(1)BFT algorithm

a b c d e

Let's update our vertex struct so that we can keep track of which vertecies have been visited:

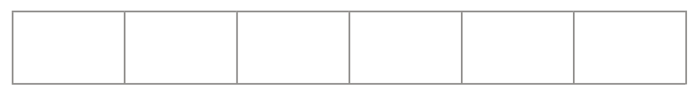
```
struct vertex{
   string key;
   bool visited = false;
   vector<edge> adj;
};

struct edge{
   int weight; // unuused in BFT
   vertex *v;
};
```

e.g. bFT(a)

Now the BFT algorithm:

```
void breadthFirstTraverse(keyStart) {
    a. find starting vertex
    b. set vStart as "visited"
    c. Create a queue (q)
```

exmple (a)



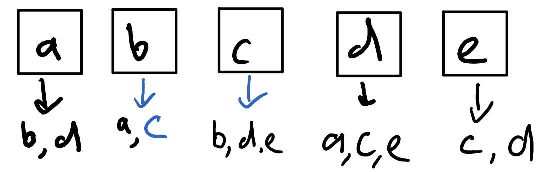
queue

n =

standard output:

(2) BFT algo - distance

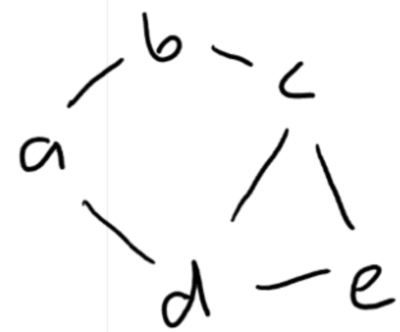
Modify the algorithm the keep track of the distance from starting vertex.




```
struct vertex{
   string key;
   bool visited = false;
   int distance = 0;
   vector<edge> adj;
};

struct edge{
    vertex *v;
};
```

e.g. bFT(a)

n →



Now the BFT algorithm:


```
void breadthFirstTraverse(keyStart)

    a. Find key of starting vertex ("root")
    b. Set vStart as "visited"
    c. Create a queue (q)
    d. Enqueue vStart onto q
    e. Loop until q is empty
       i. n = dequeue (remember queues are FIFO)
      ii. print(n->key, n->distance)
     iii. loop across n's adjacency list (x)
          1) if !n->adj[x].visited
             a) mark visited=true
             b) enqueue onto q
             c) Real C++ syntax:
                n->adj[x].v->distance = n->distance+1;
```

std output:

(3) Breadth First Search

The breadth first order of traversing a graph has a very useful property: we can figure out the shortest-path between any two points.

Update the BFT algorithm so it takes in two values:
1. start key
2. end key

Now we are searching for the shortest path from one vertex to another.

Some tweaks to the BFT are required:
- update the vertex struct to include a distance member (integer; initialize to 0)
- in the traversal, increment distance when visiting each vertex from adjacency list
- add a check for if the end key is found, return

(4) BFS

```
struct vertex{
   string key;
   bool visited;
   vector <adjVertex> adj;
};
```

Now the BFT algorithm:

```
breadthFirstSearch(keyStart,searchKey)

   Find key of starting vertex (can think of it as root)
   Set vStart as "visited"
   Set vStart distance = 0
   Create a queue (q)
   Enqueue vStart onto q
   Loop until q is empty
      n = dequeue (remember queues are FIFO)
      loop across n's adjacency list
         if !n->adj[x].visited
            n->adj[x].v->distance = n->distance+1
            if n->adj[x].v->key == searchKey
               return n->adj[x].v
            else
               mark visited=true
               enqueue onto q
   return null
```

e.g.:
bfs(a,e)

(5) Depth First Traverse

Sometimes it is more useful to explore each "branch" of the graph until the end is reached, before retreating and searching the next "branch".

