CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

Copyright © 2014 by Clifford A. Shaffer

Title page

Students should be familiar with inductive proofs, recursion, data structures, and programming at the CS3114 level.

Graph Algorithms

Graphs are useful for representing a variety of concepts:

- Data Structures
- Relationships
- Families
- Communication Networks
- Road Maps

Graph Algorithms

Graphs are useful for representing a variety of concepts:
- Data Structures
- Relationships
- Families
- Communication Networks
- Road Maps

- A **graph** $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ consists of a set of **vertices** V, and a set of **edges** E, such that each edge in E is a connection between a pair of vertices in V.
- Directed vs. Undirected
- Labeled graph, weighted graph
- Labels for edges vs. weights for edges
- Multiple edges, loops
- Cycle, Circuit, path, simple path, tours
- Bipartite, acyclic, connected
- Rooted tree, unrooted tree, free tree

A Tree Proof

- **Definition**: A **free tree** is a connected, undirected graph that has no cycles.
- **Theorem**: If T is a free tree having n vertices, then T has exactly $n - 1$ edges.
- **Proof**: By induction on n.
- **Base Case**: $n = 1$. T consists of 1 vertex and 0 edges.
- **Inductive Hypothesis**: The theorem is true for a tree having $n - 1$ vertices.
- **Inductive Step**:
  - If T has n vertices, then T contains a vertex of degree 1.
  - Remove that vertex and its incident edge to obtain T', a free tree with $n - 1$ vertices.
  - By IH, T' has $n - 2$ edges.
  - Thus, T has $n - 1$ edges.

A Tree Proof

- **Definition**: A free tree is a connected, undirected graph that has no cycles.
- **Theorem**: If T is a free tree having n vertices, then T has exactly $n - 1$ edges.
- **Proof**: By induction on n.
- **Base Case**: $n = 1$. T consists of 1 vertex and 0 edges.
- **Inductive Hypothesis**: The theorem is true for a tree having $n - 1$ vertices.
- **Inductive Step**:
  - If T has n vertices, then T contains a vertex of degree 1.
  - Remove that vertex and its incident edge to obtain T', a free tree with $n - 1$ vertices.
  - By IH, T' has $n - 2$ edges.
  - Thus, T has $n - 1$ edges.

This is close to a satisfactory definition for free tree. There are several equivalent definitions for free trees, with similar proofs to relate them.

Why do we know that some vertex has degree 1? Because the definition says that the Free Tree has no cycles.

Graph Traversals

Various problems require a way to **traverse** a graph – that is, visit each vertex and edge in a systematic way.

Three common traversals:

1. Eulerian tours
   Traverse each edge exactly once
2. Depth-first search
   Keeps vertices on a stack
3. Breadth-first search
   Keeps vertices on a queue

Graph Traversals

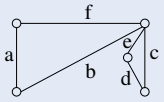
Various problems require a way to traverse a graph – that is, visit each vertex and edge in a systematic way.

Three common traversals:
1. Eulerian tours
   Traverse each edge exactly once
2. Depth-first search
   Keeps vertices on a stack
3. Breadth-first search
   Keeps vertices on a queue

a vertex may be visited multiple times

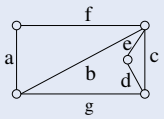
Eulerian Tours

A circuit that contains every edge exactly once.
Example:

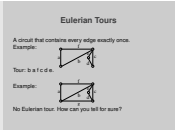


Tour: b a f c d e.

Example:



No Eulerian tour. How can you tell for sure?

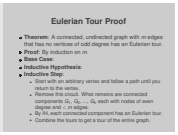


Why no tour? Because some vertices have odd degree.

All even nodes is a necessary condition. Is it sufficient?

Eulerian Tour Proof

- **Theorem**: A connected, undirected graph with m edges that has no vertices of odd degree has an Eulerian tour.
- **Proof**: By induction on m.
- **Base Case**:
- **Inductive Hypothesis**:
- **Inductive Step**:
  - Start with an arbitrary vertex and follow a path until you return to the vertex.
  - Remove this circuit. What remains are connected components G_1, G_2, ..., G_k each with nodes of even degree and $< m$ edges.
  - By IH, each connected component has an Eulerian tour.
  - Combine the tours to get a tour of the entire graph.



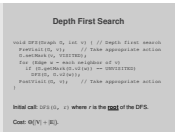
Base case: 0 edges and 1 vertex fits the theorem.
IH: The theorem is true for $< m$ edges.
Always possible to find a circuit starting at any arbitrary vertex, since each vertex has even degree.

