Directed Graphs

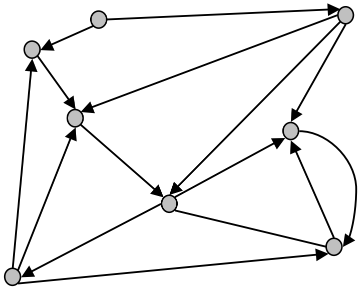
Some of these lecture slides have been adapted from:

- *Algorithms in C, Part 5, 3rd Edition,* Robert Sedgewick.

Directed Graphs

DIGRAPH: directed graph.

- Edge from v to w.
- One-way street.
- Hyperlink from Yahoo to Princeton.



Graphs

Graph	Vertices	Edges
communication	telephones, computers	fiber optic cables
circuits	gates, registers, processors	wires
mechanical	joints	rods, beams, springs
hydraulic	reservoirs, pumping stations	pipelines
financial	stocks, currency	transactions
transportation	street intersections, airports	highways, airway routes
scheduling	tasks	precedence constraints
software systems	functions	function calls
internet	web pages	hyperlinks
games	board positions	legal moves
social relationship	people, actors	friendships, movie casts

A Few Graph Problems

REACHABILITY. Is there a directed path from s to t?

CYCLE. Is there a directed cycle in the graph?

TOPOLOGICAL SORT. Can you draw the graph so that all of the edges point from left to right?

STRONG CONNECTIVITY. Are all vertices mutually reachable?

PAGERANK. What is the importance of a web page (according to Google)?

Graph ADT in C

Typical client program.

- Call `GRAPHinit()` or `GRAPHrand()` to create instance.
- Uses `Graph` handle as argument to ADT functions.
- Calls `Graph` ADT function to do graph processing.

client.c

```
#include <stdio.h>
#include "digraph.h"

int main(int argc, char *argv[]) {
   int V = atoi(argv[1]);
   int E = atoi(argv[2]);
   Graph G = GRAPHrand(V, E);
   GRAPHshow(G);
   GRAPHtc(G);
   return 0;
}
```

Graph ADT in C

Standard method to separate clients from implementation.

- Opaque pointer to `Graph` ADT.
- Plus simple `typedef` for `Edge`.

digraph.h

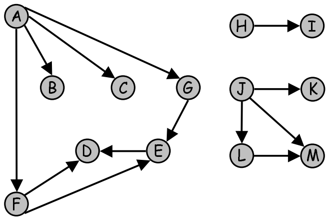
```
typedef struct graph *Graph;
typedef struct { int v, w; } Edge;
Edge EDGEinit(int v, int w);

Graph GRAPHinit(int V);
Graph GRAPHrand(int V, int E);
void  GRAPHdestroy(Graph G);
void  GRAPHshow(Graph G);
void  GRAPHinsertE(Graph G, Edge e);
void  GRAPHremoveE(Graph G, Edge e);
void  GRAPHtc(Graph G);
int   GRAPHisacyclic(Graph G);
. . .
```

Graph Representation

Vertex names. (A B C D E F G H I J K L M)

- C program uses integers between `0` and `V-1`.
- Convert via associative indexing symbol table.

Orientation of edge matters.



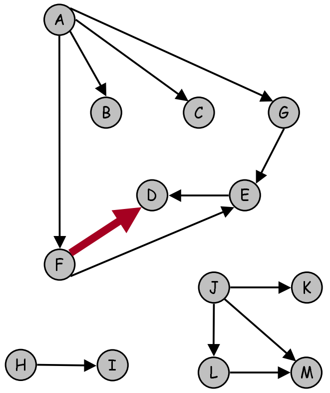
Set of edges representation.

- A-B A-G A-C L-M J-M J-L J-K E-D F-D H-I F-E A-F G-E

Adjacency Matrix Representation

Adjacency matrix representation.

- Two-dimensional V × V array.
- Edge `v-w` in graph: `adj[v][w] = 1`.



