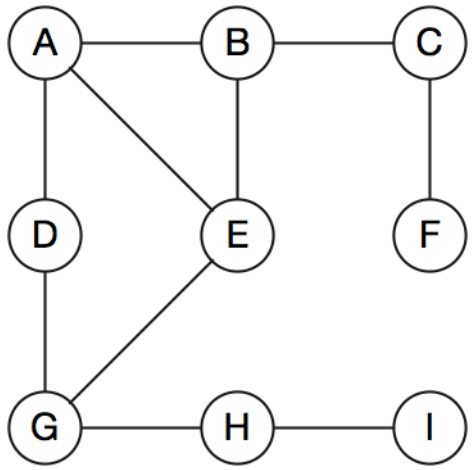
Graphs

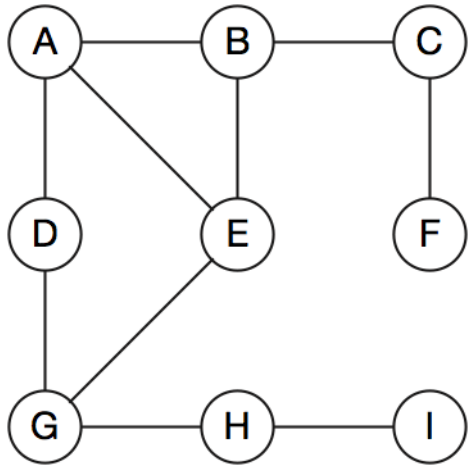
The Graph Data Structure

- Represents a relationship between pairs of objects
- A graph is a set of objects—vertices—with pairwise connections between them —edges.
- It’s not a plot of values on a grid.
- An alternative notation: *nodes* connected by *arcs*.



Graph Terminology

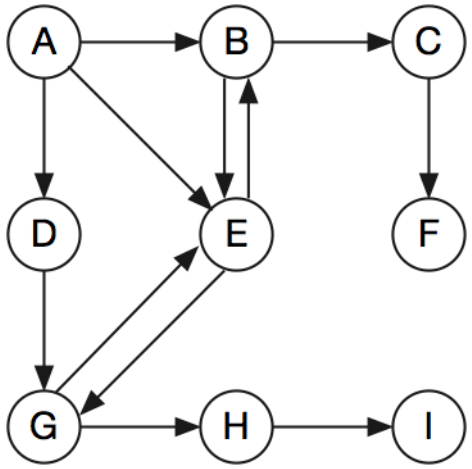
- Two vertices u and v are *adjacent* if there's an edge whose endpoints are u and v.
- Two adjacent vertices are *neighbors*; a vertex can have several neighbors.
- A *path* is a sequence of edges.
- *Path length* is the number of edges in the sequence.
- *Distance* is the number of edges in the shortest path.



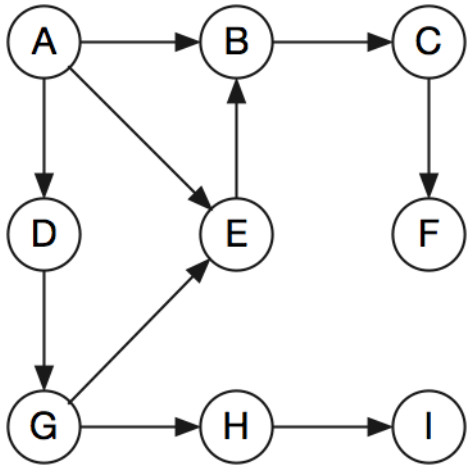
- A is adjacent to E
- C – B – A – E is a path of length 3
- C's distance from E is 2 (path C – B – E)

More Terminology

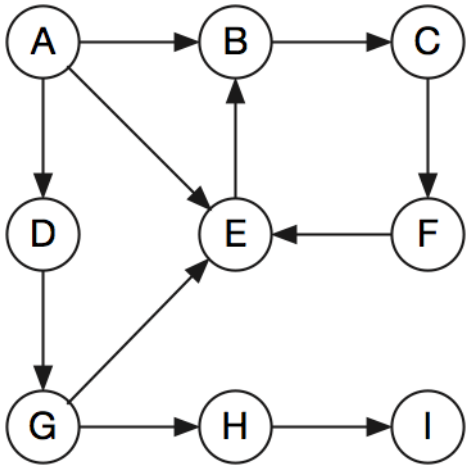
- *Directed graph*: its paths are one-way; vertices can be connected by edge pairs.
- *Acyclic graph*: a directed graph where no path starts and ends at the same vertex
- *Cyclic graph*: a directed graph where some paths start and end at the same vertex



