Data Structures & Algorithms 2

Topic 7 – Graphs (part 3)

Lecturer: Dr. Hadi Tabatabaee
Materials: Dr. Phil Maguire & Dr. Hadi Tabatabaee
Maynooth University
Online at http://moodle.maynoothuniversity.ie

Overview

Aims

- Introducing graph search approaches

Learning outcomes: You should be able to…

- Learn different graph search methods
- Learn to implement a breadth-first search method

Searching (revisit)

One of the most useful operations you can perform on your graph is searching.

For example, find all the towns that can be visited by rail, leaving Dublin.

There are two very well-known approaches for searching a space

- Depth-first search (DFS)
- Breadth-first search (BFS)

- Depth-first search explores one possibility as far as it can and then backtracks when it meets a dead end.

- Breadth-first search explores all possibilities of the same depth at the same time and spreads itself equally.

Breadth-First Search

Depth-first search acts like it wants to get as far from the starting point as quickly as possible.

Breadth-first search likes to stay as close as possible to the starting point.

- It visits all possibilities at the same depth before going any deeper
- It ventures one vertex from the origin, then two, then three, etc.

Breadth-first search can be implemented using a queue instead of a stack.

Breadth-First Search

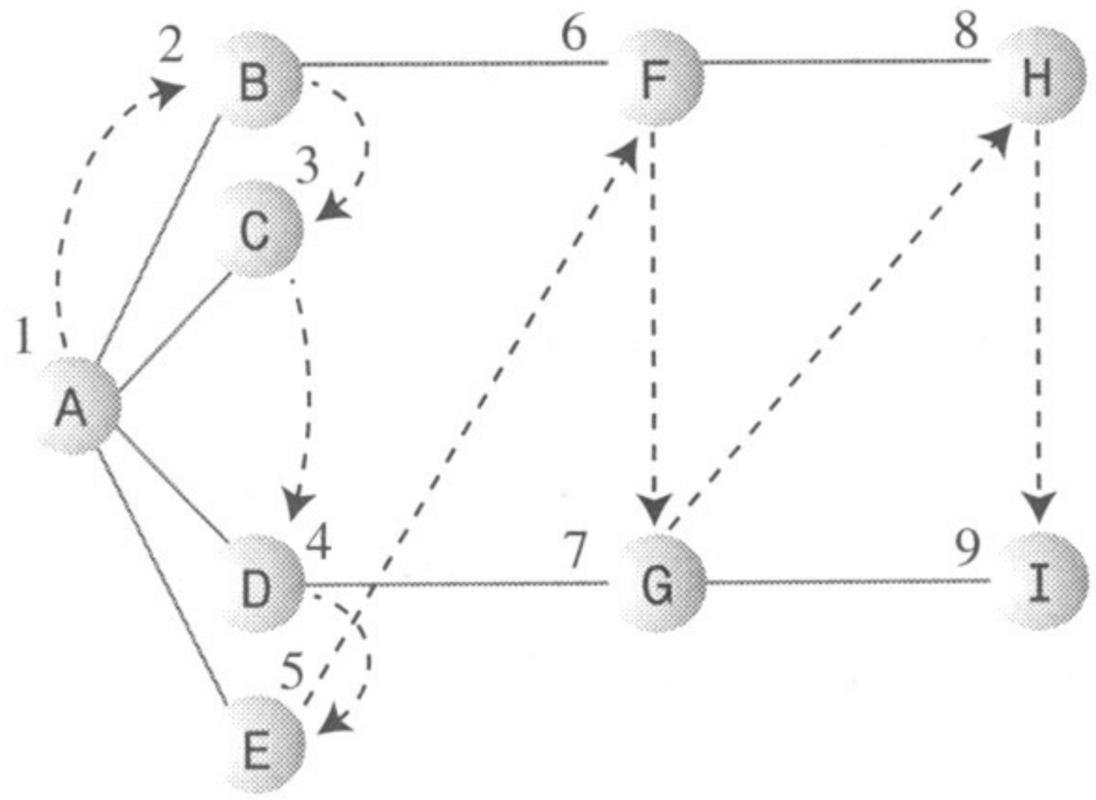
The rules are:

- Visit the next unvisited vertex (if there is one) adjacent to the current vertex, mark it and add it to the back of the queue.
- Only when you have visited all adjacent vertices, remove a vertex from the queue and make it the current one.

Breadth-First Search

TABLE 13.4 Queue Contents During Breadth-First Search

Event	Queue (Front to Rear)
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	