Depth First Search

```
void DFS(Graph G, int v) { // Depth first search
  PreVisit(G, v);     // Take appropriate action
  G.setMark(v, VISITED);
  for (Edge w = each neighbor of v)
    if (G.getMark(G.v2(w)) == UNVISITED)
      DFS(G, G.v2(w));
  PostVisit(G, v);    // Take appropriate action
}
```

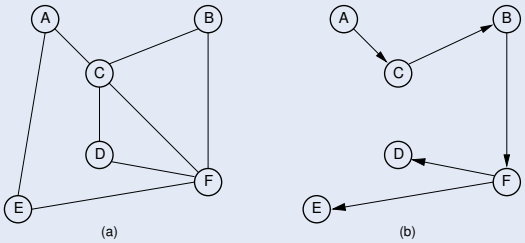
Initial call: `DFS(G, r)` where r is the **root** of the DFS.

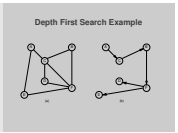
Cost: $\Theta(|\mathrm{V}| + |\mathrm{E}|)$.



no notes

Depth First Search Example





The directions are imposed by the traversal. This is the Depth First Search Tree.

DFS Tree

If we number the vertices in the order that they are marked, we get **DFS numbers**.

Lemma 7.2: Every edge $e \in E$ is either in the DFS tree T, or connects two vertices of G, one of which is an ancestor of the other in T.

Proof: Consider the first time an edge (v, w) is examined, with v the current vertex.

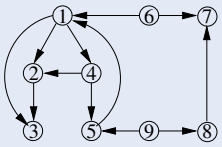
- If w is unmarked, then (v, w) is in T.
- If w is marked, then w has a smaller DFS number than v AND (v, w) is an unexamined edge of w.
- Thus, w is still on the stack. That is, w is on a path from v.

DFS Tree
If we number the vertices in the order that they are marked, we get DFS numbers.
Lemma 7.2: Every edge $e \in E$ is either in the DFS tree T, or connects two vertices of G, one of which is an ancestor of the other in T.
Proof: Consider the first time an edge (v, w) is examined, with v the current vertex.
- If w is unmarked, then (v, w) is in T.
- If w is marked, then w has a smaller DFS number than v AND (v, w) is an unexamined edge of w.
- Thus, w is still on the stack. That is, w is on a path from v.

Results: No “cross edges.” That is, no edges connecting vertices sideways in the tree.

DFS for Directed Graphs

- Main problem: A connected graph may not give a single DFS tree.



- Forward edges: (1, 3)
- Back edges: (5, 1)
- Cross edges: (6, 1), (8, 7), (9, 5), (9, 8), (4, 2)
- **Solution**: Maintain a list of unmarked vertices.
  - Whenever one DFS tree is complete, choose an arbitrary unmarked vertex as the root for a new tree.

DFS for Directed Graphs
- Main problem: A connected graph may not give a single DFS tree.
- Forward edges: (1, 3)
- Back edges: (5, 1)
- Cross edges: (6, 1), (8, 7), (9, 5), (9, 8), (4, 2)
- Solution: Maintain a list of unmarked vertices.
  - Whenever one DFS tree is complete, choose an arbitrary unmarked vertex as the root for a new tree.

no notes

Directed Cycles

Lemma 7.4: Let G be a directed graph. G has a directed cycle iff every DFS of G produces a back edge.

Proof:

1. Suppose a DFS produces a back edge (v, w).
   - v and w are in the same DFS tree, w an ancestor of v.
   - (v, w) and the path in the tree from w to v form a directed cycle.
2. Suppose G has a directed cycle C.
   - Do a DFS on G.
   - Let w be the vertex of C with smallest DFS number.
   - Let (v, w) be the edge of C coming into w.
   - v is a descendant of w in a DFS tree.
   - Therefore, (v, w) is a back edge.

Directed Cycles
Lemma 7.4: Let G be a directed graph. G has a directed cycle iff every DFS of G produces a back edge.
Proof:
1. Suppose a DFS produces a back edge (v, w).
   - v and w are in the same DFS tree, w an ancestor of v.
   - (v, w) and the path in the tree from w to v form a directed cycle.
2. Suppose G has a directed cycle C.
   - Do a DFS on G.
   - Let w be the vertex of C with smallest DFS number.
   - Let (v, w) be the edge of C coming into w.
   - v is a descendant of w in a DFS tree.
   - Therefore, (v, w) is a back edge.