		A	B	C	D	E	F	G	H	I	J	K	L	M
0	A	0	1	1	0	0	1	1	0	0	0	0	0	0
1	B	0	0	0	0	0	0	0	0	0	0	0	0	0
2	C	0	0	0	0	0	0	0	0	0	0	0	0	0
3	D	0	0	0	0	0	0	0	0	0	0	0	0	0
4	E	0	0	0	1	0	0	0	0	0	0	0	0	0
5	F	0	0	0	1	1	0	0	0	0	0	0	0	0
6	G	1	1	0	0	1	0	0	0	0	0	0	0	0
7	H	0	0	0	0	0	0	0	0	1	0	0	0	0
8	I	0	0	0	0	0	0	0	0	0	0	0	0	0
9	J	0	0	0	0	0	0	0	0	0	0	1	1	1
10	K	0	0	0	0	0	0	0	0	0	0	0	0	0
11	L	0	0	0	0	0	0	0	0	0	0	0	0	1
12	M	0	0	0	0	0	0	0	0	0	1	0	1	0

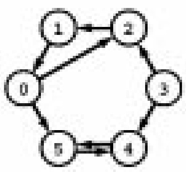
Adjacency Matrix

Transitive Closure

Reflexive transitive closure. G* has an edge from v to w if and only if there is directed path from v to w in G.

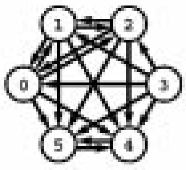
- Not symmetric.
- Supports O(1) reachability queries with $O(V^2)$ space.

G



	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1

G*

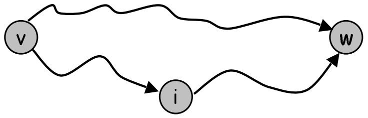


	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	1	1
2	1	1	1	0	1	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

Warshall's Algorithm

Warshall's algorithm.

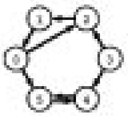
- Initialize tc[v][w] = 1 if v-w exists, 0 otherwise.
- Find path from v to w?
- Take path from v to i and then from i to w if both exist.



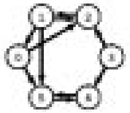
```
for (i = 0; i < G->V; i++)
   for (v = 0; v < G->V; v++)
      for (w = 0; w < G->V; w++)
         if (G->tc[v][i] && G->tc[i][w])
            G->tc[v][w] = 1;
```

Invariant. After ith iteration tc[v][w] = 1 if and only if there exists a path from v to w whose intermediate nodes are 0, 1, …, i.

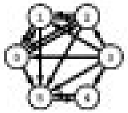
Warshall's Algorithm: Example



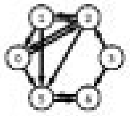
	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1



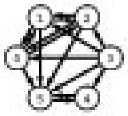
	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	1	0	0	1
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1



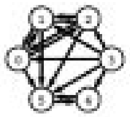
	0	1	2	3	4	5
0	1	1	1	0	0	1
1	1	1	1	0	0	1
2	1	1	1	0	0	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1



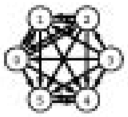
	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	1	0	0	1
2	1	1	1	0	0	1
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1



	0	1	2	3	4	5
0	1	1	1	0	0	1
1	1	1	1	0	0	1
2	1	1	1	0	0	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1



	0	1	2	3	4	5
0	1	1	1	0	0	1
1	1	1	1	0	0	1
2	1	1	1	0	0	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1



	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	1	1
2	1	1	1	0	1	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

Transitive Closure: Cost Summary

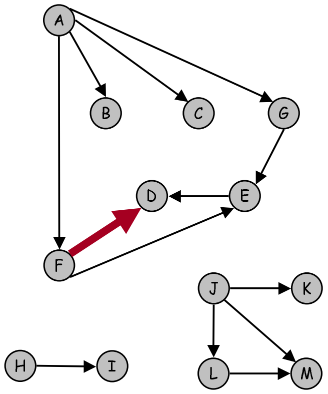
TRANSITIVE CLOSURE. Is there a directed path from v to w?

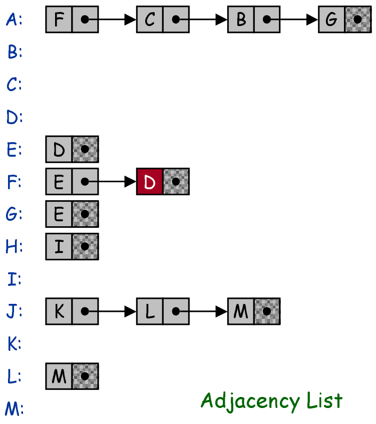
Method	Preprocess	Query	Space
Warshall	V^3	1	V^2

Adjacency List Representation

Vertex indexed array of lists.

- Space proportional to number of edges.
- One representations of each directed edge.