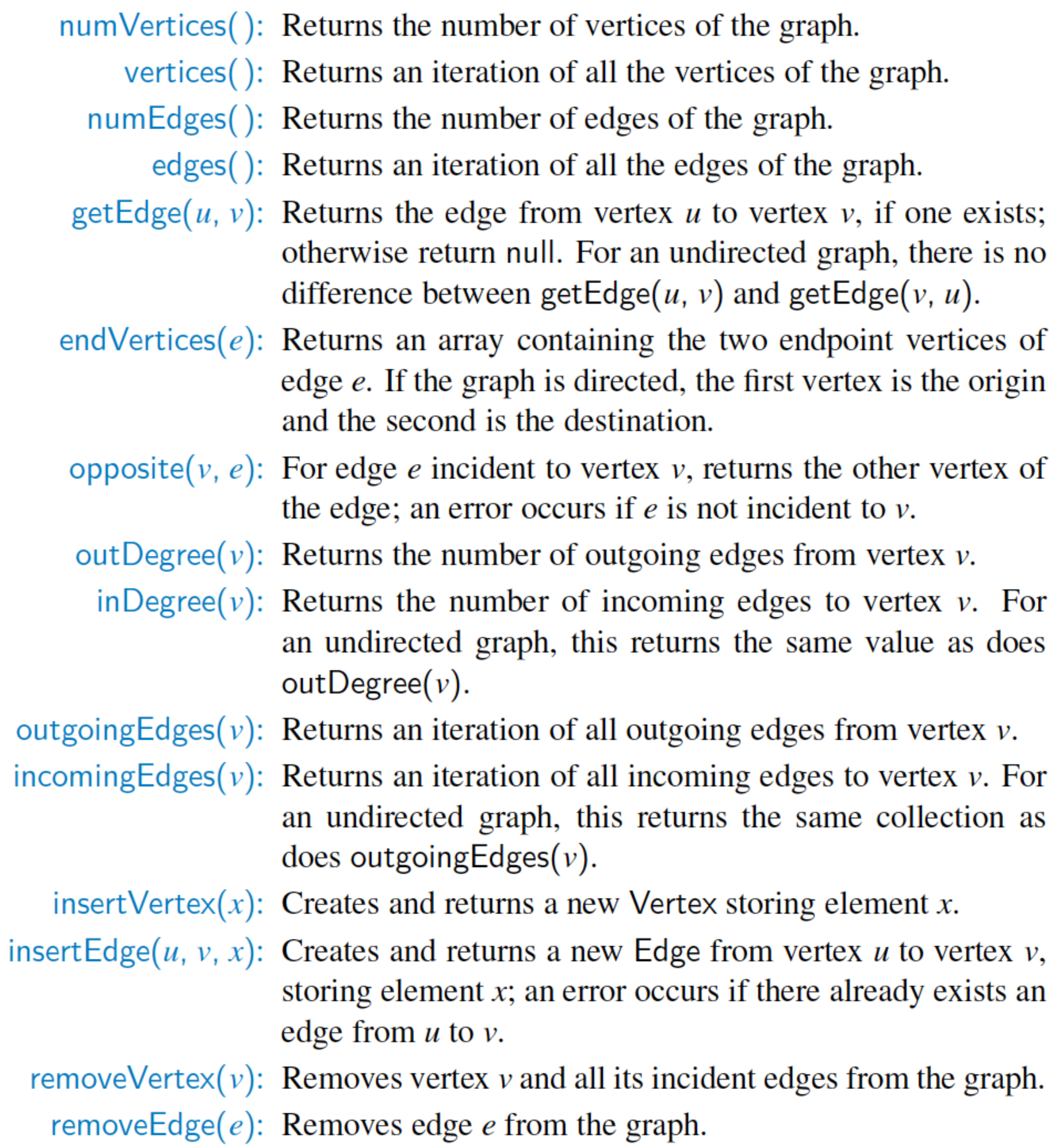
Implementing BFS

- Again, check all the adjacent vertices using the adjacency matrix.
- Insert them to the back of the queue
- Keep removing one item from the queue and inserting all its adjacent vertices to the back.
- This way, all the vertices at one particular depth will be explored before you move onto a vertex at the next depth

Analogy

- Breadth-first search is like the ripples spreading out on a pond.
- The ripples cover all the search space and move further and further away from the origin at the same rate.
- Breadth-first search is used when you want to explore all of the possibilities at a given depth.

Graph ADT
(revisit)

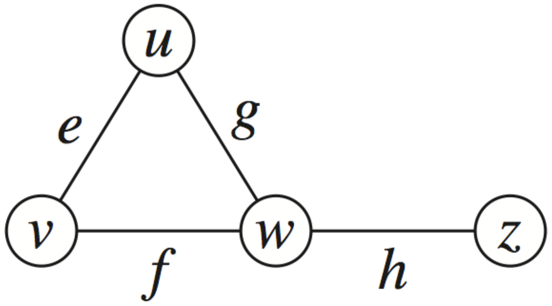


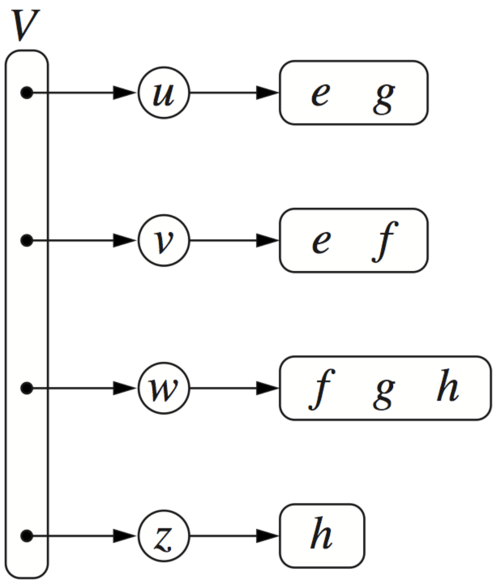
numVertices( ):	Returns the number of vertices of the graph.
vertices( ):	Returns an iteration of all the vertices of the graph.
numEdges( ):	Returns the number of edges of the graph.
edges( ):	Returns an iteration of all the edges of the graph.
getEdge(u, v):	Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
endVertices(e):	Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
opposite(v, e):	For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
outDegree(v):	Returns the number of outgoing edges from vertex v.
inDegree(v):	Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
outgoingEdges(v):	Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v):	Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
insertVertex(x):	Creates and returns a new Vertex storing element x.
insertEdge(u, v, x):	Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeVertex(v):	Removes vertex v and all its incident edges from the graph.
removeEdge(e):	Removes edge e from the graph.

Adjacency List Structure (revisit)

The adjacency list structure for a graph adds extra information to the edge list structure that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex.

- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



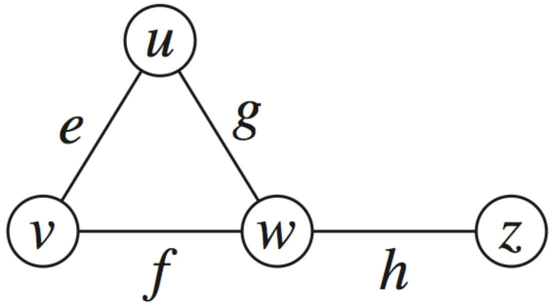


Adjacency Matrix Structure (revisit)

The adjacency matrix is a two-dimensional array in which the elements indicate whether an edge is present between two vertices.

If a graph has N vertices, then the adjacency matrix is an N x N array.

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non-nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge



$$
\begin{array}{ccc|c|c|c|c|}
 & & & 0 & 1 & 2 & 3 \\
\hline
u & \longrightarrow & 0 & & e & g & \\
\hline
v & \longrightarrow & 1 & e & & f & \\
\hline
w & \longrightarrow & 2 & g & f & & h \\
\hline
z & \longrightarrow & 3 & & & h & \\
\hline
\end{array}
$$

Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G.
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems.
  - Find and report a path with the minimum number of edges between two given vertices.
  - Find a simple cycle, if there is one.

BFS Algorithm

The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

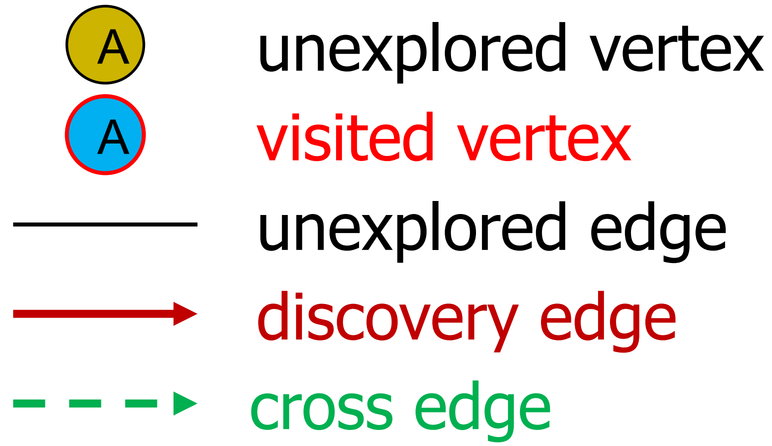
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

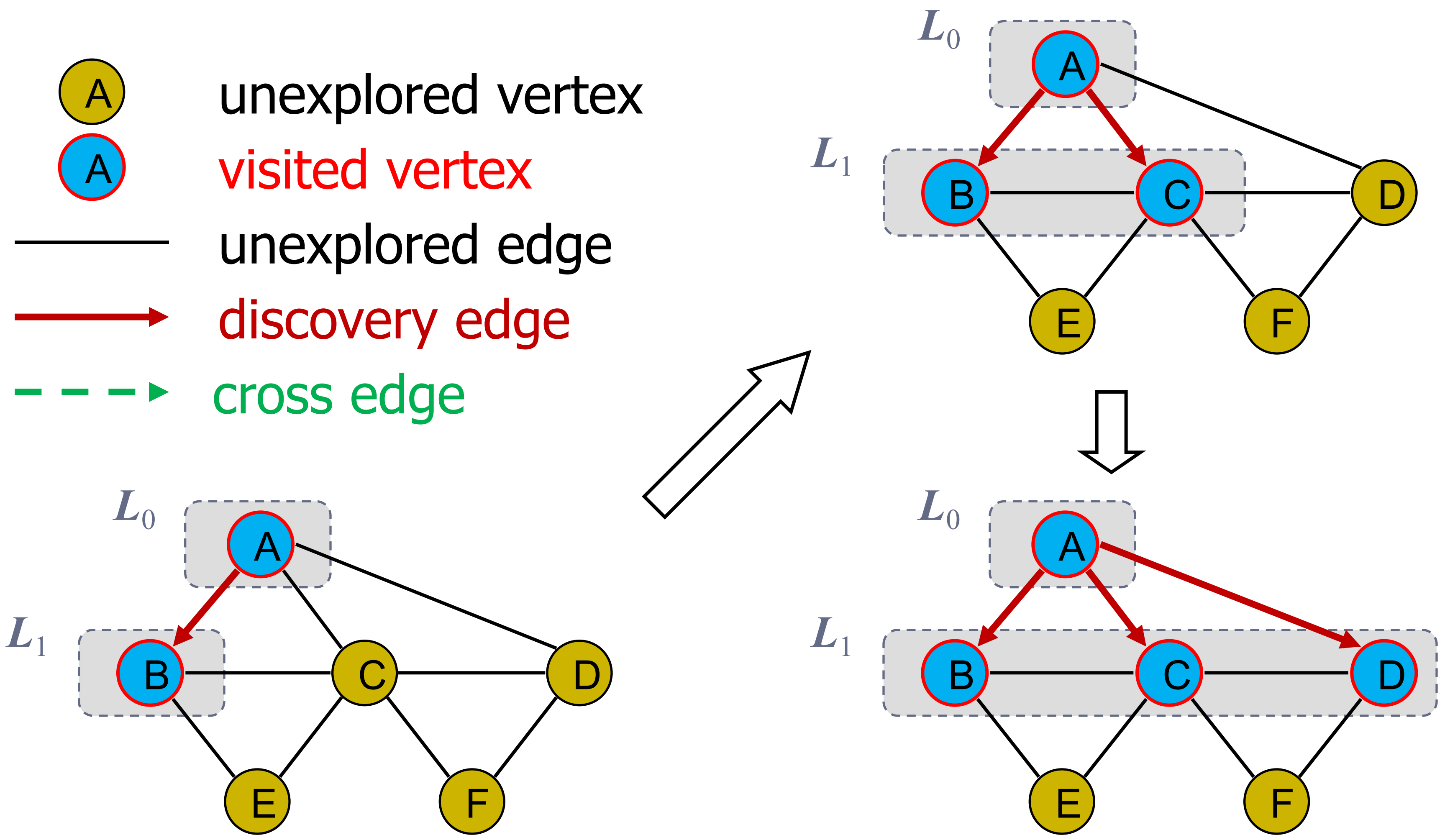
```
Algorithm BFS(G, s)
  $L_0$ ← new empty sequence
  $L_0$.addLast(s)
  setLabel(s, VISITED)
  $i$ ← 0
  while ¬$L_i$.isEmpty()
    $L_{i+1}$ ← new empty sequence
    for all v ∈ $L_i$.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            $L_{i+1}$.addLast(w)
          else
            setLabel(e, CROSS)
    $i$ ← $i$ +1
```