See earlier lemma.

Breadth First Search

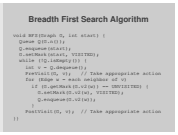
- Like DFS, but replace stack with a queue.
- Visit vertex’s neighbors before going deeper in tree.

Breadth First Search
- Like DFS, but replace stack with a queue.
- Visit vertex’s neighbors before going deeper in tree.

no notes

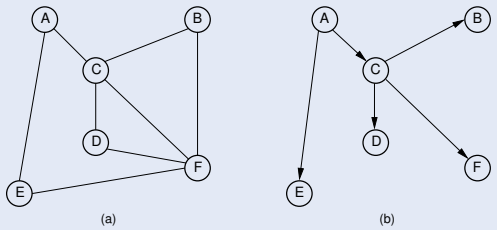
Breadth First Search Algorithm

```
void BFS(Graph G, int start) {
  Queue Q(G.n());
  Q.enqueue(start);
  G.setMark(start, VISITED);
  while (!Q.isEmpty()) {
    int v = Q.dequeue();
    PreVisit(G, v);   // Take appropriate action
    for (Edge w = each neighbor of v)
      if (G.getMark(G.v2(w)) == UNVISITED) {
        G.setMark(G.v2(w), VISITED);
        Q.enqueue(G.v2(w));
      }
    PostVisit(G, v);  // Take appropriate action
}}
```

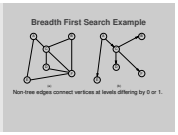



no notes

Breadth First Search Example



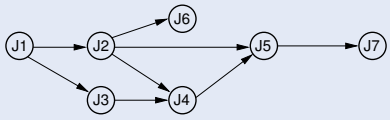
Non-tree edges connect vertices at levels differing by 0 or 1.

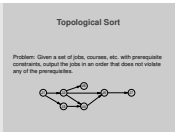


We know this because if an edge had connected to a deeper level, then that target node would have been placed on the queue when the edge was encountered.

Topological Sort

Problem: Given a set of jobs, courses, etc. with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.

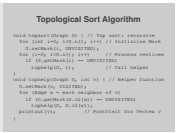




no notes

Topological Sort Algorithm

```
void topsort(Graph G) { // Top sort: recursive
  for (int i=0; i<G.n(); i++) // Initialize Mark
    G.setMark(i, UNVISITED);
  for (i=0; i<G.n(); i++)     // Process vertices
    if (G.getMark(i) == UNVISITED)
      tophelp(G, i);          // Call helper
}
void tophelp(Graph G, int v) { // Helper function
  G.setMark(v, VISITED);
  for (Edge w = each neighbor of v)
    if (G.getMark(G.v2(w)) == UNVISITED)
      tophelp(G, G.v2(w));
  printout(v);          // PostVisit for Vertex v
}
```

Prints in reverse order.

Queue-based Topological Sort

```
void topsort(Graph G) { // Top sort: Queue
  Queue Q(G.n());  int Count[G.n()];
  for (int v=0; v<G.n(); v++) Count[v] = 0;
  for (v=0; v<G.n(); v++) // Process every edge
    for (Edge w each neighbor of v)
       Count[G.v2(w)]++;  // Add to v2's count
  for (v=0; v<G.n(); v++) // Initialize Queue
    if (Count[v] == 0) Q.enqueue(v);
  while (!Q.isEmpty()) {  // Process the vertices
    int v = Q.dequeue();
    printout(v);          // PreVisit for v
    for (Edge w = each neighbor of v) {
      Count[G.v2(w)]--;   // One less prereq
      if (Count[G.v2(w)]==0) Q.enqueue(G.v2(w));
}}}
```


Queue-based Topological Sort

no notes

Shortest Paths Problems

Input: A graph with **weights** or **costs** associated with each edge.

Output: The list of edges forming the shortest path.

Sample problems:

- Find the shortest path between two specified vertices.
- Find the shortest path from vertex S to all other vertices.
- Find the shortest path between all pairs of vertices.

Our algorithms will actually calculate only **distances**.

Shortest Paths Problems

Input: A graph with weights or costs associated with each edge.

Output: The list of edges forming the shortest path.

Sample problems:
- Find the shortest path between two specified vertices.
- Find the shortest path from vertex S to all other vertices.
- Find the shortest path between all pairs of vertices.