Approach:

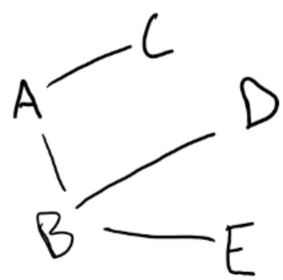
Keep visiting the first non-visited vertex in each adj list. Once adj list with 0 non-visited vertices is encountered, go back to the last intersection and visit the *next* vertex in the adj list.

Example:

DFS(A)

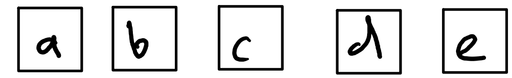
Start with A, then visit the first unvisited vertex in A's adj list.

Next, visit the first unvisited item in B's ajd list



A -> B,C
B -> A,D,E
C -> A
D -> B
E -> B

(6) DFT - stack implementation

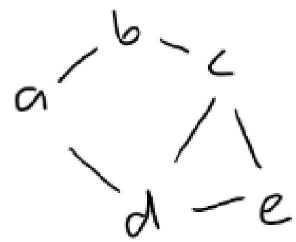


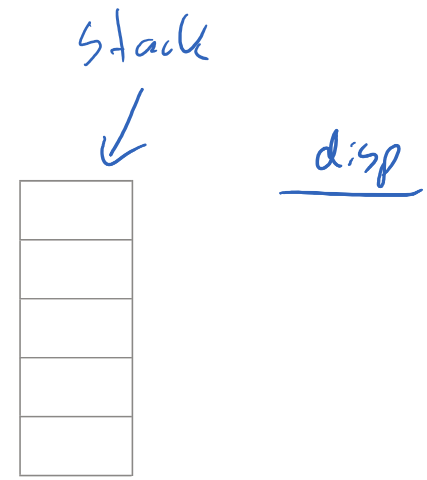
Change queue to stack to change traversal from breadth first to depth first

Now the DFT algorithm:

void depthFirstIterative(keyStart)

1.
    a. Find key of starting vertex ("root")
    b. Set vStart as "visited"
    c. Create a stack (s)
    d. Push vStart onto s
    e. Loop until s is empty
        i. n = pop
        ii. print(n->key)
        iii. loop across n's adjacency list (x), in reverse order
            1) if (!n->adj[x].visited)
                a) mark visited=true
                b) push onto s





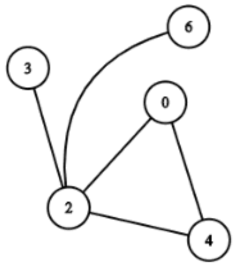
(7) DFT - recursive

ALGO:

```
depthFirstTraverse(value)
   root = find(value)
   disp(vertex key)
   depthFirstRecurse(root)
```

← public

```
depthFirstRecurse(vertex)
   vertex.visted = true
   loop adj list of vertex
     if(adj vertex is not visited)
       disp(disp adj vertex)
     ->depthFirstRecurse(adj vertex)
```



e.g. dft(2)

3->	2
2->	3,6,0,4
6->	2
0->	2,4
4->	2,0

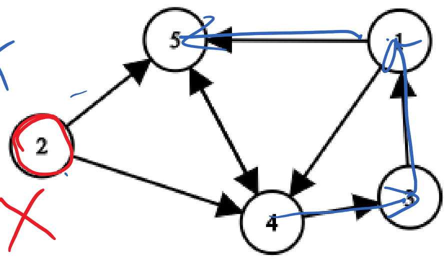
(8) DFT: Example Application

Strongly connected components:

- A directed graph is considered strongly connected, if for every vertex, there is a path to every other vertex.

Graph A

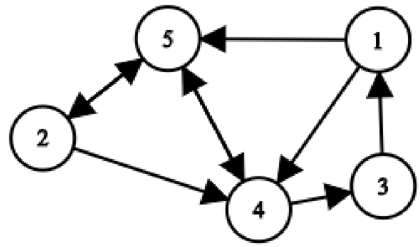
DFT



not strongly connected

DFT

Graph B



Yes (strongly connected)

Solution:

- Deploy an instance of the DFT on EVERY vertex in the graph. After each DFT, linearly check each vertex to see if it has been visited. If *any* vertex has not been visited, then we know the graph is not strongly connected.