Java Implementation

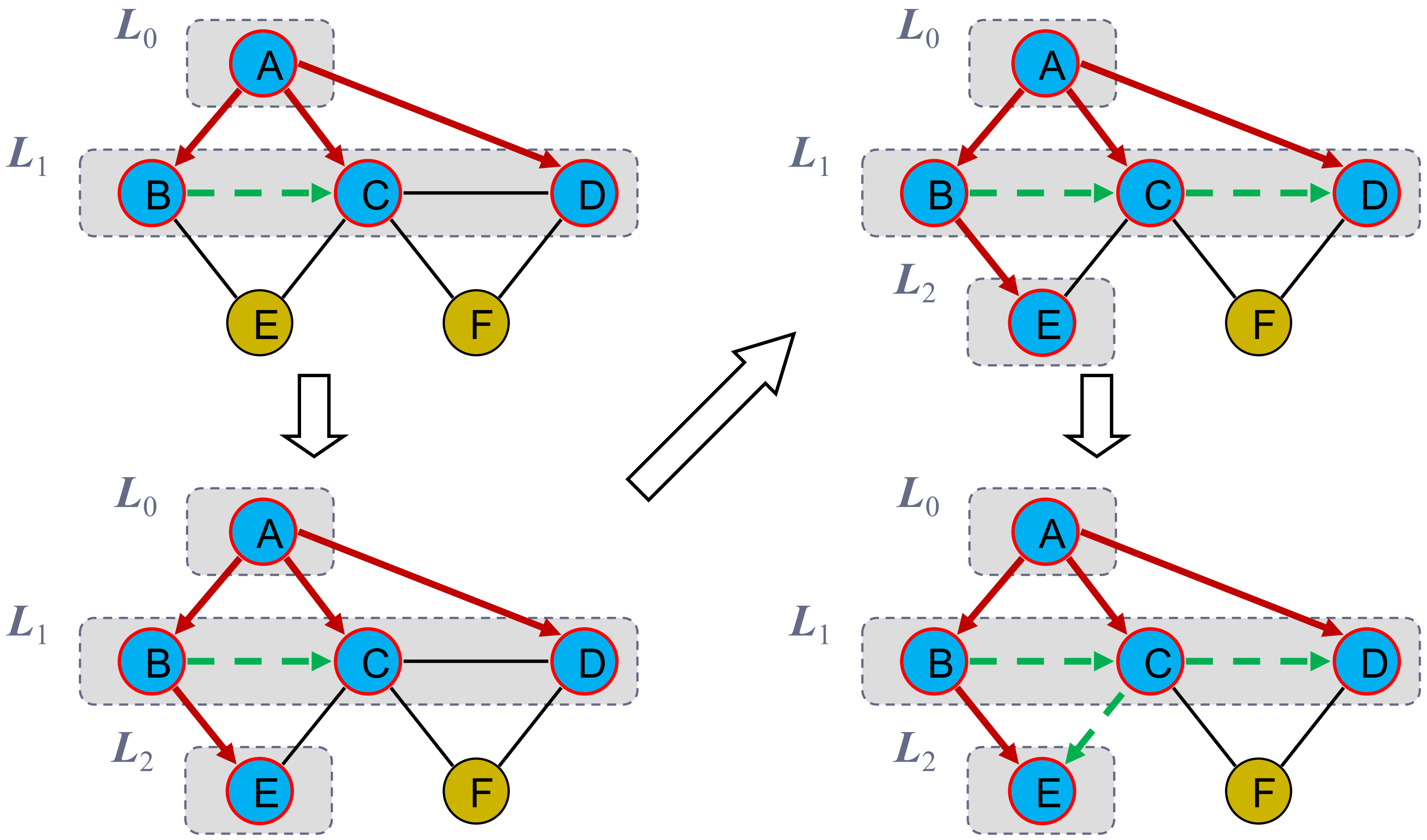
```
/** Performs breadth-first search of Graph g starting at Vertex u. */
public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
                    Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Vertex<V>> level = new LinkedPositionalList<>( );
  known.add(s);
  level.addLast(s);                         // first level includes only s
  while (!level.isEmpty( )) {
    PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>( );
    for (Vertex<V> u : level)
      for (Edge<E> e : g.outgoingEdges(u)) {
        Vertex<V> v = g.opposite(u, e);
        if (!known.contains(v)) {
          known.add(v);
          forest.put(v, e);                 // e is the tree edge that discovered v
          nextLevel.addLast(v);             // v will be further considered in next pass
        }
      }
    level = nextLevel;                      // relabel 'next' level to become the current
  }
}
```