Our algorithms will actually calculate only distances.

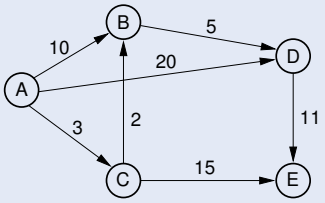
no notes

Shortest Paths Definitions

d(A, B) is the **shortest distance** from vertex A to B.

w(A, B) is the **weight** of the edge connecting A to B.

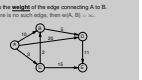
- If there is no such edge, then w(A, B) $= \infty$.



Shortest Paths Definitions

d(A, B) is the shortest distance from vertex A to B.

w(A, B) is the weight of the edge connecting A to B.
- If there is no such edge, then w(A, B) $= \infty$.



w(A, D) $= 20$; d(A, D) $= 10$ (through ACBD).

Single Source Shortest Paths

Given start vertex s, find the shortest path from s to all other vertices.

Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex x.

Problem: Shortest path to a vertex already processed might go through x.
Solution: Process vertices in order of distance from s.

Single Source Shortest Paths

Given start vertex s, find the shortest path from s to all other vertices.

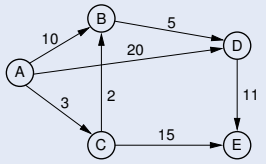
Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex x.

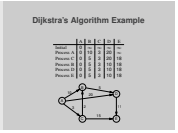
Problem: Shortest path to a vertex already processed might go through x.
Solution: Process vertices in order of distance from s.

no notes

Dijkstra's Algorithm Example

	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18

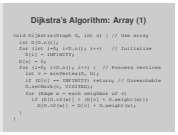




no notes

Dijkstra's Algorithm: Array (1)

```
void Dijkstra(Graph G, int s) { // Use array
  int D[G.n()];
  for (int i=0; i<G.n(); i++)   // Initialize
    D[i] = INFINITY;
  D[s] = 0;
  for (i=0; i<G.n(); i++) {  // Process vertices
    int v = minVertex(G, D);
    if (D[v] == INFINITY) return; // Unreachable
    G.setMark(v, VISITED);
    for (Edge w = each neighbor of v)
      if (D[G.v2(w)] > (D[v] + G.weight(w)))
        D[G.v2(w)] = D[v] + G.weight(w);
  }
}
```

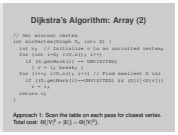



no notes

Dijkstra's Algorithm: Array (2)

```
// Get mincost vertex
int minVertex(Graph G, int* D) {
  int v;  // Initialize v to an unvisited vertex;
  for (int i=0; i<G.n(); i++)
    if (G.getMark(i) == UNVISITED)
      { v = i; break; }
  for (i++; i<G.n(); i++) // Find smallest D val
    if ((G.getMark(i)==UNVISITED) && (D[i]<D[v]))
      v = i;
  return v;
}
```

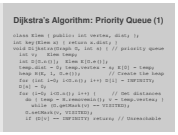
Approach 1: Scan the table on each pass for closest vertex.
Total cost: $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$.



no notes

Dijkstra's Algorithm: Priority Queue (1)

```
class Elem { public: int vertex, dist; };
int key(Elem x) { return x.dist; }
void Dijkstra(Graph G, int s) { // priority queue
  int v;   Elem temp;
  int D[G.n()];  Elem E[G.e()];
  temp.dist = 0; temp.vertex = s; E[0] = temp;
  heap H(E, 1, G.e());          // Create the heap
  for (int i=0; i<G.n(); i++) D[i] = INFINITY;
  D[s] = 0;
  for (i=0; i<G.n(); i++) {     // Get distances
    do { temp = H.removemin(); v = temp.vertex; }
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (D[v] == INFINITY) return; // Unreachable
```

no notes

Dijkstra's Algorithm: Priority Queue (2)

```
    for (Edge w = each neighbor of v)
      if (D[G.v2(w)] > (D[v] + G.weight(w))) {
        D[G.v2(w)] = D[v] + G.weight(w);
        temp.dist = D[G.v2(w)];
        temp.vertex = G.v2(w);
        H.insert(temp);  // Insert new distance
}}}
```

- Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value. Must update priority queue for each edge.
- Total cost: $\Theta((|\mathrm{V}| + |\mathrm{E}|) \log |\mathrm{V}|)$.