Directed



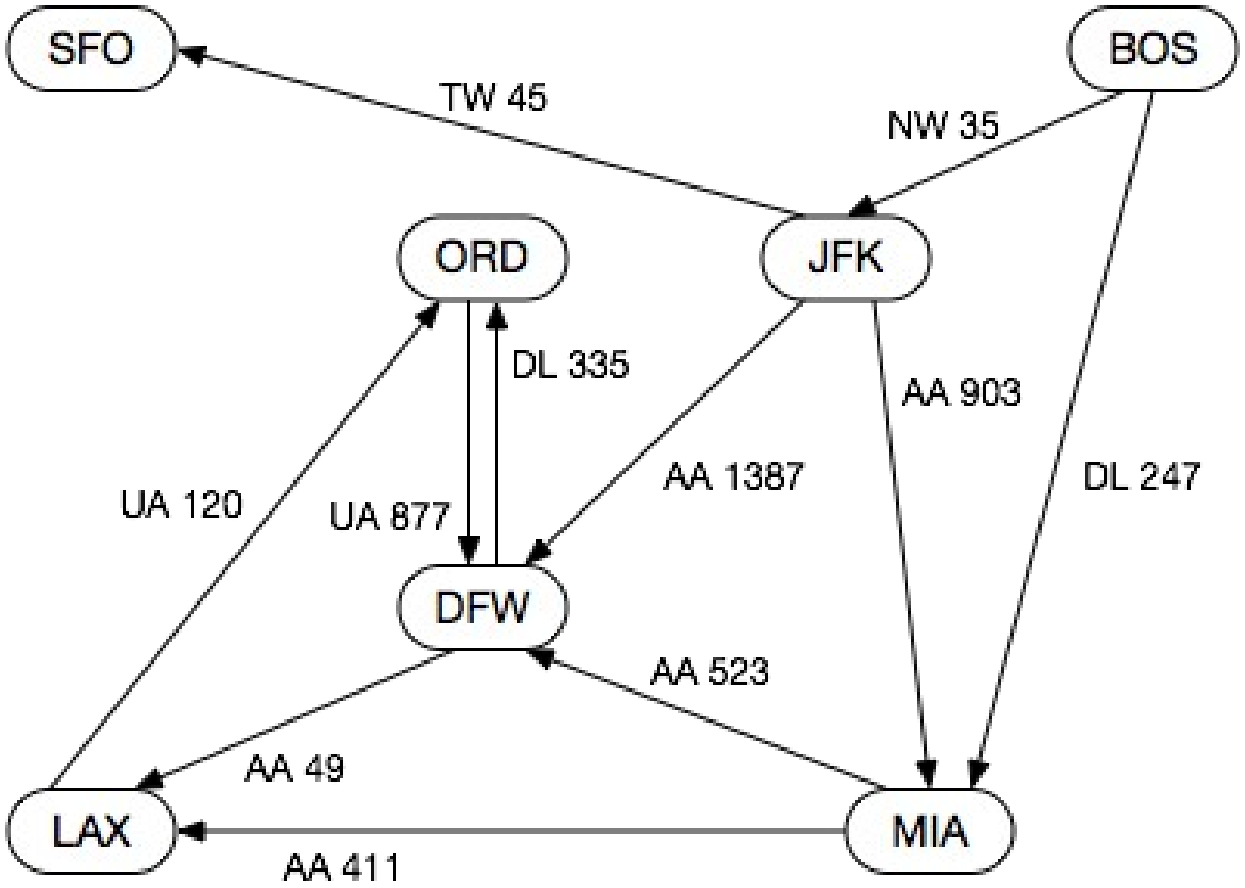
Acyclic



Cyclic

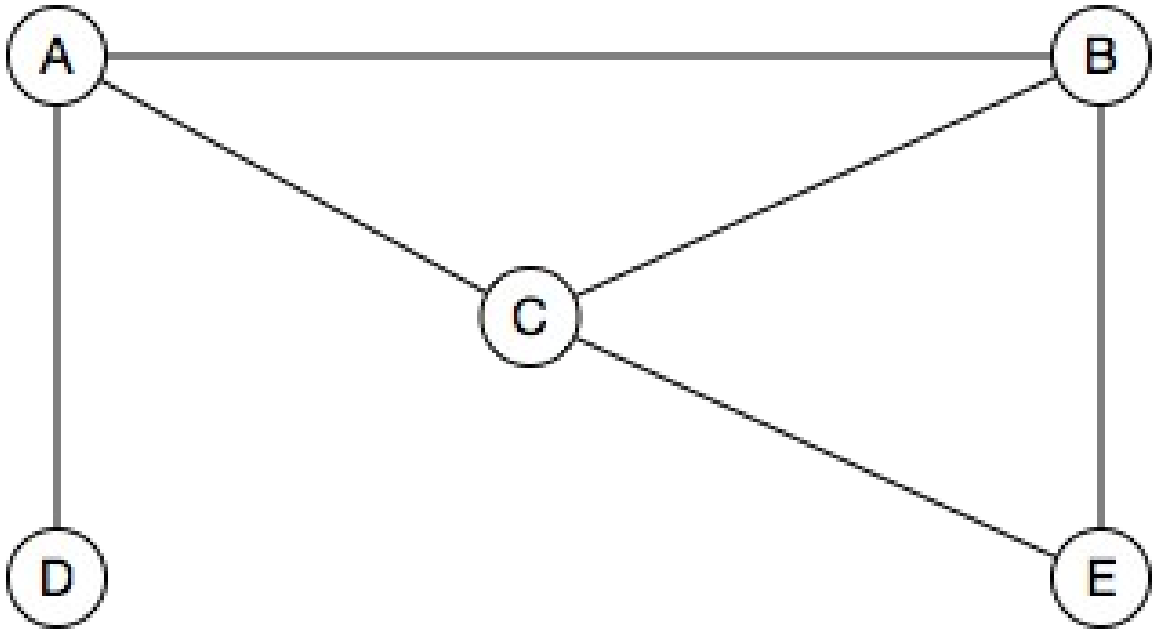
Examples of Graphs

- A city map: intersections and dead ends are vertices; streets are edges
- The plumbing or electrical systems in a building: pipes, connectors, and valves; conduits, wires, junctions, switches, and outlets.
- An aircraft flight network: airports and airplanes.
- The Internet: hosts, cables, and routers.
- A project plan and the order of its tasks:
  - concurrency: some tasks can be done in parallel.
  - precedence: some tasks must come before others.

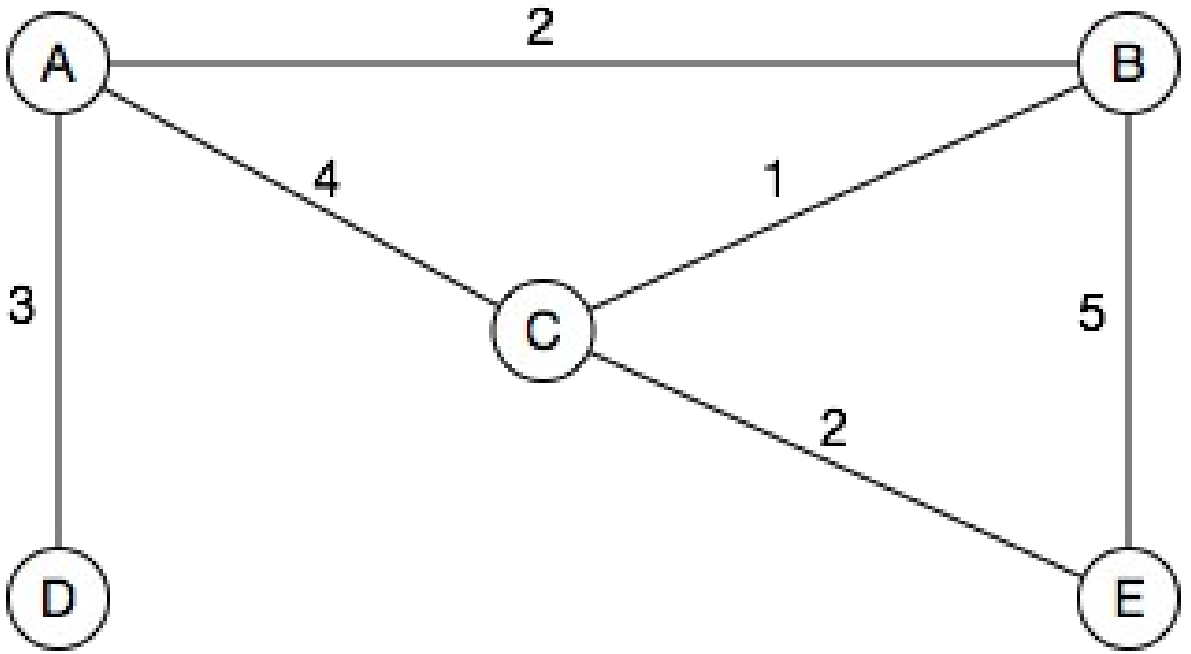


Flight Network.

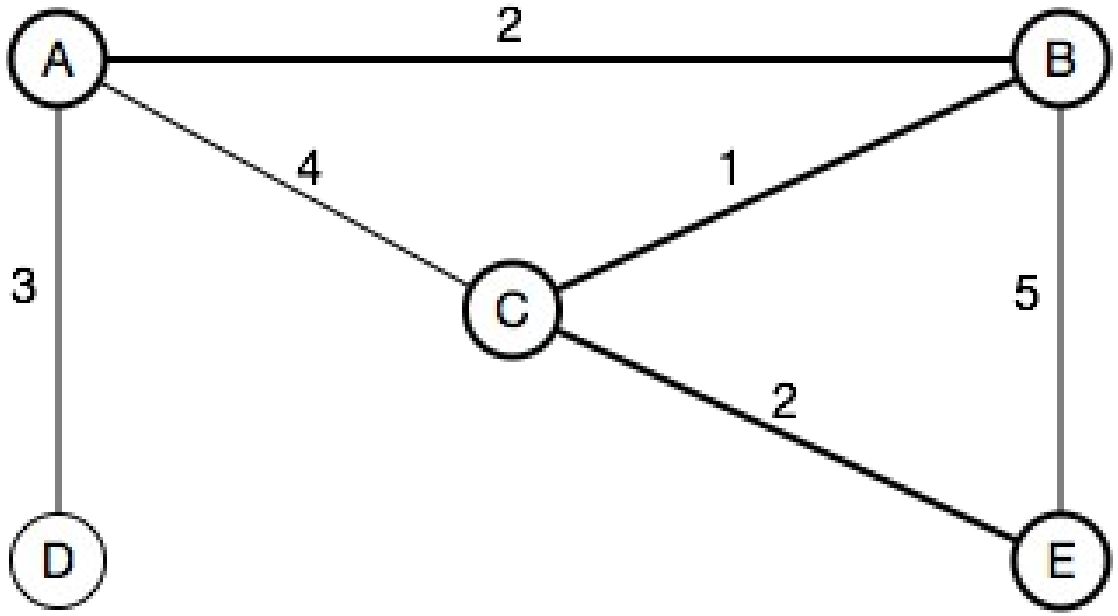
A Computer Network Example



A computer network with five nodes—hosts and routers.
What’s the best path from A to E?

