Breadth First Search

Lecture 8

Graph Exploration

- Given a graph and two specified vertices r and v, a fundamental question is:
  - *Does the graph contain a path from r to v?*
- In fact, given a root vertex r, we will be able to answer this question simultaneously for all non-root vertices.
- That is, we can efficiently determine which vertices are reachable by **path traversal** starting at the root vertex.

Search Applications

- Graph Search has a **vast** number of applications.
- These can be simple applications:
  - *How do we explore a maze?*
  - *How do I travel from Montreal toTimbuktu?*
- Or, they can be much more substantive.
  - *What is the best way to search the web?*
  - *How, and how far, will a disease spread through the population?*

The Generic Search Algorithm

- Here is the generic graph search algorithm:

The Search Algorithm

Put *r* into a bag

While the bag is non-empty

Remove *v* from the bag

If *v* is unmarked

Mark *v*

For each arc *(v,w)*

Put *w* into the bag

The Generic Search Algorithm [cont.]

- Here is a slightly more formal pseudocode description.

```
Search(r)
Set B = {r}
    While B ≠ ∅
        Let v ∈ B
        Set B ← B \ {v}
        If v is unmarked
            Mark v
            For each arc (v,w)
                Set B ← B ∪ {w}
```

The Search Algorithm

Put r into a bag

While the bag is non-empty

Remove v from the bag

If v is unmarked

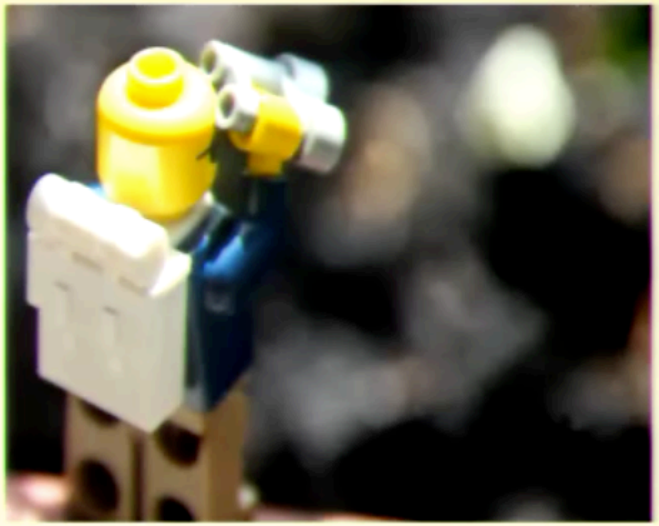
Mark v

For each edge (v,w)

Put w into the bag

- We can view a vertex as having been discovered when it is **marked**.
- We will see that if the graph is connected then *every* vertex is discovered.
- But first observe that there are one **strange** thing about this algorithm.
  - We add a vertex to the bag when it is the end-point of an examined edge.

    $\Longrightarrow$ Multiple copies of a vertex may be in the bag!
- Surprisingly this will not affect the performance of the algorithm.
  - *Indeed, this will actually be useful to us if we modify the algorithm to account for this by "bagging" edges instead of vertices...*



The (Revised) Generic Search Algorithm

- This gives the following *updated* generic graph search algorithm:

The Search Algorithm

Put $(*, r)$ into a bag

While the bag is non-empty

Remove (u,v) from the bag

If v is unmarked

Mark v

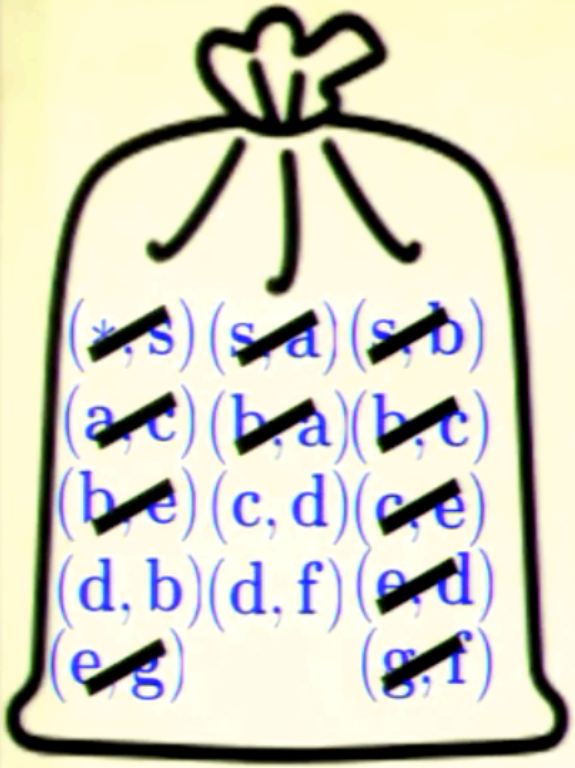
Set $p(v) \leftarrow u$

For each arc (v,w)

Put (v,w) into the bag

Keep track of the predecessor.