Dijkstra's Algorithm: Priority Queue (2)

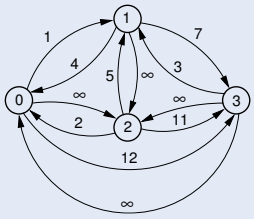
for (Edge w = each neighbor of v)
if (D[G.v2(w)] > (D[v] + G.weight(w))) {
D[G.v2(w)] = D[v] + G.weight(w);
temp.dist = D[G.v2(w)];
temp.vertex = G.v2(w);
H.insert(temp); // Insert new distance
}}}

- Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value. Must update priority queue for each edge.
- Total cost: $\Theta((|\mathrm{V}| + |\mathrm{E}|) \log |\mathrm{V}|)$.

no notes

All Pairs Shortest Paths

- For every vertex $u, v \in \mathrm{V}$, calculate d(u, v).
- Could run Dijkstra's Algorithm |V| times.
- Better is **Floyd's Algorithm**.
- Define a **k-path** from u to v to be any path whose intermediate vertices all have indices less than k.



All Pairs Shortest Paths

- For every vertex $u, v \in \mathrm{V}$, calculate d(u, v).
- Could run Dijkstra's Algorithm |V| times.
- Better is Floyd's Algorithm.
- Define a k-path from u to v to be any path whose intermediate vertices all have indices less than k.

Multiple runs of Dijkstra's algorithm Cost:
$|V||E| \log |V| = |V|^3 \log |V|$ for dense graph.

The issue driving the concept of "k paths" is how to efficiently check all the paths without computing any path more than once.

0,3 is a 0-path. 2,0,3 is a 1-path. 0,2,3 is a 3-path, but not a 2 or 1 path. Everything is a 4 path.

Floyd's Algorithm

```
void Floyd(Graph G) { // All-pairs shortest paths
  int D[G.n()][G.n()]; // Store distances
  for (int i=0; i<G.n(); i++) // Initialize D
    for (int j=0; j<G.n(); j++)
      D[i][j] = G.weight(i, j);
  for (int k=0; k<G.n(); k++) // Compute k paths
    for (int i=0; i<G.n(); i++)
      for (int j=0; j<G.n(); j++)
        if (D[i][j] > (D[i][k] + D[k][j]))
          D[i][j] = D[i][k] + D[k][j];
}
```


Floyd's Algorithm

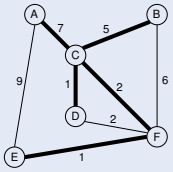
void Floyd(Graph G) { // All-pairs shortest paths
int D[G.n()][G.n()]; // Store distances
for (int i=0; i<G.n(); i++) // Initialize D
for (int j=0; j<G.n(); j++)
D[i][j] = G.weight(i, j);
for (int k=0; k<G.n(); k++) // Compute k paths
for (int i=0; i<G.n(); i++)
for (int j=0; j<G.n(); j++)
if (D[i][j] > (D[i][k] + D[k][j]))
D[i][j] = D[i][k] + D[k][j];
}

no notes

Minimum Cost Spanning Trees

Minimum Cost Spanning Tree (MST) Problem:

- Input: An undirected, connected graph G.
- Output: The subgraph of G that
  1. has minimum total cost as measured by summing the values for all of the edges in the subset, and
  2. keeps the vertices connected.



Minimum Cost Spanning Trees

Minimum Cost Spanning Tree (MST) Problem:

- Input: An undirected, connected graph G.
- Output: The subgraph of G that
  1. has minimum total cost as measured by summing the values for all of the edges in the subset, and
  2. keeps the vertices connected.

no notes

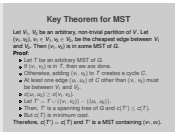
Key Theorem for MST

Let V_1, V_2 be an arbitrary, non-trivial partition of V. Let (v_1, v_2), $v_1 \in V_1, v_2 \in V_2$, be the cheapest edge between V_1 and V_2. Then (v_1, v_2) is in some MST of G.