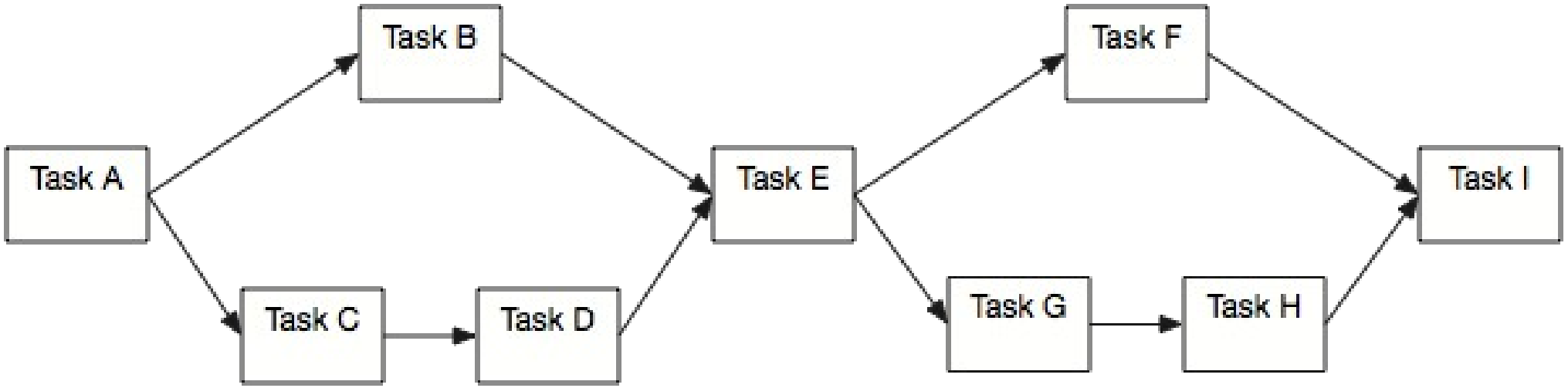


A computer network with the cost of each link.

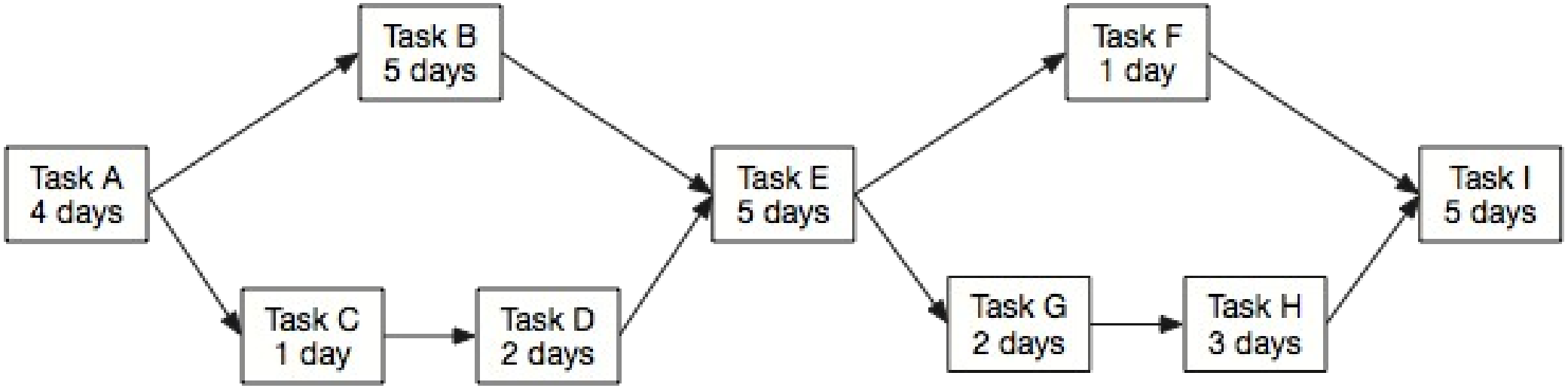


The best path isn’t the shortest one, it’s the fastest one.

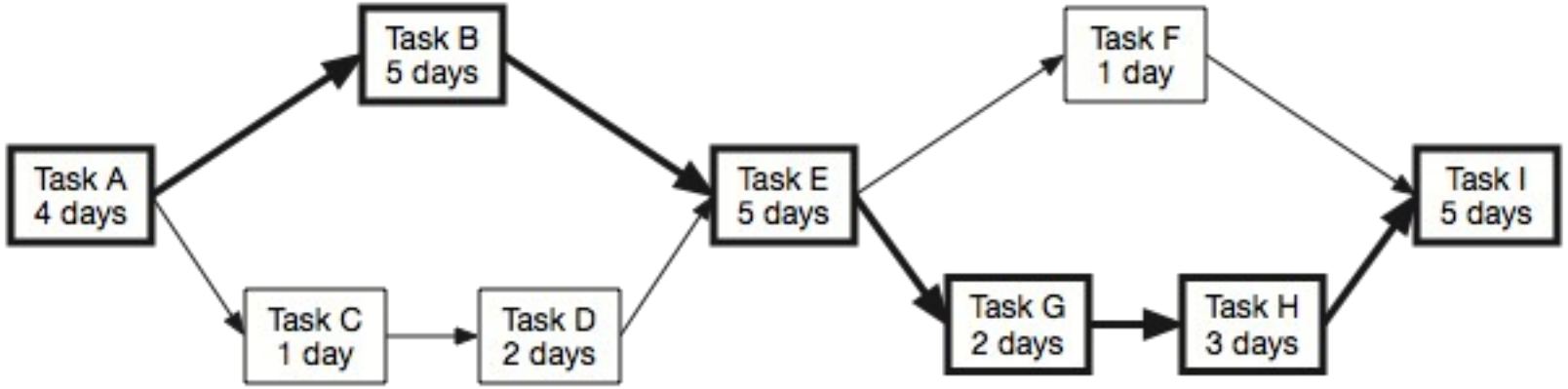
A Project Plan Example



A project precedence network, also known as a Critical Path Chart. What's the longest path through the project, which determines how long the project will take and what the most important tasks are?



The Critical Path Chart, with the duration of each task shown.