(9) BFS vs DFS

BFS:

- best for solving shortest path kind of problems
- memory constraint: stores all the nodes of the current level to go to the next level

DFS:

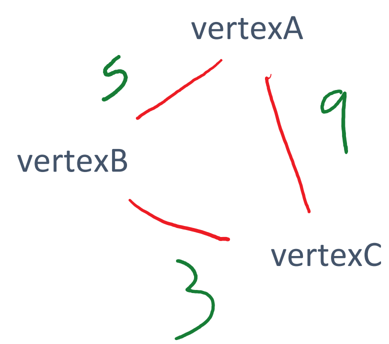
- often used in connectivity type of problems
- can be implemented recursively (however, risk of stack overflows)

does not look at weights

(10) Graph ADT - weighted

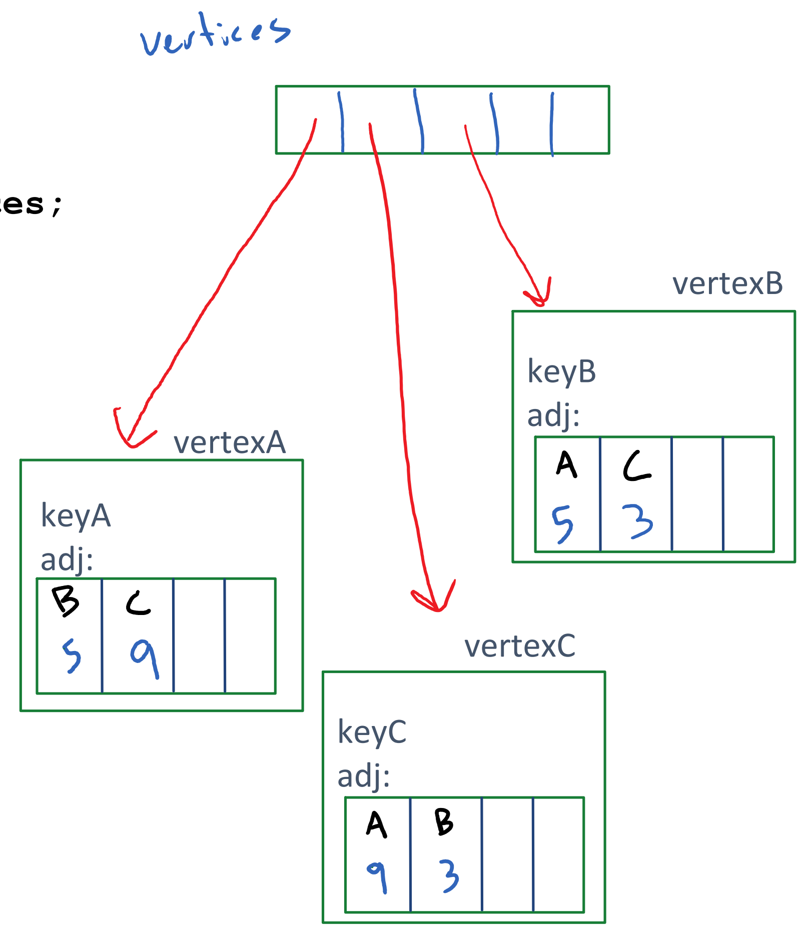
private:
- vertices
  - edges (contained within each vertex)

public:
- insertVertex(value)
- addEdge(startValue, endValue, *weight*)
- deleteVertex(value)
- deleteEdge(startValue,endValue)
- displayGraph()
- search(value)
- **BreadthFirstTraverse(startValue)**
  - *shortest path*
- **DepthFirstTraverse(startValue)**
  - *pursue a branch until the end...*
- **Dijkstra'sTraverse(startValue)**
  - *shortest path for a weighted graph*



C++ definitions:

```
private:
    std::vector<vertex*> vertices;

struct vertex{
    string s;
    vector<edge> adj;
};

struct edge{
    vertex * v;
    int weight;
};
```