Search Example



Key

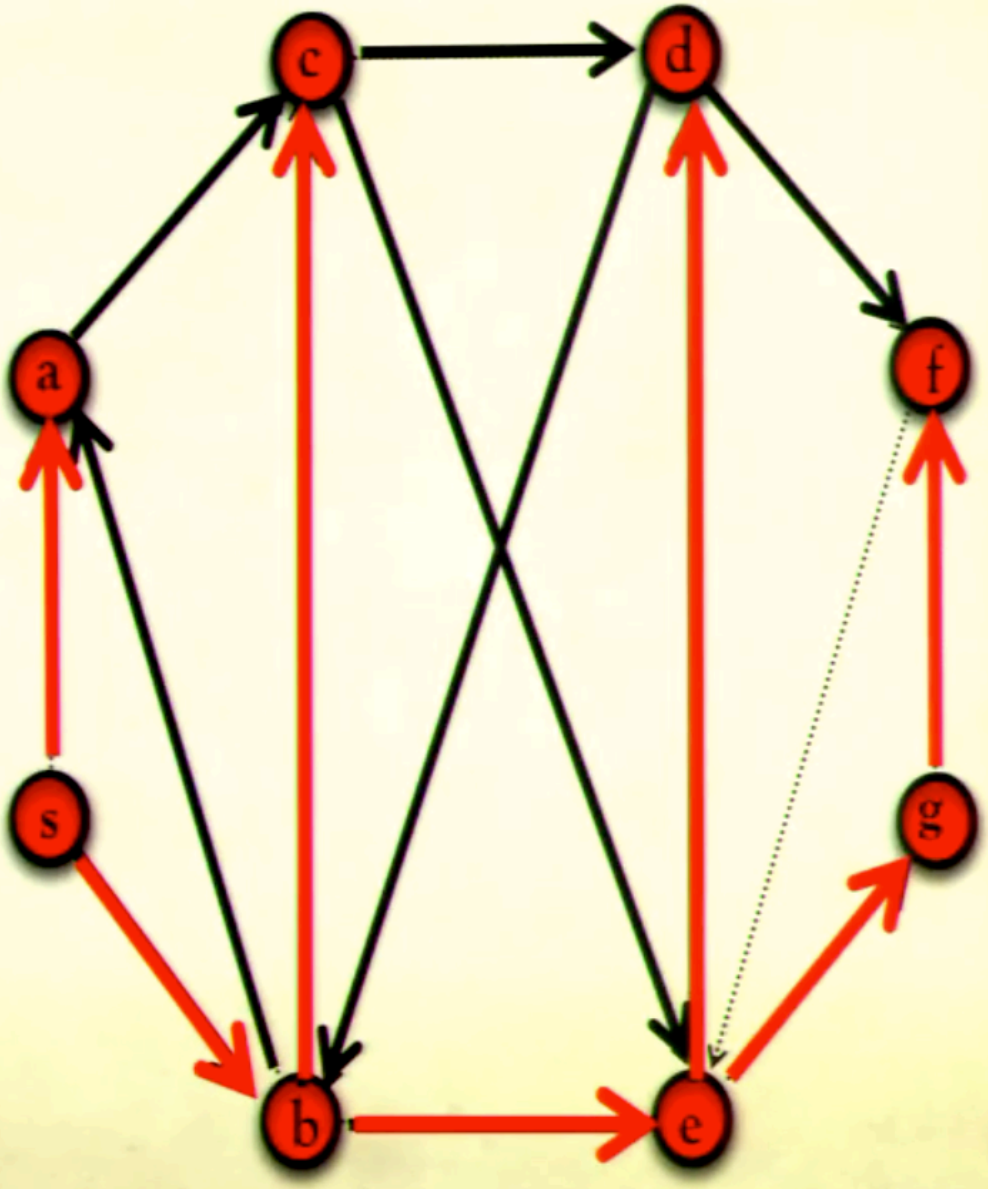
Unmarked Vertex

Marked Vertex

Unviewed Arc

Viewed Arc (In Bag)

Parent Arc



The Search Algorithm

Put $(*, r)$ into a bag

While the bag is non-empty

Remove (u,v) from the bag

If v is unmarked

Mark v

Set $p(v) \leftarrow u$

For each arc (v,w)

Put (v,w) into the bag

The Running Time

- What is the *running time* of the generic search algorithm?
- We search each arc out of v only **once**, when v is first marked.
- The arc is then added to the bag once and later removed from the bag once.

$$\Longrightarrow \quad \text{Runtime} = O(m)$$

- So this is a **linear time** algorithm.

The Search Algorithm

```
Put (*, r) into a bag
  While the bag is non-empty
      Remove (u,v) from the bag
      If v is unmarked
          Mark v
          Set p(v) ← u
          For each edge (v,w)
              Put (v,w) into the bag
```

Validity

Theorem. Let G be a connected, undirected graph. Then the search algorithm finds every vertex in G.

Proof.

- We must show that each vertex v is **marked** by the search algorithm.
- We prove this by induction on the smallest number k of edges in a path from the vertex to the root.

Base Case: $k = 0$

- Then v is the root vertex r.
- But r is the first vertex marked in the algorithm, so the base case holds.

Proof [cont].