Adjacency List

Depth First Search

TRANSITIVE CLOSURE. Is there a directed path from v to w?

Use DFS to calculate all nodes reachable from v.

To visit a node v:
- mark it as visited
- recursively visit all unmarked nodes w adjacent to v

Enables direct solution of simple graph problems.

- Transitive closure.
- Directed cycles.
- Topological sort.

Basis for solving difficult graph problems.

- Strong connected components.
- Directed Euler path.

Depth First Search: Transitive Closure

digraph.c (transitive closure)

```
int GRAPHtc(Graph G) {
   int s;
   for (s = 0; s < G->V; s++) dfs(G, s, s);
}

int GRAPHreachable(int v, int w) { return G->tc[v][w] == 1; }

void dfs(Graph G, int s, int v) {
   link t;
   int w;

   G->tc[s][v] = 1;

   for (t = G->adj[v]; t != NULL; t = t->next) {
      w = t->w;
      if (G->tc[s][w] == 0) dfs(G, s, w);
   }
}
```

do reachability using each vertex as source

is w reachable from v?

reachability from s

made it to v

assumes `G->tc[][]` was initialized to 0

Transitive Closure: Cost Summary

TRANSITIVE CLOSURE. Is there a directed path from v to w?

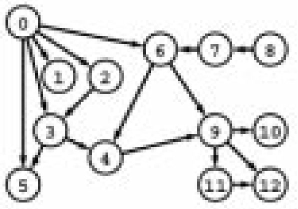
Method	Preprocess	Query	Space
Warshall	V^3	1	V^2
DFS (preprocess)	E V	1	V^2
DFS (online)	1	E + V	E

Open research problem. O(1) query, O(V^2) preprocessing time.

Application: Scheduling

Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

- Task 0: read programming assignment.
- Task 1: download files.
- Task 2: write code.
- . . .
- Task 12: sleep.

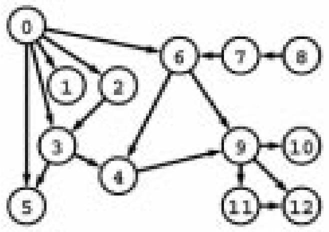


Graph model.

- Create a vertex for each task.
- Create an edge v-w if task v must precede task w.

Directed Acyclic Graph

DAG: directed acyclic graph.



Topological sort: all edges point left to right.

	0	1	2	3	4	5	6	7	8	9	10	11	12
ts	0	1	2	3	8	7	6	4	5	9	10	11	12

Depth First Search: Topological Sort

digraph.c

```
int cnt;         // global

int GRAPHts(Graph G) {
    int v;
    cnt = G->V;
    for (v = 0; v < G->V; v++) G->visited[v] = FALSE;
    for (v = 0; v < G->V; v++)
       if (!G->visited[v]) ts(G, v);
}
                                      run DFS from each vertex
void ts(Graph G, int v) {
   link t;
   G->visited[v] = TRUE;
   for (t = G->adj[v]; t != NULL; t = t->next) {
      int w = t->w;
      if (!G->visited[w]) ts(G, w);
   }
   G->ts[--cnt] = v;        assign numbers in reverse order
}
```

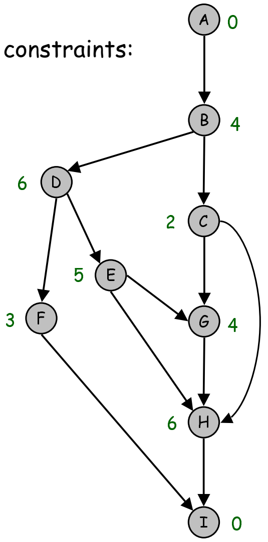
What happens if graph is not a DAG?

Application: PERT/CPM

Program Evaluation and Review Technique / Critical Path Method.

- Task v requires t[v] units of processing time.
- Can work on jobs in parallel subject to precedence constraints:
  - must finish task v before beginning w
- What's the earliest we can complete each task?