Proof:

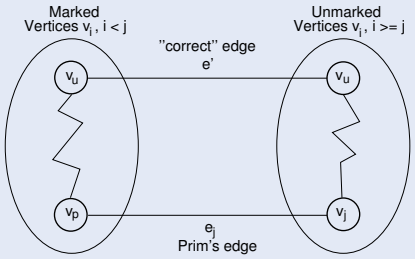
- Let T be an arbitrary MST of G.
- If (v_1, v_2) is in T, then we are done.
- Otherwise, adding (v_1, v_2) to T creates a cycle C.
- At least one edge (u_1, u_2) of C other than (v_1, v_2) must be between V_1 and V_2.
- $c(u_1, u_2) \geq c(v_1, v_2)$.
- Let $T' = T \cup \{(v_1, v_2)\} - \{(u_1, u_2)\}$.
- Then, T' is a spanning tree of G and $c(T') \leq c(T)$.
- But $c(T)$ is minimum cost.

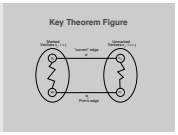
Therefore, $c(T') = c(T)$ and T' is a MST containing (v_1, v_2).



There can only be multiple MSTs when there are edges with equal cost.

Key Theorem Figure

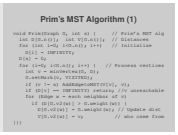




no notes

Prim's MST Algorithm (1)

```
void Prim(Graph G, int s) {     // Prim's MST alg
  int D[G.n()];  int V[G.n()];  // Distances
  for (int i=0; i<G.n(); i++)   // Initialize
    D[i] = INFINITY;
  D[s] = 0;
  for (i=0; i<G.n(); i++) {   // Process vertices
    int v = minVertex(G, D);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == INFINITY) return; //v unreachable
    for (Edge w = each neighbor of v)
      if (D[G.v2(w)] > G.weight(w)) {
        D[G.v2(w)] = G.weight(w); // Update dist
        V[G.v2(w)] = v;          // who came from
}}}
```

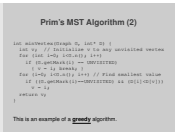



no notes

Prim's MST Algorithm (2)

```
int minVertex(Graph G, int* D) {
  int v;  // Initialize v to any unvisited vertex
  for (int i=0; i<G.n(); i++)
    if (G.getMark(i) == UNVISITED)
      { v = i; break; }
  for (i=0; i<G.n(); i++) // Find smallest value
    if ((G.getMark(i)==UNVISITED) && (D[i]<D[v]))
      v = i;
  return v;
}
```

This is an example of a **greedy** algorithm.

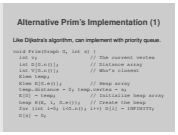


no notes

Alternative Prim's Implementation (1)

Like Dijkstra's algorithm, can implement with priority queue.

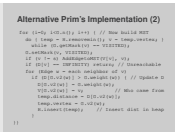
```
void Prim(Graph G, int s) {
  int v;                  // The current vertex
  int D[G.n()];           // Distance array
  int V[G.n()];           // Who's closest
  Elem temp;
  Elem E[G.e()];          // Heap array
  temp.distance = 0; temp.vertex = s;
  E[0] = temp;            // Initialize heap array
  heap H(E, 1, G.e());    // Create the heap
  for (int i=0; i<G.n(); i++) D[i] = INFINITY;
  D[s] = 0;
```

no notes

Alternative Prim's Implementation (2)

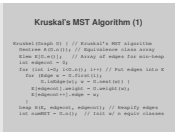
```
  for (i=0; i<G.n(); i++) { // Now build MST
    do { temp = H.removemin(); v = temp.vertex; }
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == INFINITY) return; // Unreachable
    for (Edge w = each neighbor of v)
      if (D[G.v2(w)] > G.weight(w)) { // Update D
        D[G.v2(w)] = G.weight(w);
        V[G.v2(w)] = v;           // Who came from
        temp.distance = D[G.v2(w)];
        temp.vertex = G.v2(w);
        H.insert(temp);    // Insert dist in heap
      }
}}
```

no notes

Kruskal's MST Algorithm (1)

```
Kruskel(Graph G) { // Kruskal's MST algorithm
  Gentree A(G.n()); // Equivalence class array
  Elem E[G.e()];   // Array of edges for min-heap
  int edgecnt = 0;
  for (int i=0; i<G.n(); i++) // Put edges into E
    for (Edge w = G.first(i);
         G.isEdge(w); w = G.next(w)) {
      E[edgecnt].weight = G.weight(w);
      E[edgecnt++].edge = w;
    }
  heap H(E, edgecnt, edgecnt); // Heapify edges
  int numMST = G.n();  // Init w/ n equiv classes
```