Example

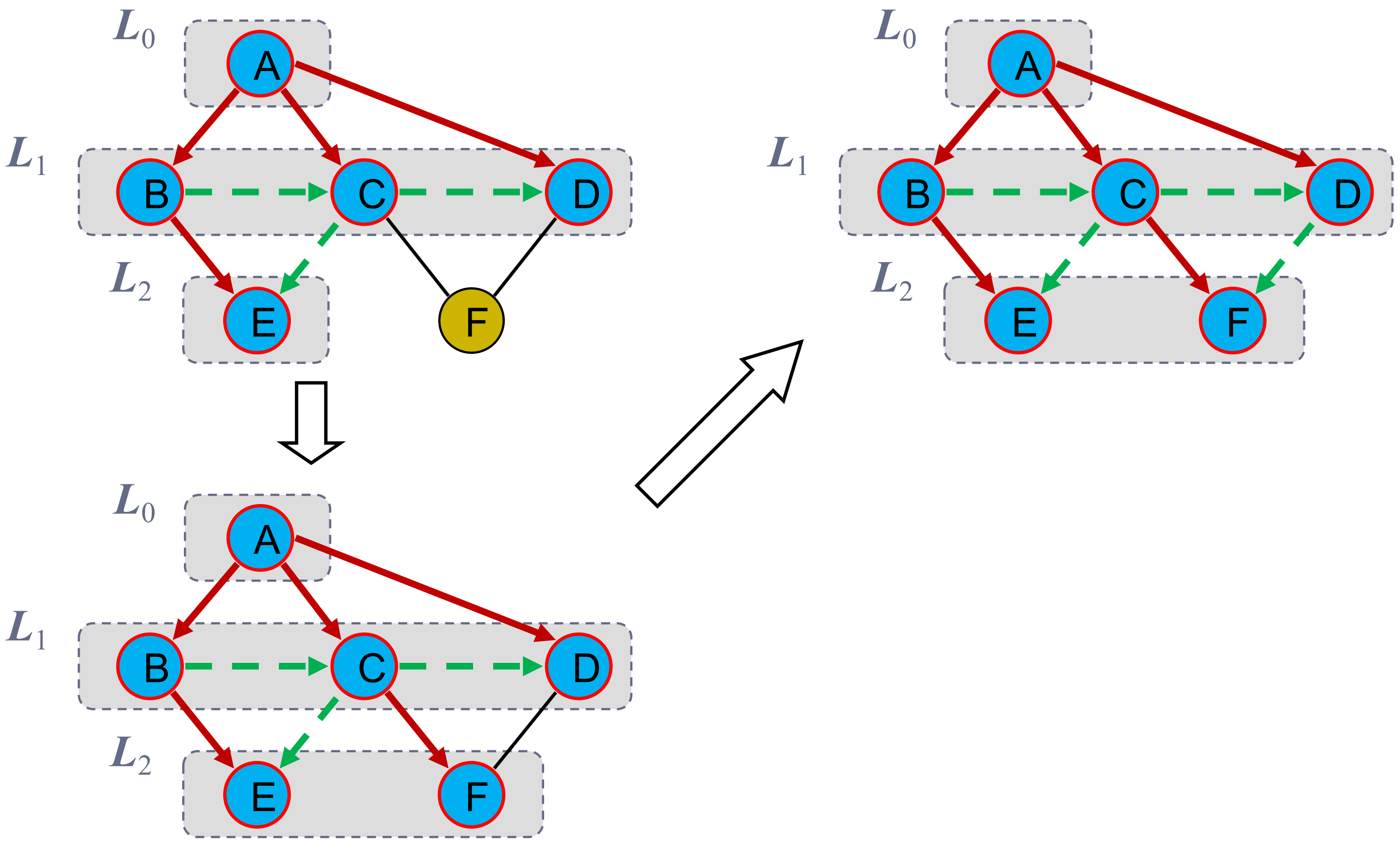

A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge


L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F

Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

BFS(G, s) visits all the vertices and edges of G_s

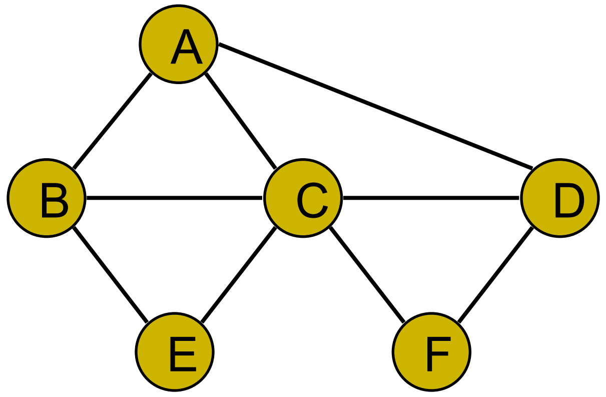
Property 2

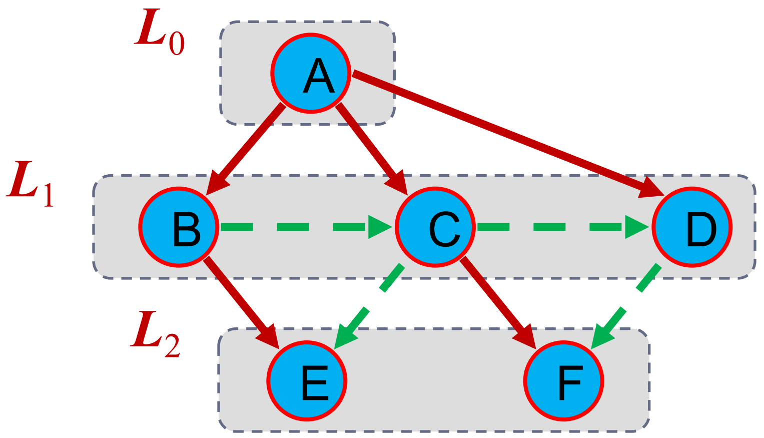
The discovery edges labeled by BFS(G, s) form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges.





Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Self-test

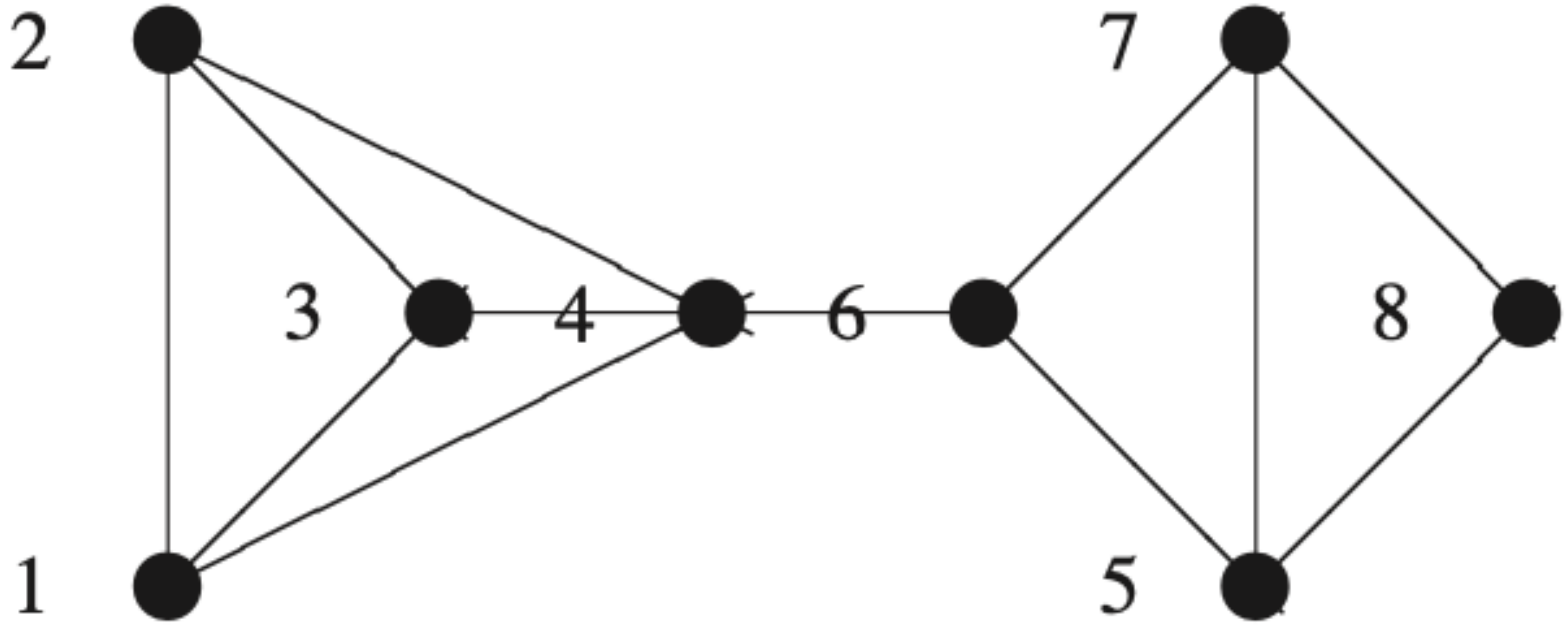
Let G be an undirected graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given:

vertex	adjacent vertices
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the table above.

1. Draw G.
2. Give the sequence of vertices visited using a BFS traversal starting at vertex 1.

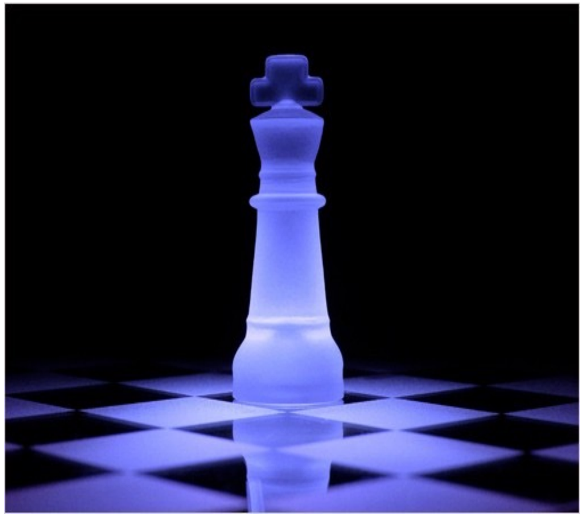
Solution



Sequence of visited vertices: 1, 2, 3, 4, 6, 5, 7, 8.