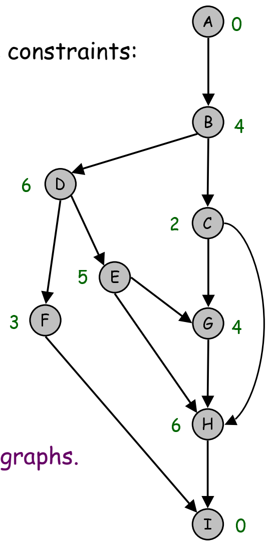
Index	Task	Duration	Prerequisite
A	Framing	0	-
B	Framing	4	A
C	Roofing	2	B
D	Siding	6	B
E	Windows	5	D
F	Plumbing	3	D
G	Electricity	4	C, E
H	Paint	6	C, E
I	Finish	0	F, H



Application: PERT/CPM

Program Evaluation and Review Technique / Critical Path Method.

- Task v requires t[v] units of processing time.
- Can work on jobs in parallel subject to precedence constraints:
  - must finish task v before beginning w
- What's the earliest we can complete each task?



Longest path algorithm in DAG.

- Initialize finish[v] = 0 for all vertices v.
- Consider vertices v in topological order:
  - for each edge v-w
    finish[w] = max(finish[w], finish[v] + t[w]);

Warning: longest path problem is NP-hard in general graphs.

Application: Google's PageRank Algorithm

Goal. Determine which web pages on Internet are important.
Solution. Ignore keywords and content, focus on hyperlink structure.

Random surfer model.

- Start at random page.
- With probability 0.85, randomly selects a link on page to visit next.
  With probability 0.15, randomly select a page.
- Never hit "Back" button.
- PageRank = proportion of time random surfer spends on each page.

Intuition.

- Each page evenly distributes its rank to all pages that it points to.
- Each page receives rank from all pages that point to it.
- Hard to cheat.

Application: Google's PageRank Algorithm

Solution 1: Simulate random surfer for a long time.

Solution 2: Compute ranks directly.

```
for (i = 0; i < PHASES; i++) {
   for (v = 0; v < G->V; v++) oldrank[v] = rank[v];
   for (v = 0; v < G->V; v++) rank[v] = 0;

   for (v = 0; v < G->V; v++) {
      for (t = G->adj[v]; t != NULL; t = t->next) {
         w = t->w;
         rank[w] += 1.0 * oldrank[v] / outdegree[v];
      }
   }
}
```

Solution 3: Compute eigenvalues of adjacency matrix!

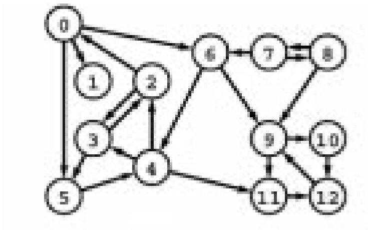
PageRank Caveats

Dead end: page with no outgoing links.

- All importance will leak out of web.
- Easy to detect and ignore.

Spider trap: group of pages with no links leaving the group.

- Group will accumulate all importance of Web.
- Compute strongly connected components.
  - use transitive closure – O(E V) time
  - ingenious algorithms using DFS - O(E + V) time

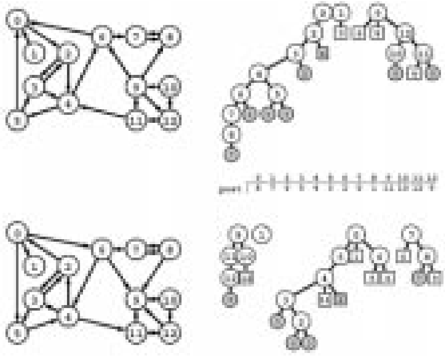


	0	1	2	3	4	5	6	7	8	9	10	11	12
sc	2	1	2	2	2	2	2	3	3	0	0	0	0

Strongly Connected Components

Kosaraju's algorithm.

- Run DFS on reverse digraph and compute postorder.
- Run DFS on original digraph. In search loop that calls dfs(), consider vertices in reverse postorder.



Theorem. Trees in second DFS are strong components. (!)

Strongly Connected Components

Kosaraju's algorithm.

- Run DFS on reverse digraph and compute postorder.
- Run DFS on original digraph. In search loop that calls dfs(), consider vertices in reverse postorder.

```
void dfs(Graph G, int v) {
   link t;
   G->scc[v] = component;
   for (t = G->adj[w]; t != NULL; t = t->next) {
      w = t->w;
      if (G->scc[w] == -1)
         dfs(G, w);
   }
   postorder[cnt++] = v;
}
```

unique id for each strongly connected component

```
for (v = G->V – 1; v >= 0; v--)
   if (G->scc[postorder[v]] == -1) {
      dfs(G, postorder[v]);
      component++;
   }
```

second search loop that calls dfs()