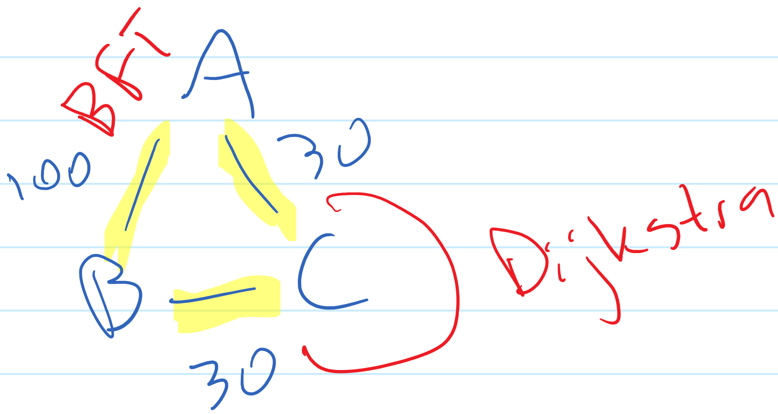
(11) Dijkstra's Concept

Breadth First finds the shortest path in an *unweighted* graph.

If given a graph with edge weights, BFS cannot be modified to find the shortest path based on the weights.

Instead, we need a traversal algorithm for weighted graphs such as Dijkstra's.

Consider flight routing example: number of lay-overs (BFS) vs. total distance.



1. Find the starting vertex using a simple linear search.
2. Add the starting vertex to a list (call it solvedList)
3. Visit all adjacent vertices in each of the items in the solved list
    a. Mark the min distance vertex as "solved", and add it to the solved list
4. Go back to step 3; repeat until all vertices have been marked as solved

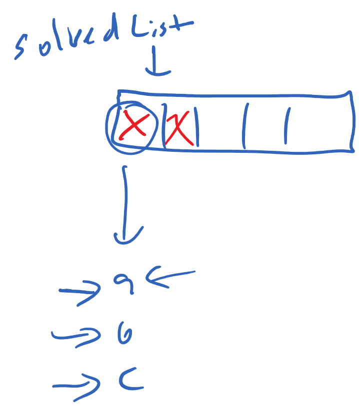
(12) Dijkstra's Implementation

Vertex Struct Definition:

```
struct vertex{
    string key;
    vector<edge> edges;
    bool solved;    // similar to "visited"
    int distDijk;   // total distance from start
};
```

Pseudo-code:

```
dijkstraTraverse(key){
    vStart = search(key)
    vStart solved = true
    create a list (vector), call it solvedList
    add vStart to solvedList
    create boolean variable, call it allSolved
    set allSolved = false

    loop as long as allSolved is NOT true{
        create a pointer, call it solvedV
        loop through entire solvedList (use iterator i){
            loop through the adjacency list of sovedList[i] (use j){
                if (solvedList[i]->adj[j].v is NOT solved)
                    calculate the distance from vStart, call it dist
                    if( dist < minDist )
                        solvedV = solvedList[i]->adj[j].v
                        minDist = dist
                    allSolved = false
            }
        }
        if(!allSolved)
            solvedV->distDijk = minDist
            solvedV->solved = true
            add solvedV to solvedList
    }
}
```



(13) Dijkstra's: Simulation

Need to update the vertex struct:

```
struct vertex{
    string key;
    vector<edge> edges;
    bool solved;    // similar to "visited"
    int distDijk;   // total distance from start
};


struct edge{
    vertex *v;
    int weight;
};
```

Adjacency lists (with weights):