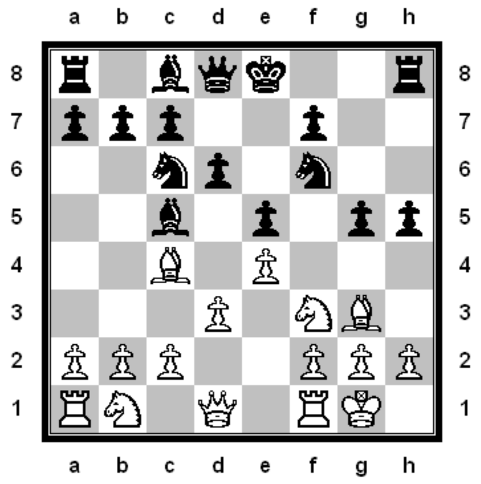
BFS in Games

- In a typical board game, you can choose one of several possible moves.
- Each choice leads to further choices leading to an ever-expanding tree of possibilities.
- We can represent each state of the board as a vertex and each choice as an edge leading to another board state.
- A smart algorithm will search forward through the possibilities to see what could potentially happen based on each choice.



Game trees

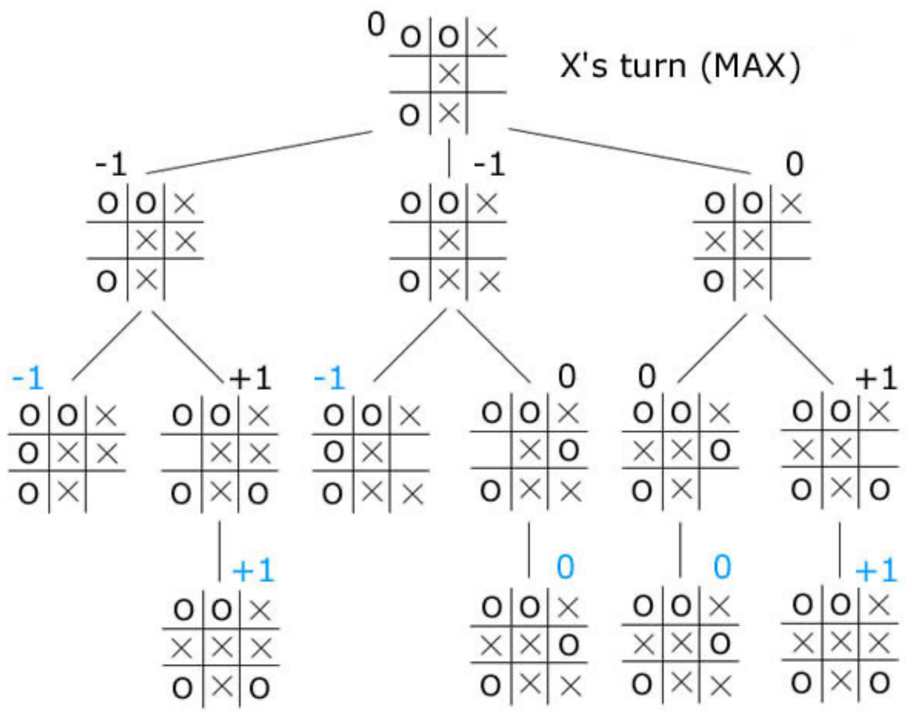
- Because you can’t predict what move the person you’re playing against will make, the trick is to look at every sequence of moves that *could* be made.
- A simple chess-playing algorithm might look at all the possible game states four moves ahead.
- A depth-first search to the endgame is inappropriate because we’re not trying to find a full solution to the game – we cannot know how the opponent will choose to move.
- Instead, we want to consider all of the possibilities and a certain number of moves ahead.



Minimax algorithm

Minimax is a decision rule used in decision theory, game theory, and statistics for minimizing the possible loss of a worst-case scenario.

We assume the opponent will make the best possible move and choose the least good best move option.