Induction Hypothesis:

- Assume any vertex v that has a path of k-1 (or fewer) edges to the root r is marked.

Induction Step:

- Assume there is a path P with k edges from v to r. Specifically let

$$P = \{v = v_k, v_{k-1}, \ldots, v_1, v_0 = r\}$$

- Thus there is a path Q with k-1 edges from $u = v_{k-1}$ to r. That is,

$$Q = \{u = v_{k-1}, \ldots, v_1, v_0 = r\}$$

- So, by the induction hypothesis, the vertex u is marked.
- After we mark u, we place the edges incident to it in the bag.

  $\Longrightarrow$ The edge (u, v) is added to the bag.

- Thus, when (u, v) is removed from the bag we will **mark** v (if it is not already marked).

□

The Search Algorithm

Put (*, r) into a bag

While the bag is non-empty

Remove (u,v) from the bag

If v is unmarked

Mark *v*

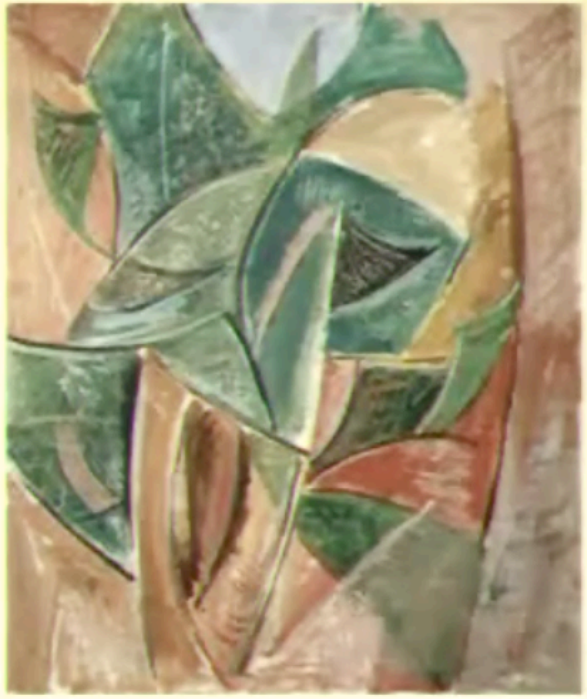
Set $p(v) \leftarrow u$

For each arc (*v*,*w*)

Put (*v*,*w*) into the bag

Validity in Directed Graphs

- For directed graphs, a similar argument proves that each vertex that has a directed path to it from the root *r* is **marked**.



Search Trees

- Observe that each non-root vertex has exactly one predecessor.

Theorem. Let G be a connected, undirected graph. Then the predecessor edges form a tree rooted at r.

Proof.

- By induction on the number k of marked vertices.

Base Case: $k = 1$

- The root vertex r is the first vertex marked.

  $\Longrightarrow$ *Trivially*, we have a tree rooted at r, so the base case holds.

Proof [cont].

Induction Hypothesis:

- Assume the predecessor edges for the first k-1 marked vertices form a tree rooted at r.

Induction Step:

- Let v be the kth vertex to be marked.
- Assume v was marked when we removed the edge (u, v).
  $\Longrightarrow u = p(v)$
- But (u, v) was added to the bag when we marked vertex u.
  $\Longrightarrow$ Vertex u is in the set S of the first k-1 vertices to be marked.
- By the induction hypothesis, the predecessor edges for S form a tree T rooted at r.

$\Longrightarrow$ $T \cup (p(v), v)$ is a tree rooted at r on the first k marked vertices. □

The Search Algorithm

```
Put (*, r) into a bag
  While the bag is non-empty
      Remove (u,v) from the bag
    If v is unmarked
        Mark v
        Set p(v) ← u
        For each edge (v,w)
            Put (v,w) into the bag
```

Implementational Aspects

- There is some flexibility in how to implement the search algorithm?
  - If the bag contains many arcs, which one should we remove?
- In fact, what in computer science is a "bag" ?
- The "bag" is just short-hand for a data structure.
- *Moreover, the choice of data structure used has important consequences...*

Data Structures

- Three choices of **data structure** for the bag give fundamental algorithms:

Queue.

Breadth First Search.

Stack.

Depth First Search.

Priority Queue.

Minimum Spanning Tree Algorithm.

Breadth First Search

- Using a **Queue** (FIFO) data structure produces:

Breadth First Search Algorithm

Add (*, r) to a Queue

While the Queue is non-empty

Remove the first arc (u,v) from the Queue

If v is unmarked

Mark v

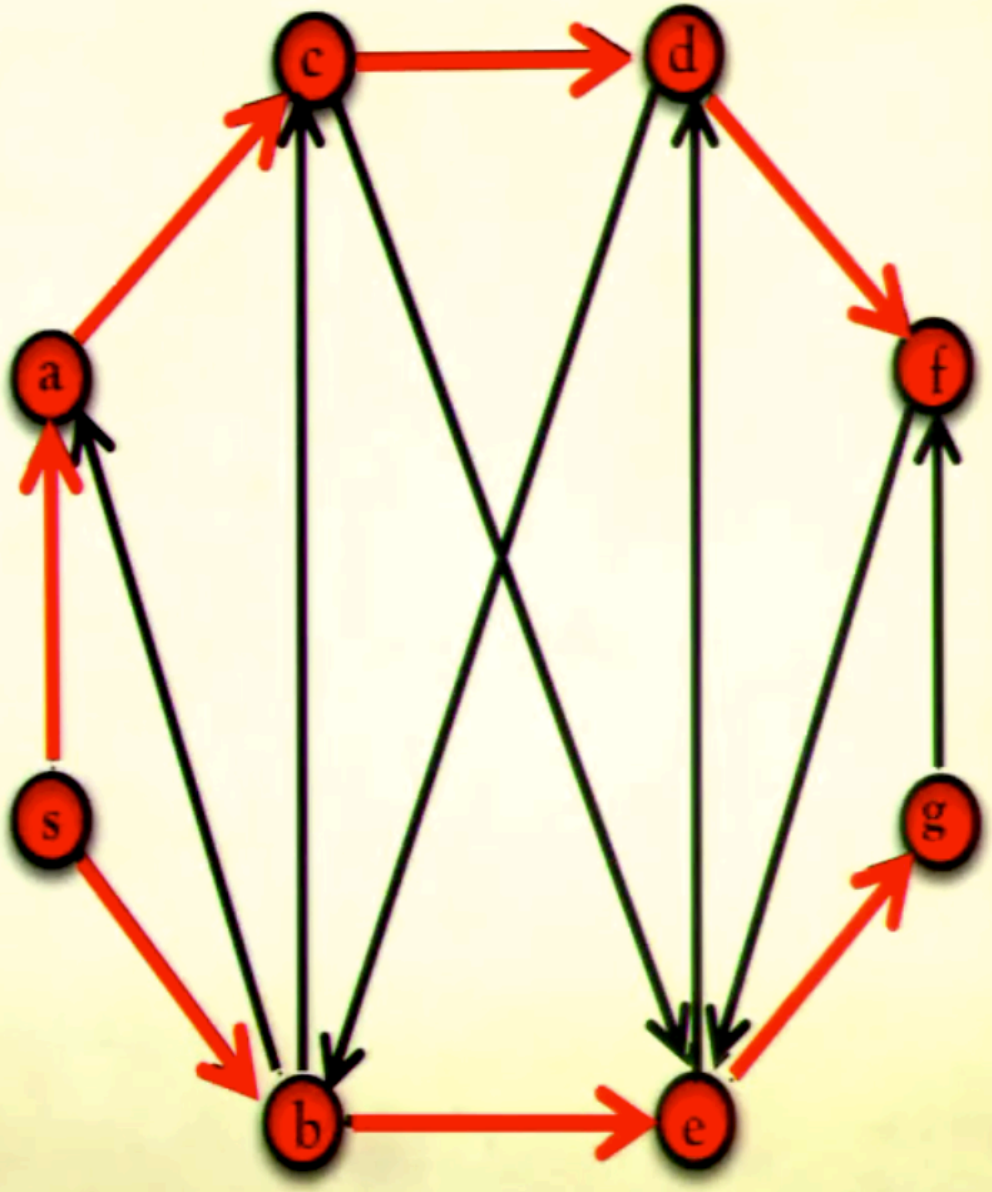
Set $p(v) \leftarrow u$

For each arc (v,w)

Add (v,w) into the back of the Queue

(∗, s)(s, a)(s, b)(a, c)(b, a)(b, c)(b, e)(c, d)(c, e)(e, d)(e, g)(d, b)(d, f) (g, f) (f, e)

BFS Example



Key

Unmarked Vertex

Marked Vertex

Unviewed Arc

Viewed Arc (In Bag)