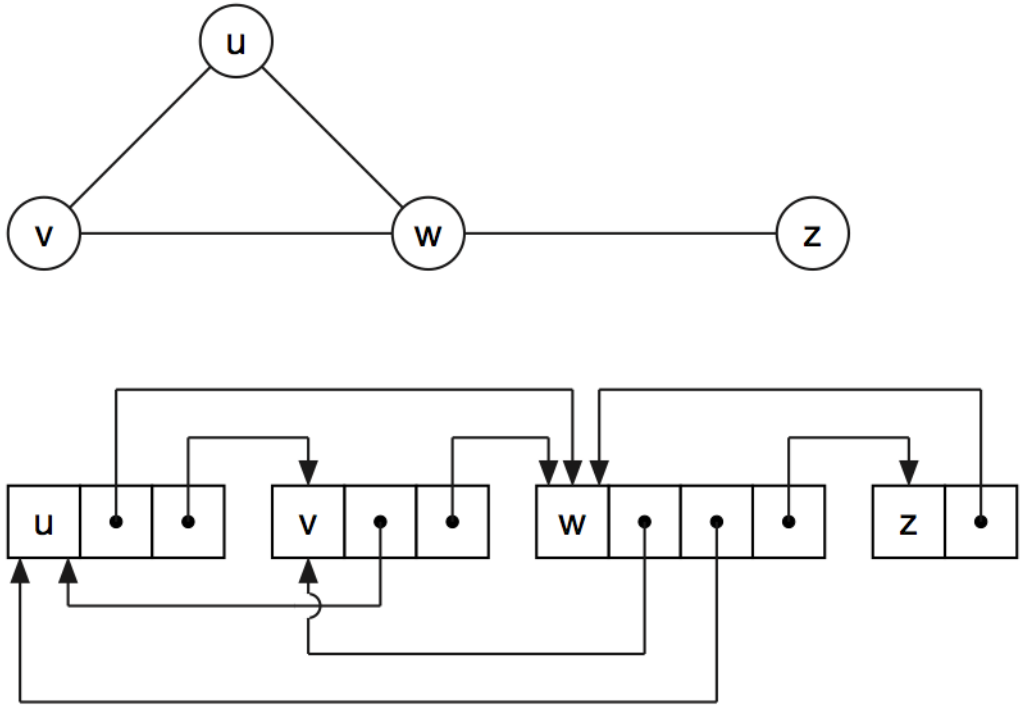


- The Critical Path Chart, with the critical path shown. The project will take 24 days, if nothing goes wrong.
- Tasks A, B, E, G, H, and I are on the critical path; if any one of them is delayed, the project will finish late. They need careful watching.
- Tasks C, D, and F aren't critical; they can slip without jeopardizing the project. Resources could be diverted to help critical tasks.

Data Structures for Graphs

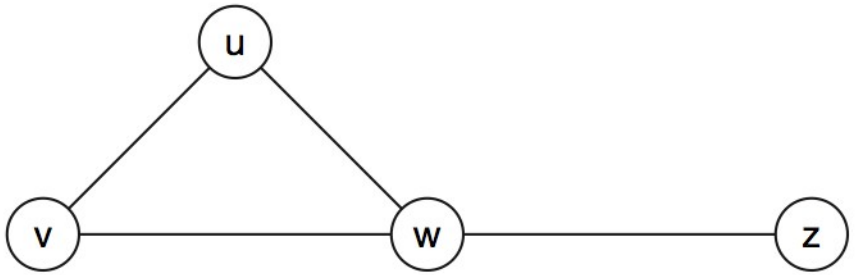
Adjacency List

- Compact, usually sparse
- Determining adjacency of neighbors can be slow



Adjacency Matrix

- Quick determination of neighbors
- Uses considerable memory



	u	v	w	z
u		•	•	
v	•		•	
w	•	•		•
z			•	

Edge Data

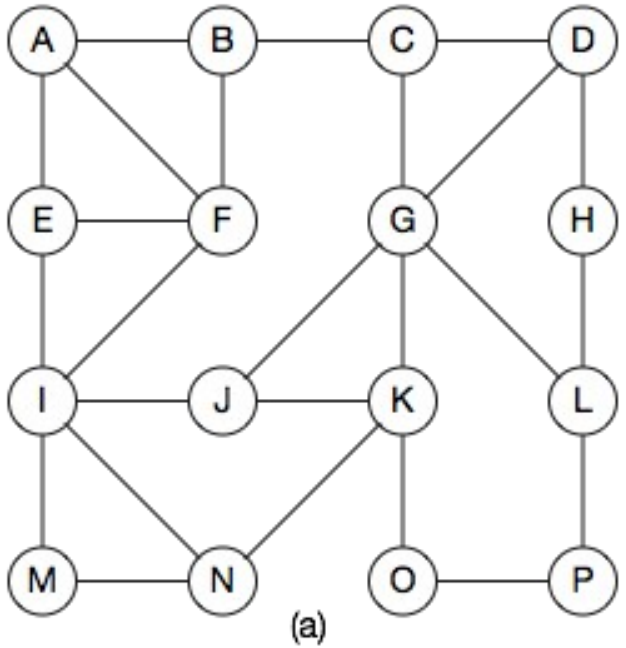
- Examples:
  - Computer network: link costs
  - Flight network: airline and flight number
  - Project plan: task names and durations
- Adjacency list vertices can point to edge data structs containing:
  - Edge data
  - Pointer to neighbor
- Adjacency matrix cells can point to edge data structs

Graph Traversals

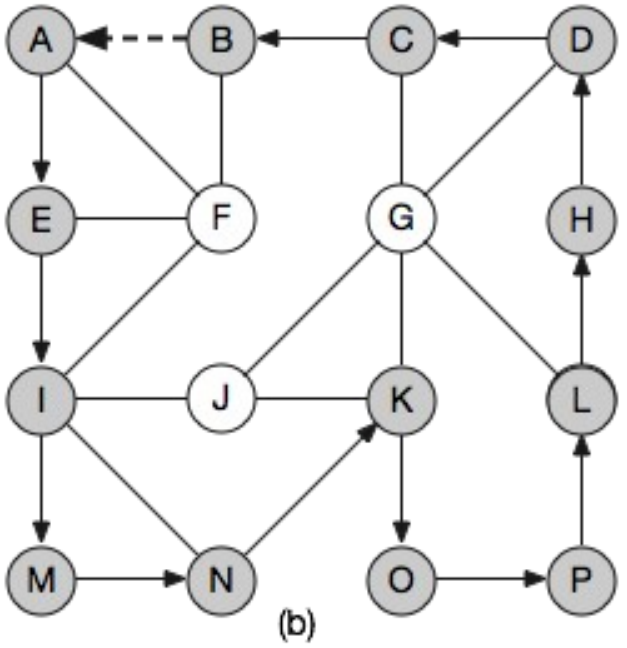
Depth-First Traversal

- For an undirected graph (also works for a directed graph)
- Traversing a labyrinth with a string, a paint can, and a brush
- Pay out the string as we go; it will let us retrace our steps.
- Mark vertices with paint as we visit each one.
- A vertex can have several edges leading to adjacent vertices; we'll visit each one in turn.
- We could arrive at a vertex we've already visited; the paint mark will show us that and we'll back up to the previous vertex.
- When there are no more adjacent vertices that haven't been visited, we'll back up one more vertex (following the string back to where we came from)
- As we back up, we'll look for unvisited adjacent vertices.
- Eventually we'll arrive back at our starting point, having visited every vertex and every edge.

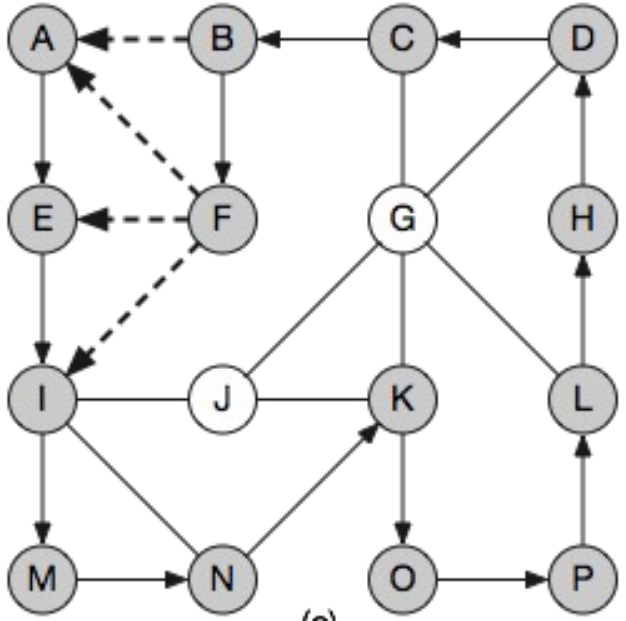
A depth-first search starting at vertex A.