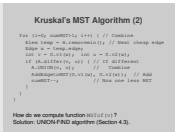



no notes

Kruskal's MST Algorithm (2)

```
  for (i=0; numMST>1; i++) { // Combine
    Elem temp = H.removemin(); // Next cheap edge
    Edge w = temp.edge;
    int v = G.v1(w);  int u = G.v2(w);
    if (A.differ(v, u)) { // If different
      A.UNION(v, u);      //   Combine
      AddEdgetoMST(G.v1(w), G.v2(w));  // Add
      numMST--;           // Now one less MST
    }
  }
}
```

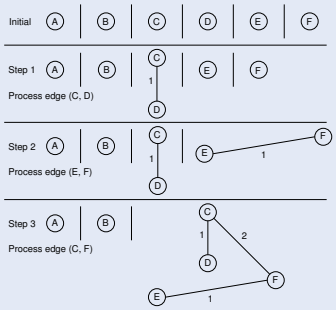
How do we compute function `MSTof(v)`?
Solution: UNION-FIND algorithm (Section 4.3).



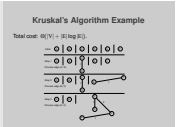
no notes

Kruskal's Algorithm Example

Total cost: $\Theta(|\mathrm{V}| + |\mathrm{E}| \log |\mathrm{E}|)$.



2014-03-20 CS 5114 — Kruskal's Algorithm Example

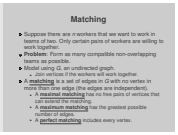


Cost is dominated by the edge sort.
Alternative: Use a min heap, quit when only one set left.
"Kth-smallest" implementation.

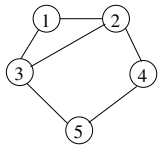
Matching

- Suppose there are n workers that we want to work in teams of two. Only certain pairs of workers are willing to work together.
- **Problem**: Form as many compatible non-overlapping teams as possible.
- Model using G, an undirected graph.
  - Join vertices if the workers will work together.
- A **matching** is a set of edges in G with no vertex in more than one edge (the edges are independent).
  - A **maximal matching** has no free pairs of vertices that can extend the matching.
  - A **maximum matching** has the greatest possible number of edges.
  - A **perfect matching** includes every vertex.

2014-03-20 CS 5114 — Matching



An example:
(1-3) is a matching.
(1-3) (5, 4) is both maximal and maximum.
Take away the edge (5-4). Then (3, 2) would be maximal but not a maximum matching.



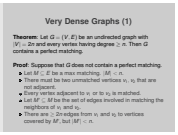
Very Dense Graphs (1)

Theorem: Let $G = (V, E)$ be an undirected graph with $|V| = 2n$ and every vertex having degree $\geq n$. Then G contains a perfect matching.

Proof: Suppose that G does not contain a perfect matching.

- Let $M \subseteq E$ be a max matching. $|M| < n$.
- There must be two unmatched vertices v_1, v_2 that are not adjacent.
- Every vertex adjacent to v_1 or to v_2 is matched.
- Let $M' \subseteq M$ be the set of edges involved in matching the neighbors of v_1 and v_2.
- There are $\geq 2n$ edges from v_1 and v_2 to vertices covered by M', but $|M'| < n$.

2014-03-20 CS 5114 — Very Dense Graphs (1)



There must be two unmatched vertices not adjacent:
Otherwise it would either be perfect (if there are no 2 free vertices) or we could just match v_1 and v_2 (because they are adjacent).

Every adjacent vertex is matched, otherwise the matching would not be maximal.

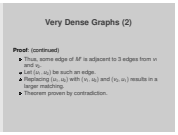
See Manber Figure 3.76.

Very Dense Graphs (2)

Proof: (continued)

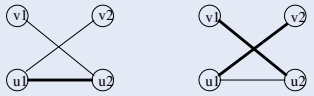
- Thus, some edge of M' is adjacent to 3 edges from v_1 and v_2.
- Let (u_1, u_2) be such an edge.
- Replacing (u_1, u_2) with (v_1, u_2) and (v_2, u_1) results in a larger matching.
- Theorem proven by contradiction.

2014-03-20 CS 5114 — Very Dense Graphs (2)

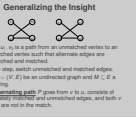


Pigeonhole Principle

Generalizing the Insight

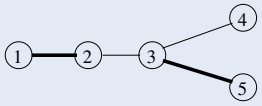


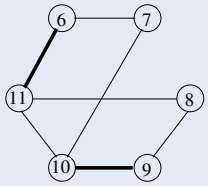
- v_1, u_2, u_1, v_2 is a path from an unmatched vertex to an unmatched vertex such that alternate edges are unmatched and matched.
- In one step, switch unmatched and matched edges.
- Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ a matching.
- An **alternating path** P goes from v to u, consists of alternately matched and unmatched edges, and both v and u are not in the match.