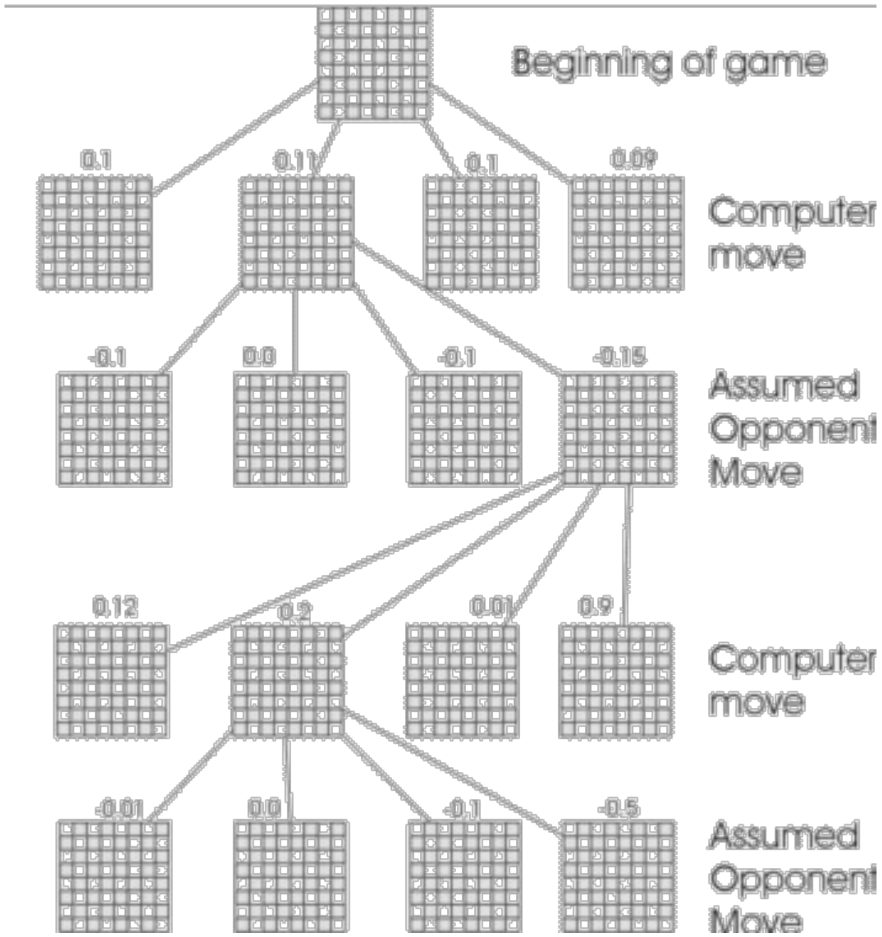


Artificial Intelligence

- If the computer knows all the possible game states four moves ahead, then it can choose the move that stops you from being able to make a good move.
- Knowing what can happen up to four moves ahead makes the computer opponent far more informed than the human opponent.
- Naïve users will usually only be thinking in terms of “if I make this move, then what moves can the computer make”
- This is only a breadth-first search of depth two.

Board Game Search Tree

- Each state of the board game can lead to several other states
- In chess, the branching factor is about 35
- To play master-level chess requires searching about 8 moves ahead, so about 2×10^{12} options must be examined.



Deep Blue

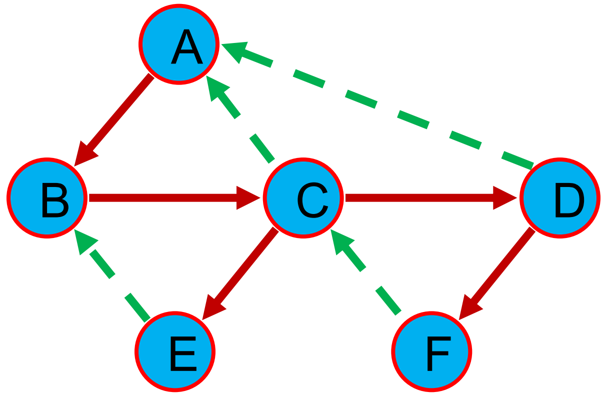
- On May 11, 1997, the Deep Blue computer won a six-game match by two wins to one and three draws against world champion Garry Kasparov.
- The computer was capable of evaluating 200 million positions per second
- It could search up to 40 moves ahead
- Kasparov said that he saw deep intelligence and creativity in the machine's moves and was sure that human players had intervened.

Applications

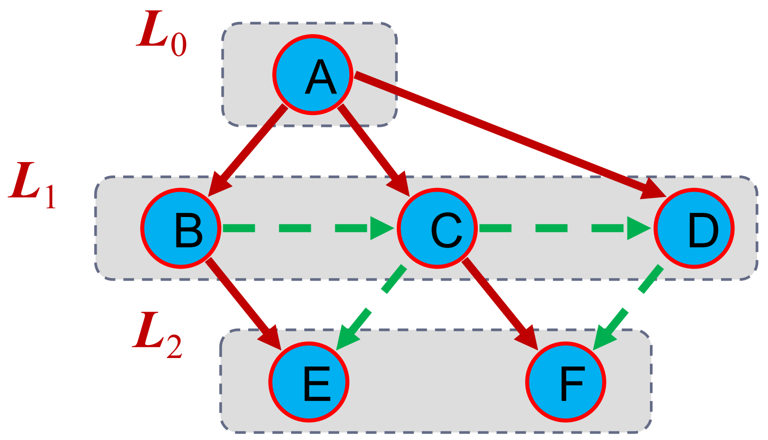
- We can specialize the BFS traversal of a graph G to solve the following problems in O(n + m) time.
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS



BFS

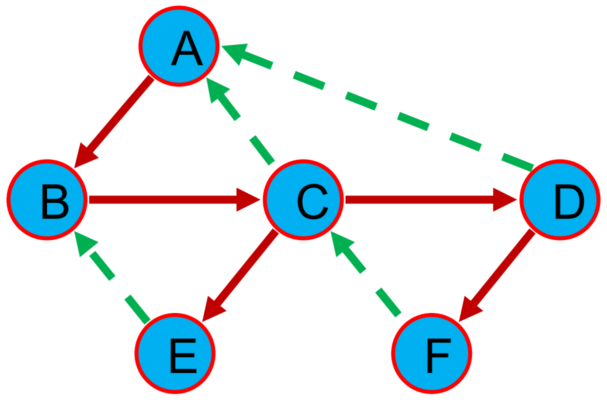
DFS vs. BFS (cont.)

Back edge (v,w)

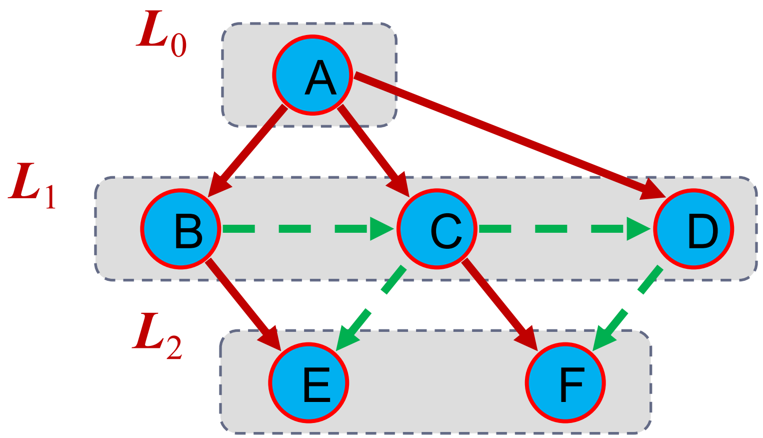
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

- w is in the same level as v or in the next level



DFS



BFS

Questions