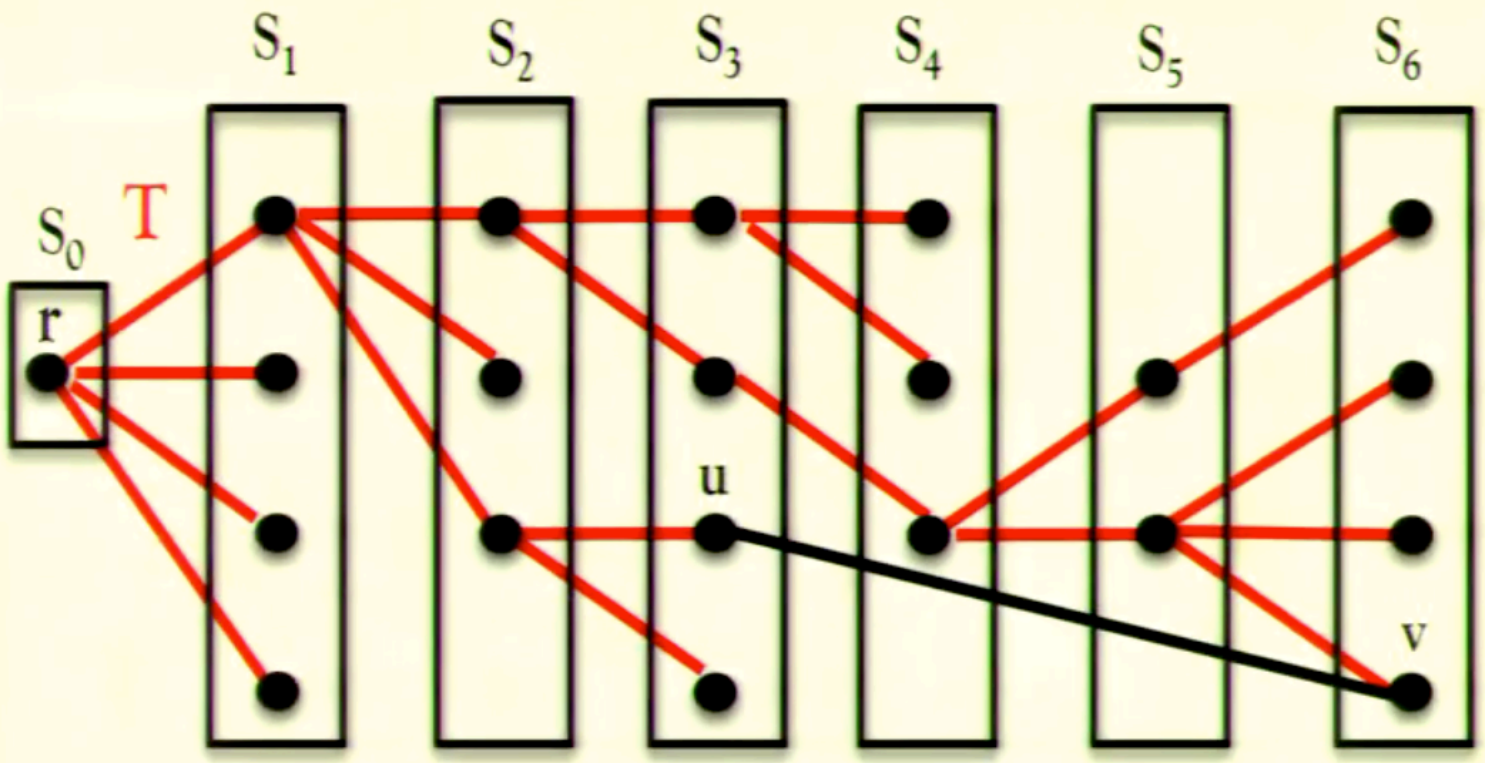
Parent Arc

Breadth First Search Trees

- Because BFS uses a *queue* it is easy to verify that:
  - The edges are **added** to the queue in order of their distance from r.
  - The vertices are **marked** in order of their distance from r.
- Specifically:

Theorem. For any vertex v, the path from v to r given by the search tree T of predecessor edges is a shortest path.

Structure



- Let S_ℓ be the set or "layer" of vertices at distance ℓ from r in T.
- Then a vertex $v \in S_\ell$ is also at distance ℓ from r in the whole graph G.
  - Thus implies for every non-tree edge (u, v), u and v are either in the same layer or in adjacent layers.
  - If not, we can find a shorter path to the root from u or v .

BFS and Bipartite Graphs

- Recall a graph $G = (V, E)$ is **bipartite** if $V = X \cup Y$ and every edge has exactly one end-point in X and one end-point in Y.

Theorem. A graph G is bipartite *if and only if* it contains no odd length cycles.

Proof.

$(\Rightarrow)$

- Assume G contains as a subgraph an odd length cycle C.
- Let $C = \{v_0, v_1, v_1, \ldots, v_{2k}\}$
- Wlog we may assume vertex $v_0 \in Y$. Therefore:

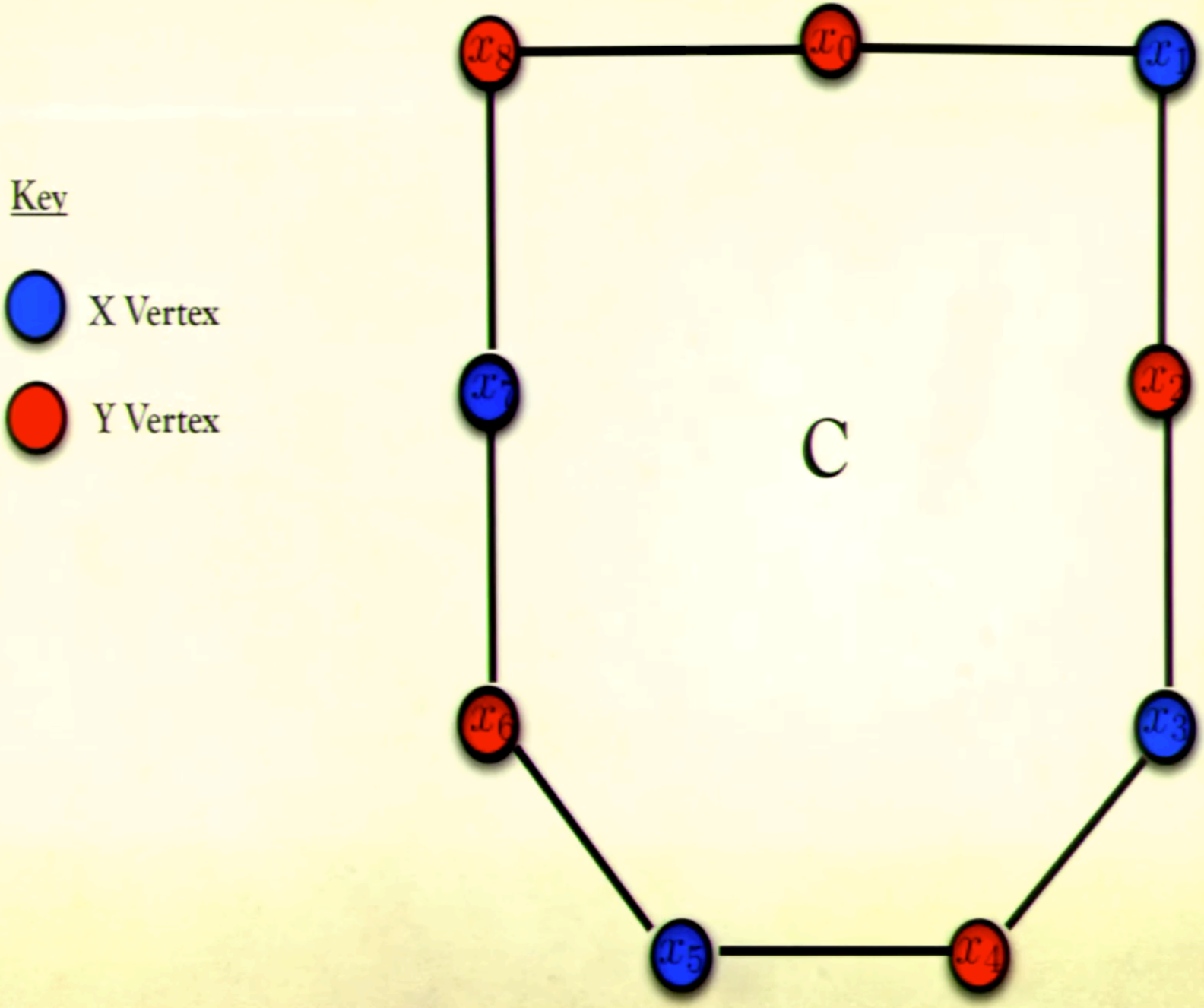
$$v_0 \in Y \Longrightarrow v_1 \in X \Longrightarrow v_2 \in Y \Longrightarrow v_3 \in X \cdots \Longrightarrow \boxed{v_{2k} \in Y}$$

- But $(v_0, v_{2k}) \in E$. Thus

$$v_0 \in Y \Longrightarrow \boxed{v_{2k} \in X}$$

- This contradiction proves that G contains **no** odd cycles.

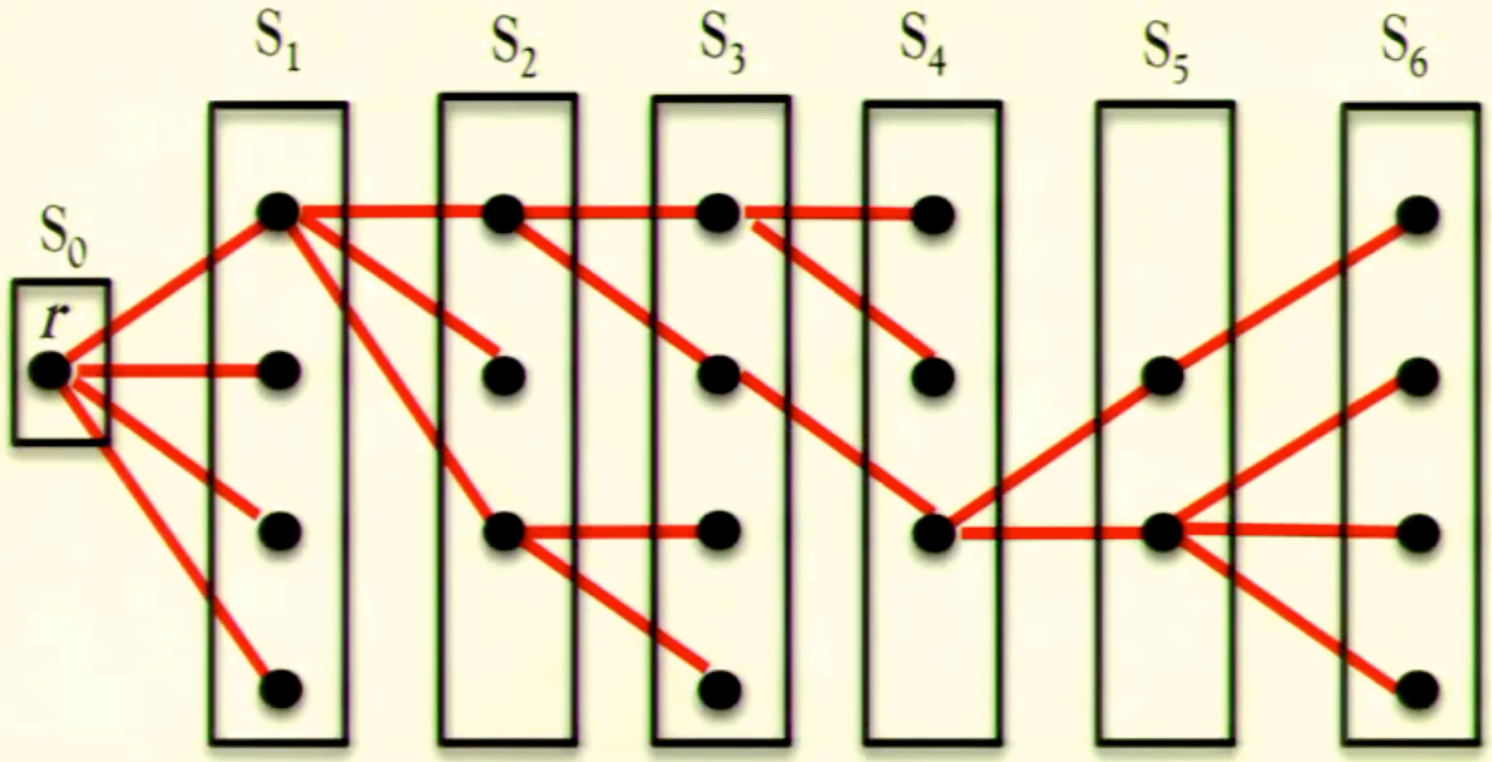
A Pictorial Proof (k=4).