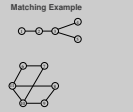


no notes

Matching Example







1, 2, 3, 5 is NOT an alternating path (it does not start with an unmatch vertex).

7, 6, 11, 10, 9, 8 is an alternating path with respect to the given matching.

Observation: If a matching has an alternating path, then the size of the matching can be increased by one by switching matched and unmatched edges along the alternating path.

The Alternating Path Theorem (1)

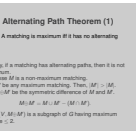
Theorem: A matching is maximum iff it has no alternating paths.

Proof:

- Clearly, if a matching has alternating paths, then it is not maximum.
- Suppose M is a non-maximum matching.
- Let M' be any maximum matching. Then, $|M'| > |M|$.
- Let $M \oplus M'$ be the symmetric difference of M and M'.

$$M \oplus M' = M \cup M' - (M \cap M').$$

- $G' = (V, M \oplus M')$ is a subgraph of G having maximum degree ≤ 2.



The first point is the obvious part of the iff. If there is an alternating path, simply switch the match and umatched edges to augment the match.

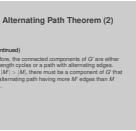
Symmetric difference: Those in either, but not both.

The max degree is ≤ 2 because a vertex matches one different vertex in M and M'.

The Alternating Path Theorem (2)

Proof: (continued)

- Therefore, the connected components of G' are either even-length cycles or a path with alternating edges.
- Since $|M'| > |M|$, there must be a component of G' that is an alternating path having more M' edges than M edges.



no notes

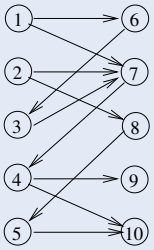
Bipartite Matching

- A **bipartite graph** $G = (U, V, E)$ consists of two disjoint sets of vertices U and V together with edges E such that every edge has an endpoint in U and an endpoint in V.
- Bipartite matching naturally models a number of assignment problems, such as assignment of workers to jobs.
- Alternating paths will work to find a maximum bipartite matching. An alternating path always has one end in U and the other in V.
- If we direct unmatched edges from U to V and matched edges from V to U, then a directed path from an unmatched vertex in U to an unmatched vertex in V is an alternating path.

Bipartite Matching
- A bipartite graph $G = (U, V, E)$ consists of two disjoint sets of vertices U and V together with edges E such that every edge has an endpoint in U and an endpoint in V.
- Bipartite matching naturally models a number of assignment problems, such as assignment of workers to jobs.
- Alternating paths will work to find a maximum bipartite matching. An alternating path always has one end in U and the other in V.
- If we direct unmatched edges from U to V and matched edges from V to U, then a directed path from an unmatched vertex in U to an unmatched vertex in V is an alternating path.

no notes

Bipartite Matching Example



2, 8, 5, 10 is an alternating path.

1, 6, 3, 7, 4, 9 and 2, 8, 5, 10 are **disjoint** alternating paths that we can augment **independently**.

Bipartite Matching Example
2, 8, 5, 10 is an alternating path.
1, 6, 3, 7, 4, 9 and 2, 8, 5, 10 are disjoint alternating paths that we can augment independently.

Naive algorithm: Find a maximal matching (greedy algorithm).

For each vertex:
Do a DFS or other search until an alternating path is found.
Use the alternating path to improve the match.

$|V|(|V| + |E|)$

Algorithm for Maximum Bipartite Matching

Construct BFS subgraph from the set of unmatched vertices in U until a level with unmatched vertices in V is found.

Greedily select a maximal set of disjoint alternating paths.

Augment along each path independently.

Repeat until no alternating paths remain.

Time complexity $O((|V| + |E|)\sqrt{|V|})$.

Algorithm for Maximum Bipartite Matching
Construct BFS subgraph from the set of unmatched vertices in U until a level with unmatched vertices in V is found.
Greedily select a maximal set of disjoint alternating paths.
Augment along each path independently.
Repeat until no alternating paths remain.
Time complexity $O((|V| + |E|)\sqrt{|V|})$.

Order doesn't matter. Find a path, remove its vertices, then repeat.Augment along the paths independently since they are disjoint.