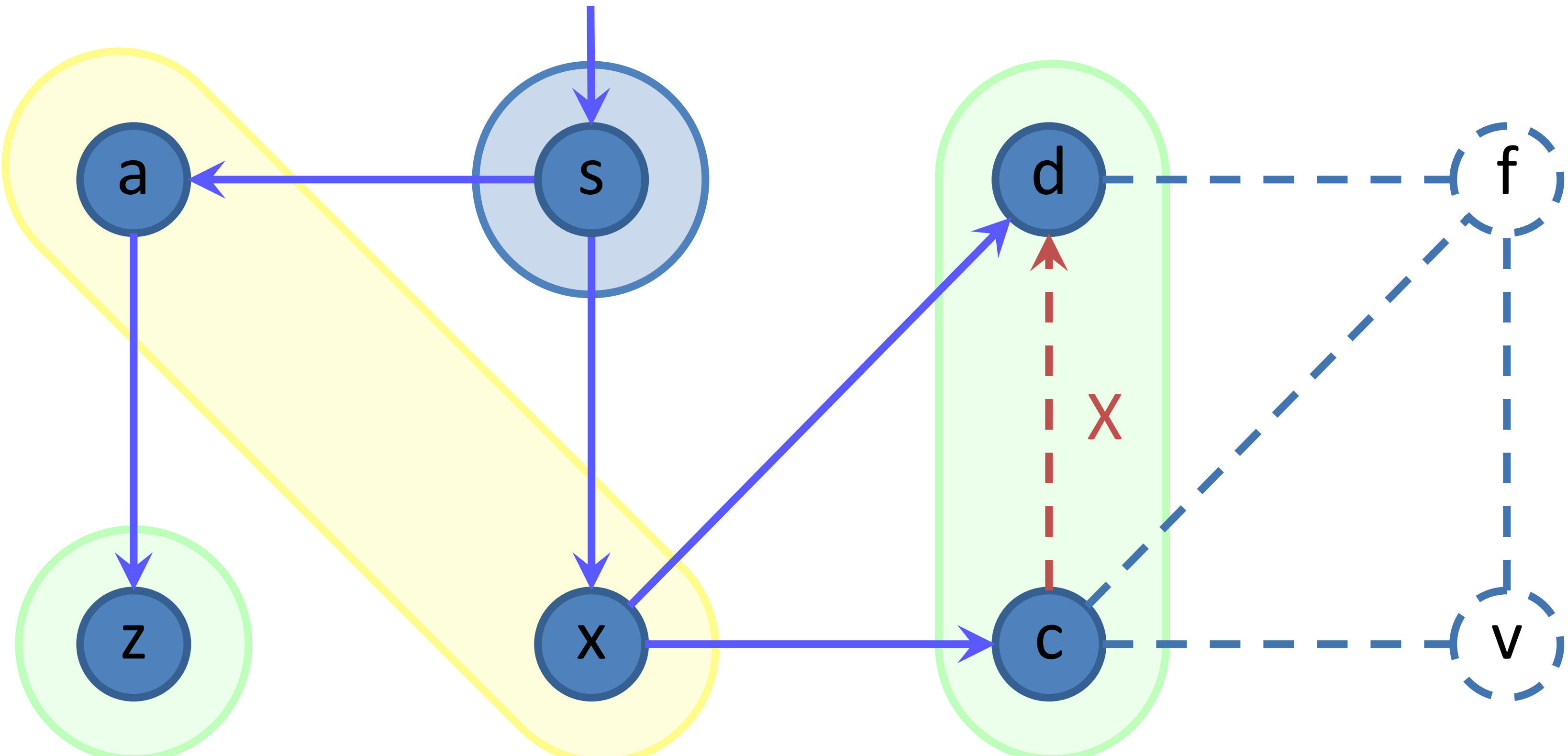
BFS: level 2



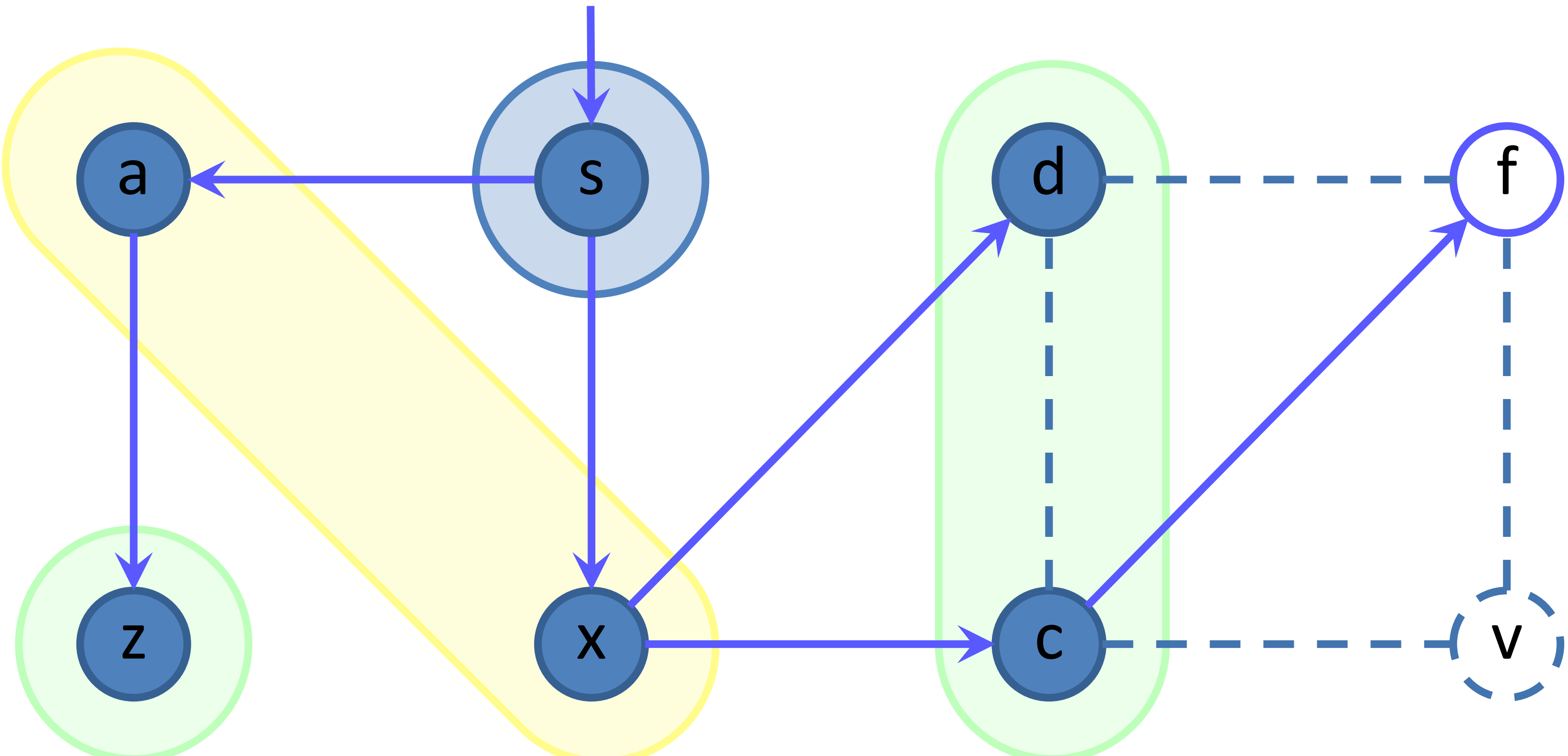
BFS: level 3



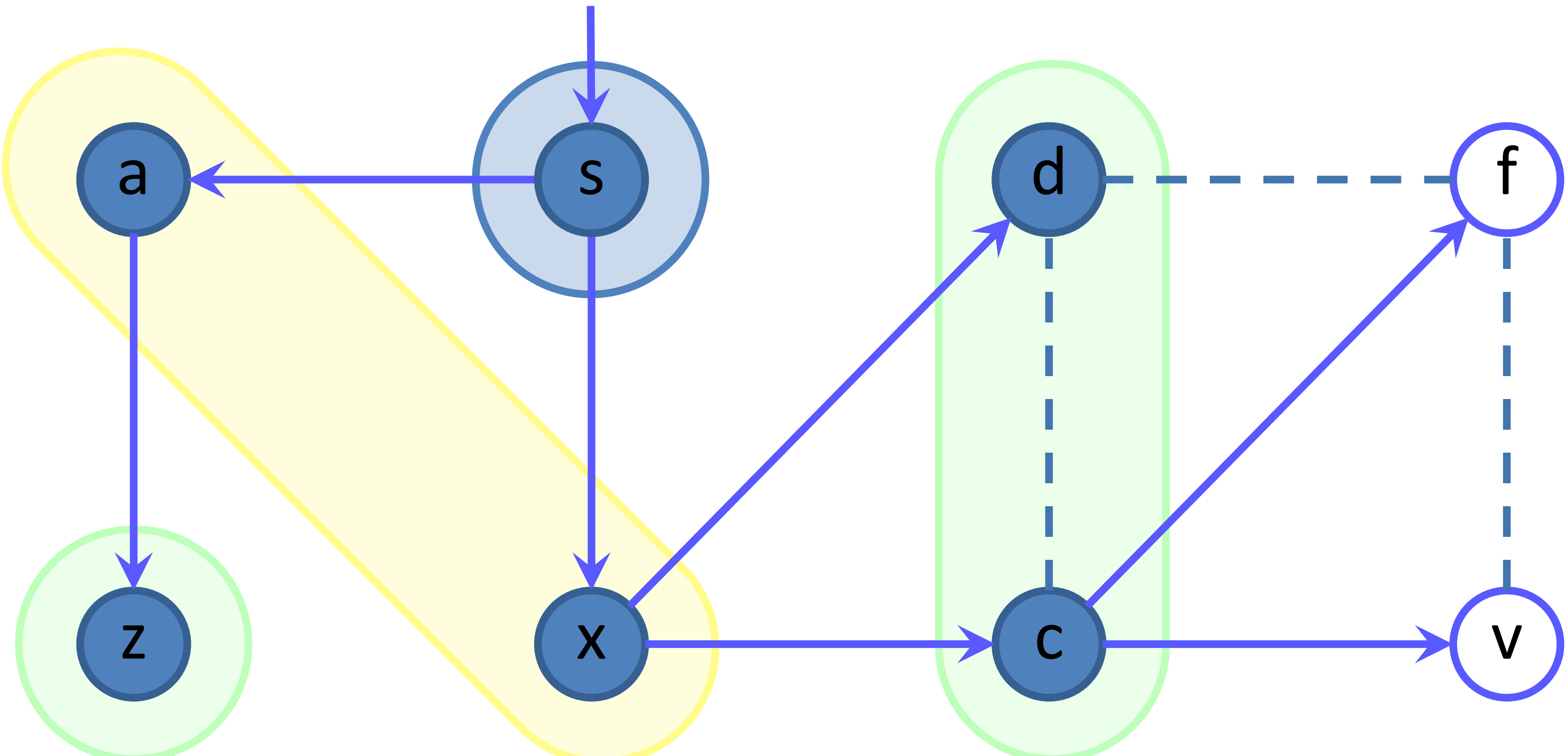
BFS: level 3



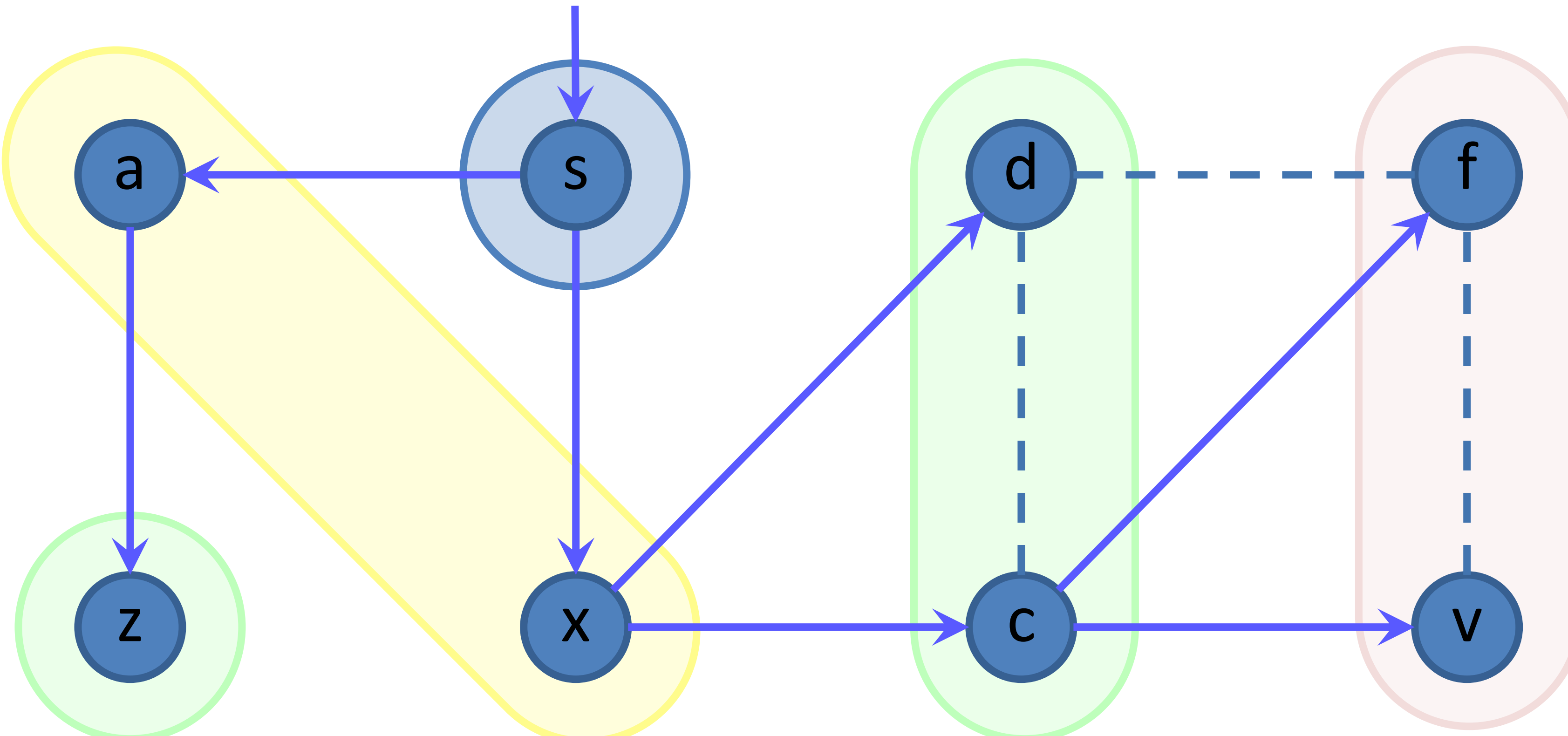
BFS: level 3



BFS: level 3



BFS: level 3



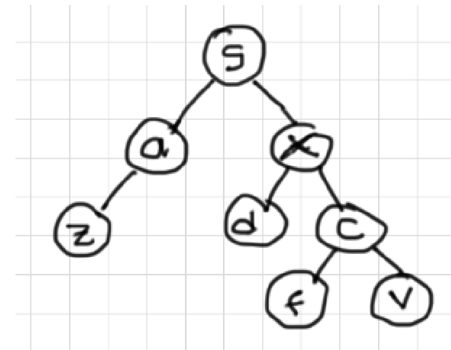
Shortest paths from s

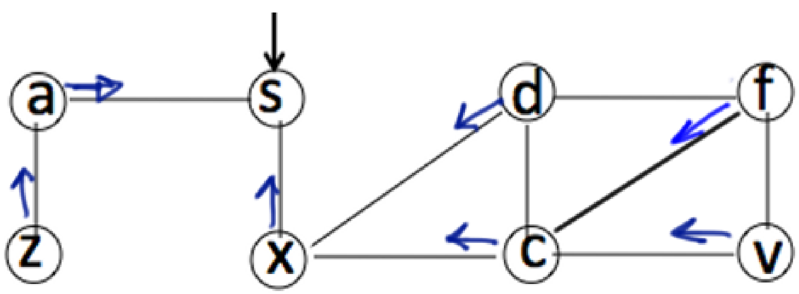
The length of shortest path from s to v