Proof [cont].

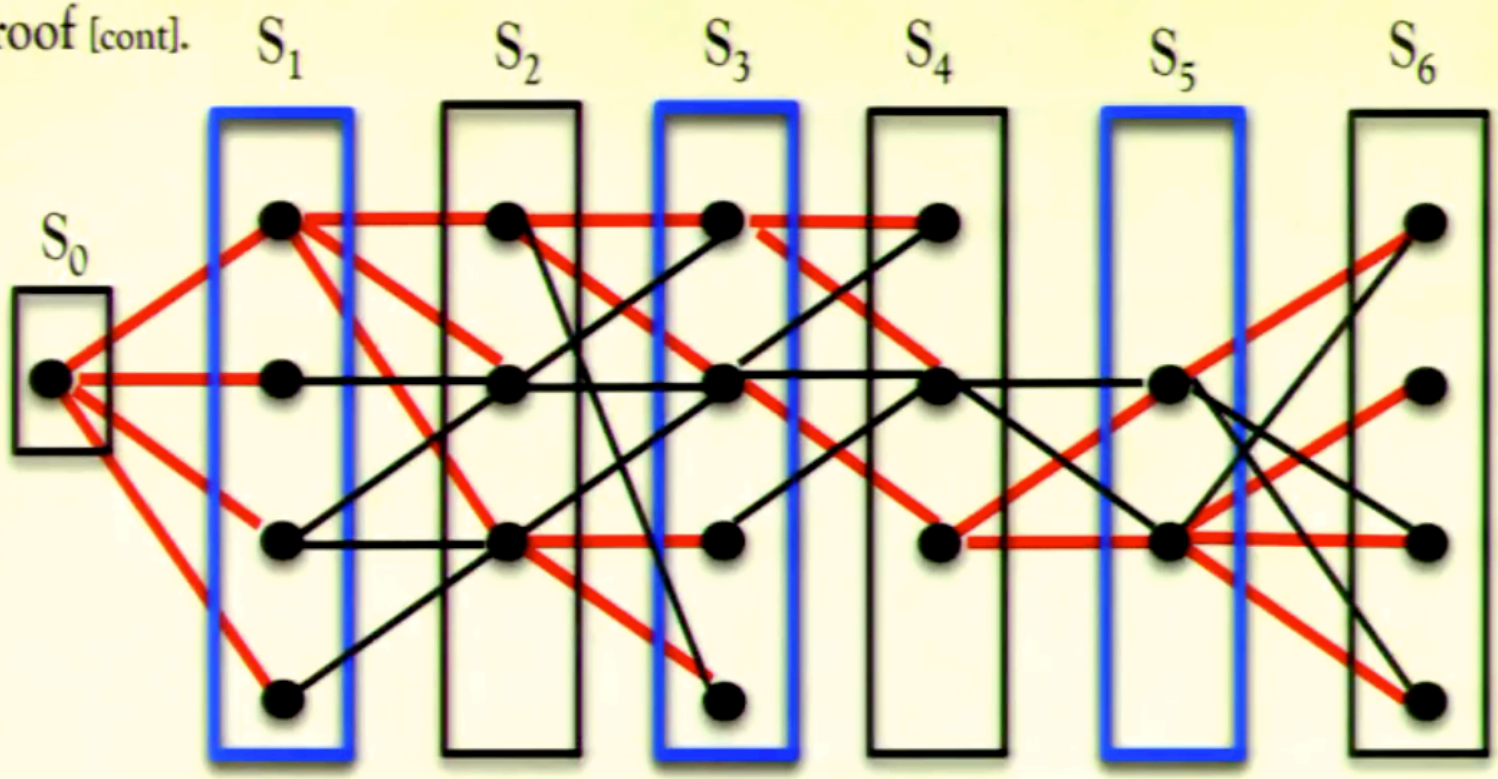
$(\Leftarrow)$

- Assume G contains **no** odd length cycles.
- Select an arbitrary root vertex r and run the BFS algorithm.