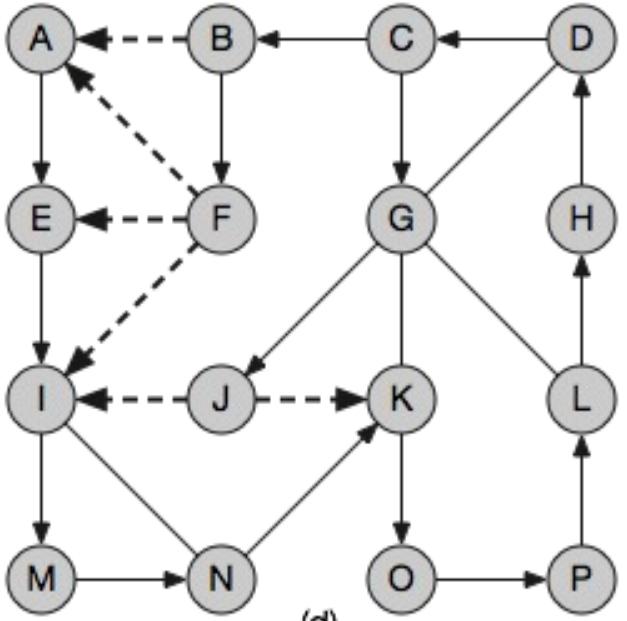
(a)



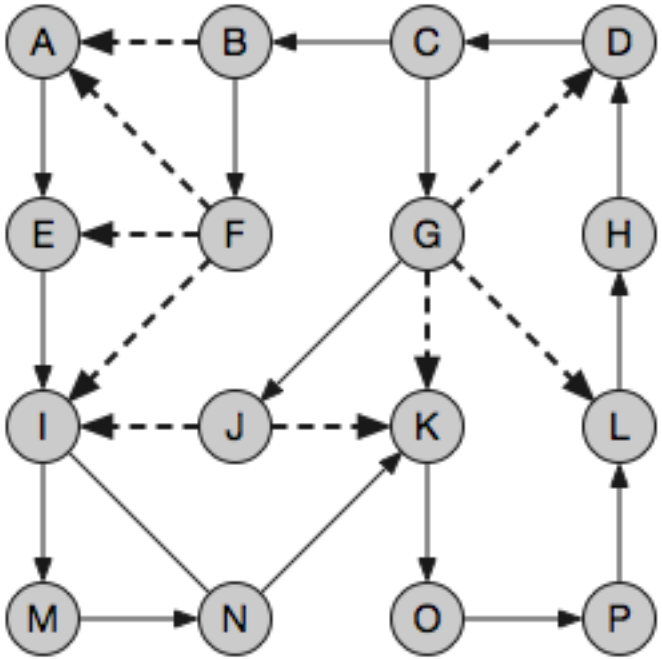
(b)



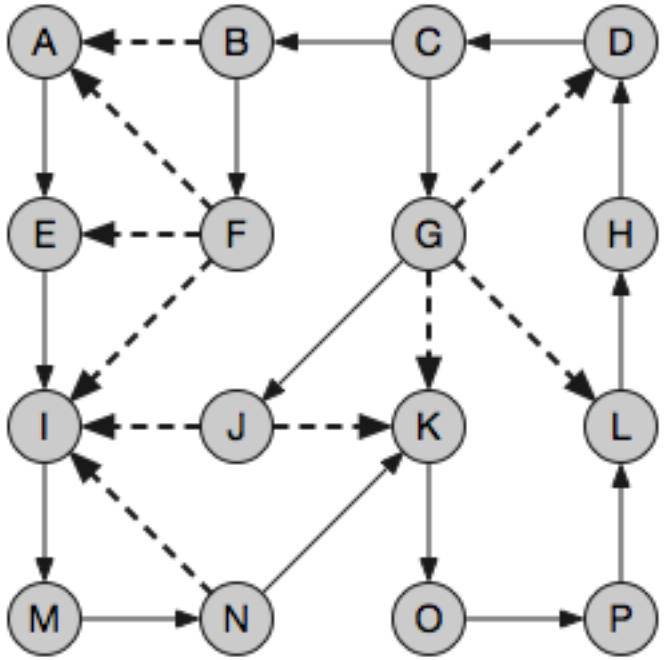
(c)



(d)



(e)



(f)

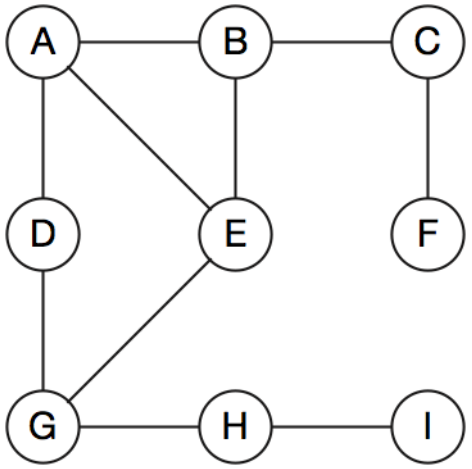
The depth-first search continued.

C++ Implementation

```
struct   Vertex
{    Vertex() : mark_(false) { }
     bool                    mark_;
     string              name_;
     vector<Vertex*>    neighborPtr_;
};

class    Graph
{
     public:
          void      AddEdge(Vertex* fromPtr, Vertex* toPtr);
          Vertex* AddVertex(const string& name);
          void      DepthFirstSearch();

     private:
          vector<Vertex*>     vertexPtr_;
};
```

```
void Graph::AddEdge(Vertex* fromPtr, Vertex* toPtr)
{
    fromPtr->neighborPtr_.push_back(toPtr);
    return;
}

Vertex* Graph::AddVertex(const string& name)
{
    Vertex* vertexPtr;
    vertexPtr = new Vertex;
    vertexPtr->name_ = name;
    vertexPtr_.push_back(vertexPtr);
    return(vertexPtr);
}
```

```
void Graph::DepthFirstSearch()
{
    Vertex*             neighborPtr;
    Vertex*             vertexPtr;
    stack<Vertex*>      vertexStack;
    vertexStack.push(vertexPtr_[0]);
    while (!vertexStack.empty())
    {
        vertexPtr = vertexStack.top();
        vertexStack.pop();
        cout << "Visit " << vertexPtr->name_ << endl;
        vertexPtr->mark_ = true;
        for (uint64_t i = 0; i < vertexPtr->neighborPtr_.size(); ++i)
        {
            neighborPtr = vertexPtr->neighborPtr_[i];
            if (!neighborPtr->mark_)
            {
                vertexStack.push(neighborPtr);
            }
        }
    }
    return;
}
```



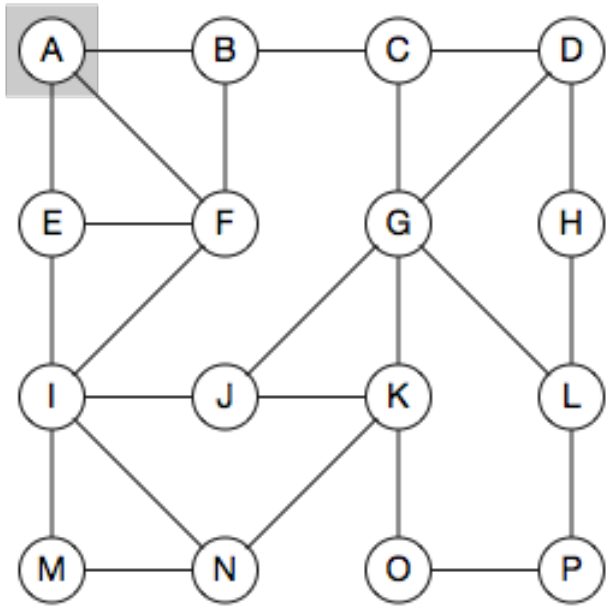
1	Push A	A
2	Pop A	<empty>
3	Visit A	<empty>
4	Push B	B
5	Push E	B, E
6	Push D	B, E, D
7	Pop D	B, E
8	Visit D	B, E
9	Push G	B, E, G
10	Pop G	B, E
11	Visit G	B, E
12	Push H	B, E, H
13	Pop H	B, E
14	Visit H	B, E

15	Push I	B, E, I
16	Pop I	B, E
17	Visit I	B, E
18	Pop E	B
19	Visit E	B
20	Pop B	<empty>
21	Visit B	<empty>
22	Push C	C
23	Pop C	<empty>
24	Visit C	<empty>
25	Push F	F
26	Pop F	<empty>
27	Visit F	<empty>
28	Nothing on stack to visit, done	

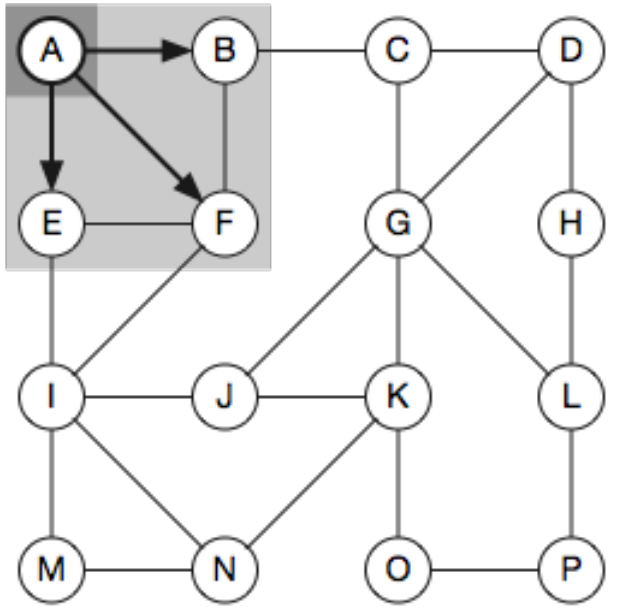
Breadth-First Traversal

- Another way to traverse a labyrinth with a string, a paint can, and a brush.
- Pay out the string in levels, each one the length of an edge.
- Visit the vertices that are one level away before going any deeper.
- Then go forward another level, rolling out one more edge length of string, deeper into the graph.
- Edges that connect to new unvisited vertices are called **discovery edges**—the heavy arrows in the diagrams that follow.
- Edges that connect to previously visited vertices are called **cross edges**—the dashed arrows in the diagrams that follow.

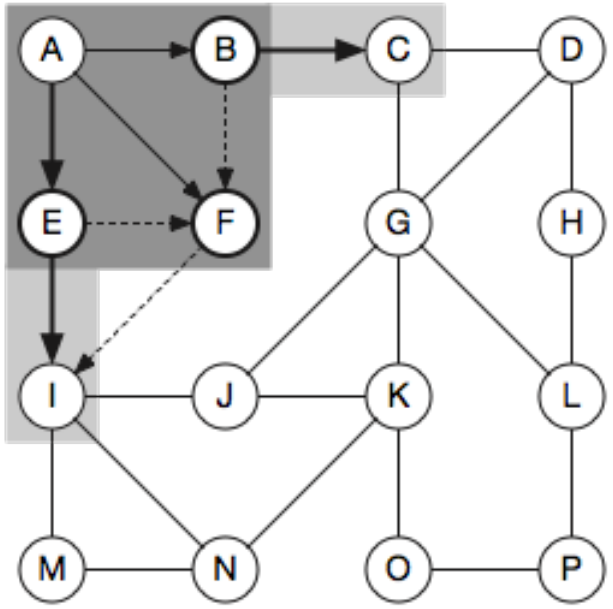
Breadth-first traversal



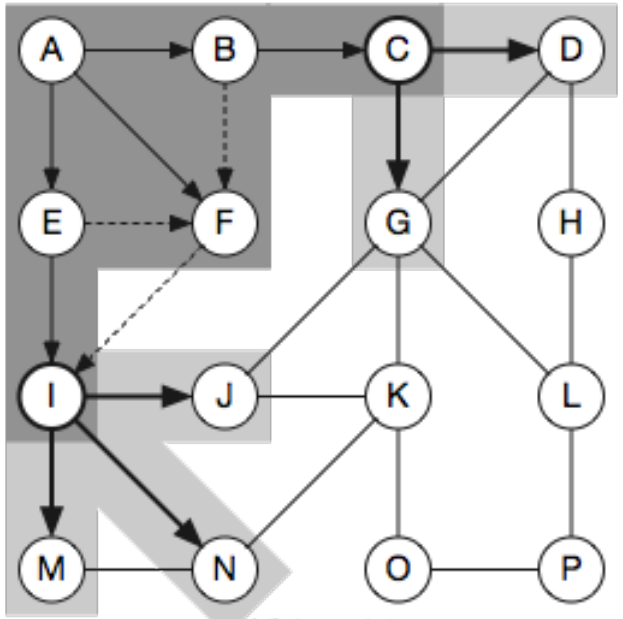
(a) Level 0



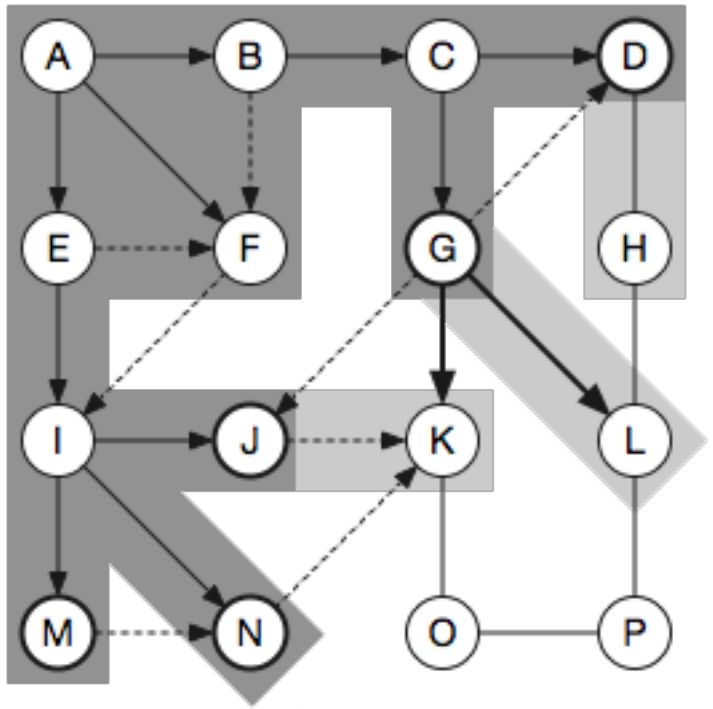
(b) Level 1



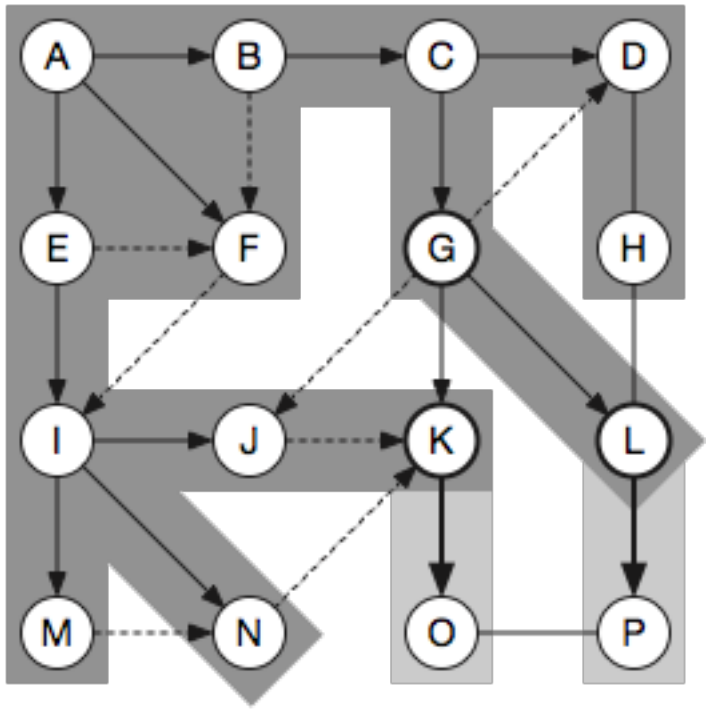
(c) Level 2



(d) Level 3

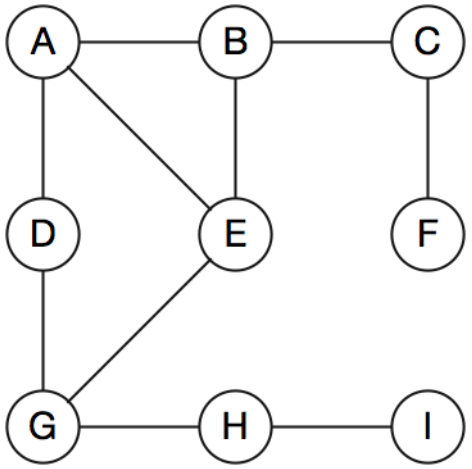


(e) Level 4



Breadth-first traversal, continued.

```
void Graph::BreadthFirstSearch()
{
	Vertex*		neighborPtr;
	Vertex*		vertexPtr;
	queue<Vertex*>		vertexQueue;
	vertexStack.push(vertexPtr_[0]);
	while (!vertexQueue.empty())
	{
		vertexPtr = vertexQueue.front();
		 vertexQueue.pop();
		if (vertexPtr->mark_)
		{
			cout << "Visit " << vertexPtr->name_ << endl;
			vertexPtr->mark_ = true;
			for (uint64_t i = 0; i < vertexPtr->neighborPtr_.size(); ++i)
			{
				neighborPtr = vertexPtr->neighborPtr_[i];
				if (!neighborPtr->mark_)
				{
					vertexQueue.push(neighborPtr);
				}
			}
		}
	}
}
```



1	Push A	A
2	Pop A	<empty>
3	**Visit A**	<empty>
4	Push B	B
5	Push E	B, E
6	Push D	B, E, D
7	Pop B	E, D
8	**Visit B**	E, D
9	Push C	E, D, C
10	Push E	E, D, C, E
11	Pop E	D, C, E
12	**Visit E**	D, C, E
13	Push G	D, C, E, G
14	Pop D	C, E, G
15	**Visit D**	C, E, G
16	Push G	C, E, G, G

17	Pop C	E, G, G
18	**Visit C**	E, G, G
19	Push F	E, G, G, F
20	Pop E (ignore)	G, G, F
21	Pop G	G, F
22	**Visit G**	G, F
23	Push H	G, F, H
24	Pop G (ignore)	F, H
25	Pop F	H
26	**Visit F**	H
27	Pop H	<empty>
28	**Visit H**	<empty>
29	Push I	I
30	Pop I	<empty>
31	**Visit I**	<empty>
32	Nothing in queue to visit, done	

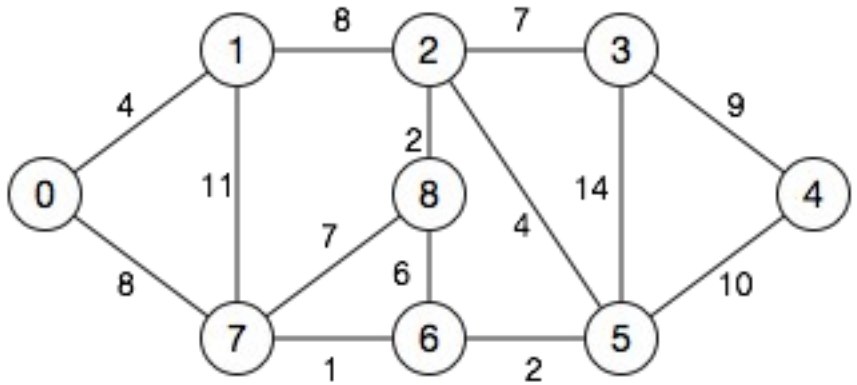
Dijkstra’s Algorithm: A Breadth-First Traversal

- Used to find the shortest paths through a graph—the basis of two Internet routing algorithms.
- The graph’s edges are labeled with the “cost” of taking that path.
- Build a “shortest path” graph, where each vertex is labeled with the cost of reaching that vertex from a given starting vertex.
- The initial graph has only the starting vertex with a cost label of 0.
- The graph is extended by adding neighbors, labeling them with costs, and advancing the search from the lowest cost node, which may be a previously added node, not a newly added neighbor.
- The example algorithm labels vertices but doesn’t track paths. That function was left out to focus on the traversal itself.

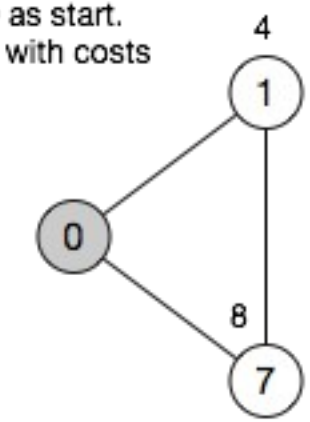
Repeat the following steps of the algorithm:

1. Choose the unvisited vertex with the lowest cost label and mark it “visited”. In the diagrams that follow, by shading in its circle.
2. Add the newly-added vertex’s neighbors to the graph. Some neighbors may already be in the graph because they are neighbors of another previously-added vertex ; if so, just add edges to those vertices.
3. Label each neighbor with the cost of the edge from that starting vertex. Some neighbors, the ones that were already in the graph, will have labels. If the new cost is lower, replace the label; otherwise, leave the label alone.
4. Add that vertex’s neighbors to the graph.
5. Repeat steps 1–4 until all vertices have been visited.

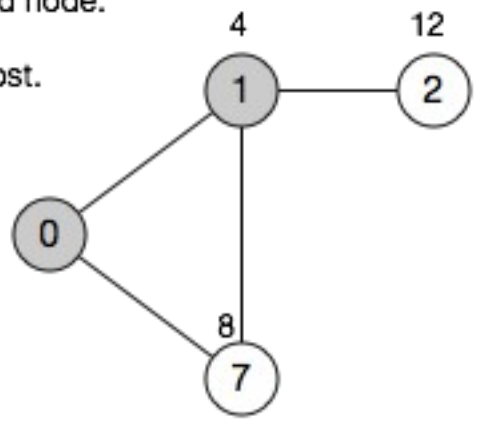
Note: On the first pass through the loop, the vertex chosen will be the starting vertex—it’s the only one in the graph. On subsequent passes, the chosen vertex can be any unvisited vertex, which is not necessarily one that was just added to the graph. It could be from a previous pass over the graph.