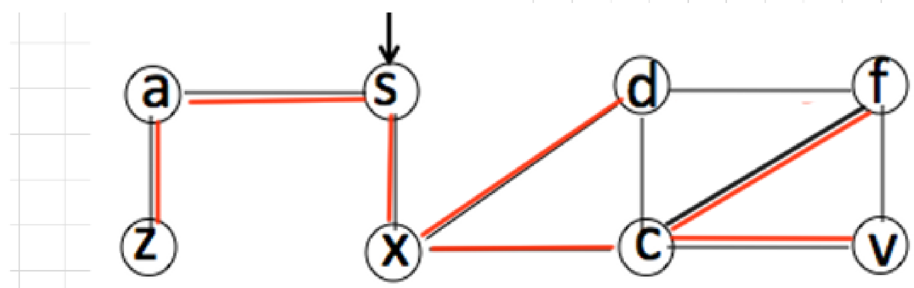
1. is level[v]
2. is ∞ (if it is non-reachable from s)

To find shortest path from s to v, follow

v->parent[v]->parent[parent[v]]->… until s







Conclusion

- Graphs: fundamental data structure
  - Directed and undirected
- 4 possible graph representations
- Basic methods of graph search (BFS)

- Next lecture:
  - DFS
  - Loops, topological sorting