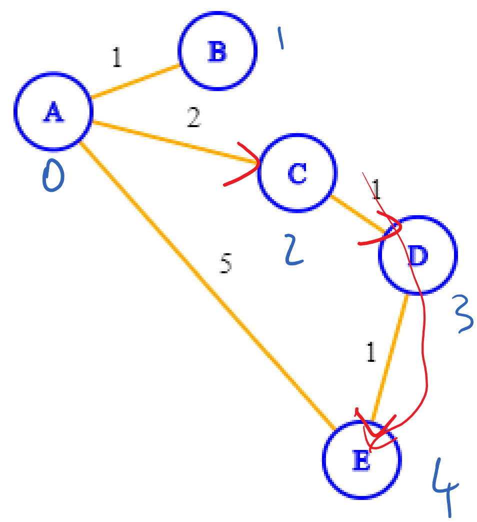
A -> B-1, C-2, E-5

B -> A-1

C -> A-2, D-1

D -> C-1, E-1

E -> A-5



Example: *dijkstra(A)*

1. *Start at A, mark as solved, add to solved list.(A-0 i)*
2. *Traverse entire solved list*
    a. *Scan A's adj list and find un-solved vertex nearest to A*
    b. *Mark closest vertex as solved, add to solved list*
3. *Traverse entire solved list*
    a. *Scan A's adj list and find un-solved vertex nearest to A*
    b. *Scan B's adj list and find un-solved vertex nearest to A*
    c. *Mark closest vertex (min(a,b)) as solved, add to solved list*
4. *Traverse entire solved list*
    a. *Scan A's adj list - closest = E (5)*
    b. *Scan B's adj list - closest = nothing*
    c. *Scan C's adj list - closest = D (3)*
    d. *min(a,b,c) = D(3)*
5. *Traverse entire solved list*
    a. *Scan A's adj list - closest = E (5)*
    b. *Scan B's adj list - closest = nothing*
    c. *Scan C's adj list - closest = nothing*
    d. *Scan D's adj list - closest = E (4)*
    e. *min(a,b,c,d) = E(4)*

solved list: *(vertex, distance from A)*

A, 0

B, 1

C, 2

D, 3

E, 4

(14) Dijkstra's in C++, pt. 1

all vertices initialized with solved = false

I.

```
void Graph::dijkstraTraverse(string sourceVertex){

    vertex *vStart = search(sourceVertex);
    if(!vStart){
        cout << "Start not found" << endl;
        return;
    }

    vStart->solved = true;
    vStart->distDijk = 0;

    // Create a list to store solved vertices
    // and append vStart
    vector<vertex*> solvedList;
    solvedList.push_back(vStart);

    // Will use this Boolean variable to leave the loop
    // if all vertices have been solved
    bool allSolved = false;

    while(!allSolved){...
}
```

(15) Dijkstra's in C++, pt. 2

II. The loop

```
while(!allSolved){
   int minDist = INT_MAX;

   // pointer to keep track of solved node
   vertex *solvedV = nullptr;
   allSolved = true;

   // iterate across list of solved vertices
   for(int i=0; i<solvedList.size(); i++){
      vertex *s = solvedList[i];

      // now iterate s's adjacency list
      for(int j=0; j<s->adj.size(); j++){
         if(!s->adj[j].v->solved){

            // calculate the distance from vStart
            int dist = s->distDijk + s->adj[j].weight;

            // check if the distance is less than
            // smallest distance thus far
            if(dist<minDist){
               solvedV = s->adj[j].v;
               minDist = dist;
            }
            allSolved = false;
         }
      }
   }
```

(16) Dijkstra's in C++, pt. 3

II. The loop

```
while(!allSolved){
   int minDist = INT_MAX;

   // pointer to keep track of solved node
   vertex *solvedV = nullptr;
   allSolved = true;

   // iterate across list of solved vertices
   for(int i=0; i<solvedList.size(); i++){
      vertex *s = solvedList[i];

      // now iterate s's adjacency list
      for(int j=0; j<s->adj.size(); j++){
         if(!s->adj[j].v->solved){
            // calculate the distance from vStart
            int dist = s->distDijk + s->adj[j].weight;

            // check if the distance is less than
            // smallest distance thus far
            if(dist<minDist){
               solvedV = s->adj[j].v;
               minDist = dist;
            }
            allSolved = false;
         }
      }
   }
   if(!allSolved){
      solvedV->distDijk = minDist;
      solvedV->solved = true;
      solvedList.push_back(solvedV);
   }
}
```