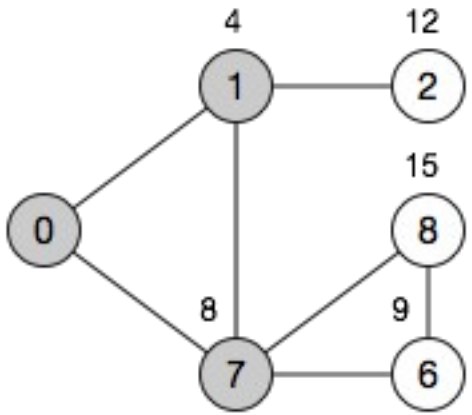
1. Select node 0 as start.
Label neighbors with costs



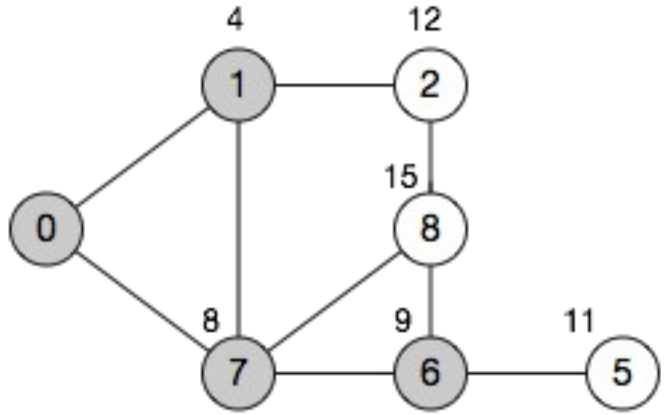
2. Select lowest cost unvisited node.
Label neighbors with costs.
Node 7 retains its previous cost.



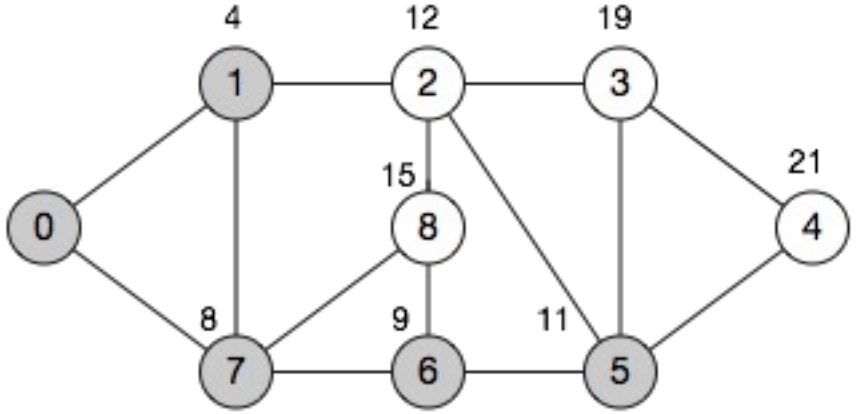
3. Select lowest cost node 7.
Label neighbors 6 and 8 with costs.
Node 1 retains its previous cost



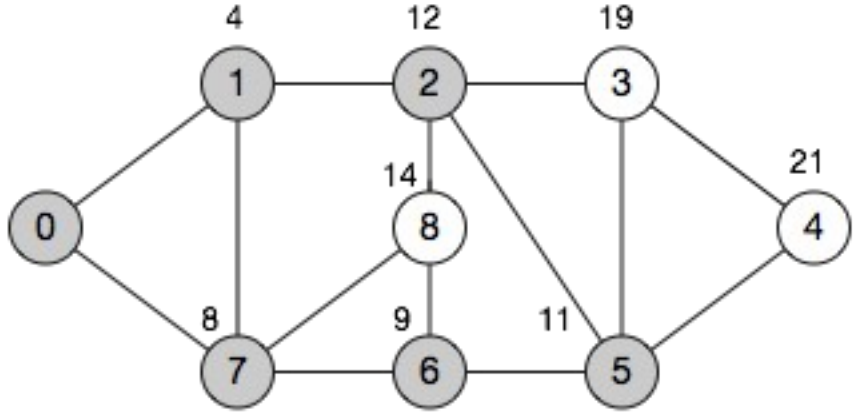
4. Select lowest cost node 6.
Label neighbor 5 with cost.
Nodes 7 and 8 retain their previous costs



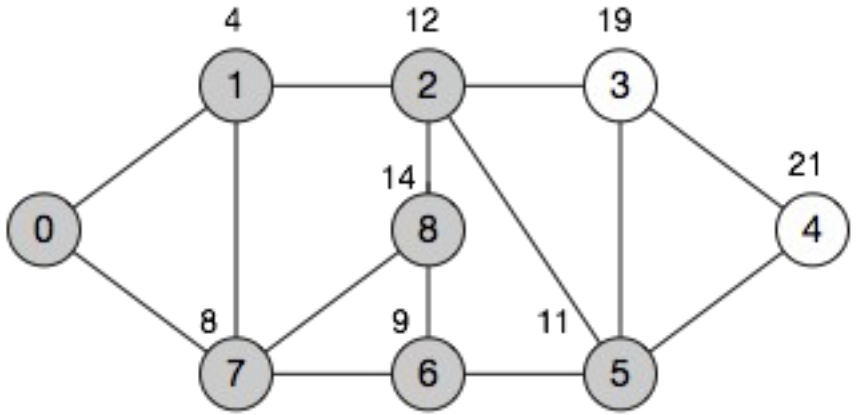
5. Select lowest cost node 5.
Label neighbors 3 and 4 with costs.
Nodes 2 and 6 retain their previous costs.



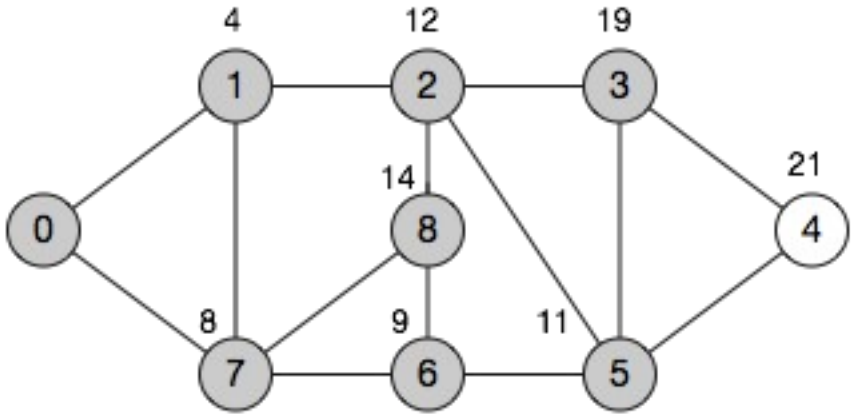
6. Select lowest cost node 2.
Nodes 1 and 3 retain their previous costs.
Node 8 gets a new lower cost of 14.



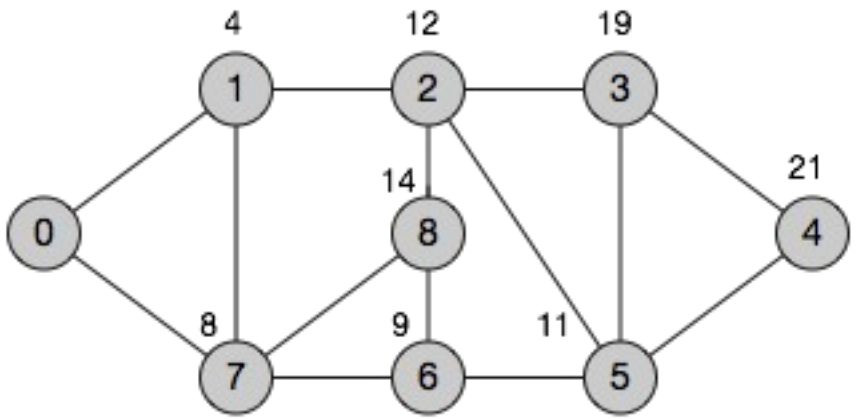
7. Select lowest cost node 8.
All neighbors, 2, 6, and 7, retain their previous costs.



8. Select lowest cost node 3.
Nodes 2, 4, and 5 retain their costs.



9. Select node 4.
Nodes 3 and 5 retain their costs.



10. There are no more unvisited nodes.
The algorithm terminates.