- Recall that for every non-tree edge (u, v), we have that u and v are either in the **same** layer or in **adjacent** layers.

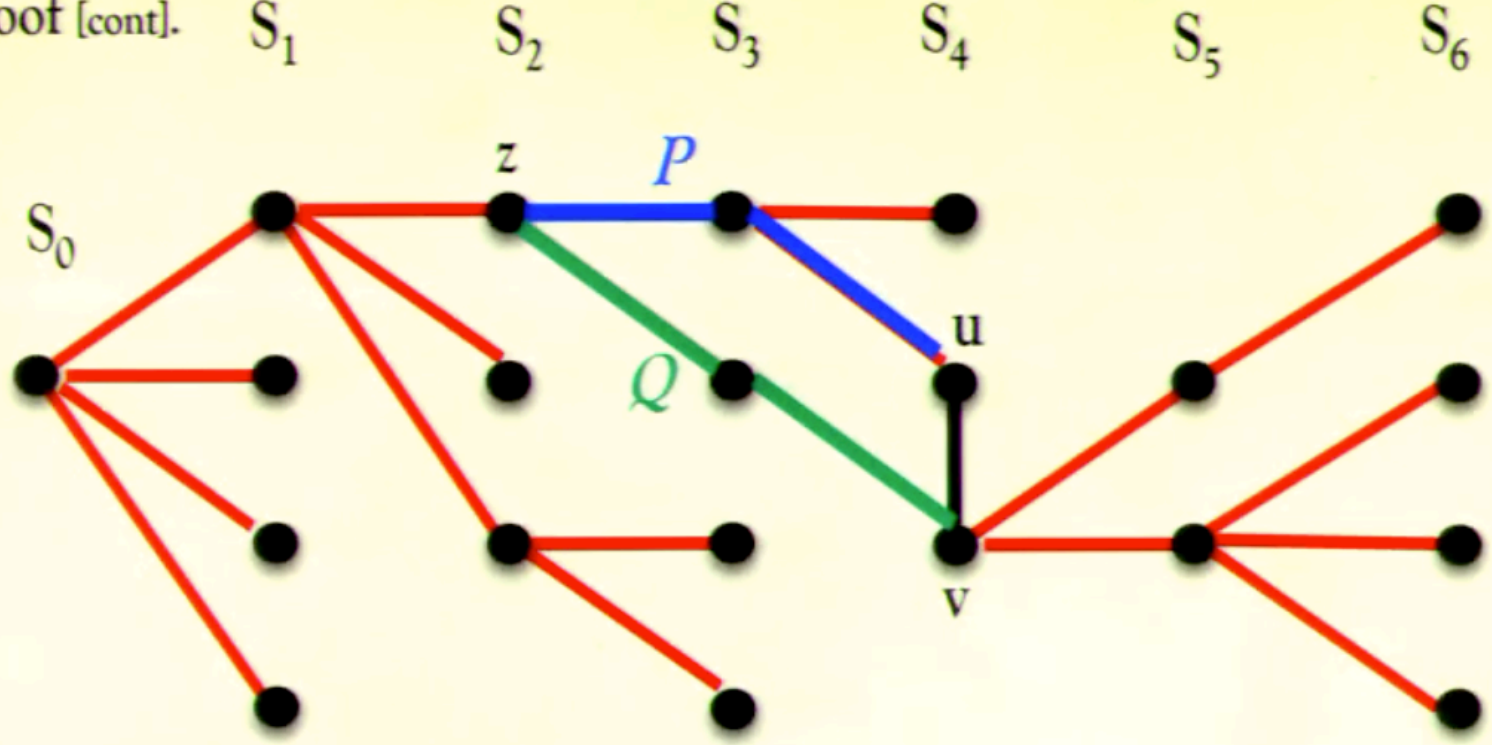
Proof [cont].



- Now set: $X = \bigcup_{\ell \text{ odd}} S_\ell \quad Y = \bigcup_{\ell \text{ even}} S_\ell$

- Suppose, for <u>every</u> non-tree edge (u, v), that u and v are in adjacent layers.
- Now, for <u>every</u> tree edge (u, v) we also have u and v in adjacent layers.
- But adjacent layers **alternate** between X and Y, so G is bipartite.

Proof [cont].



- So assume, there is non-tree edge (u, v) with u and v in the **same** layer.
- Let z be the closest common ancestor of u and v in the search tree T.
- Let P be the path in T from u to z; let Q be the path in T from v to z.
- Since u and v in the same layer we have: $|P| = |Q|$
- But then the cycle $C = P \cup Q \cup (u, v)$ has an **odd** number of edges as: