COMP1927: E: Graphs: Definition, Representation, Algorithms

|  |  |
| --- | --- |
| Graph Definitions |  |

|  |
| --- |
| Graphs |

Many problems require

a collection of *items* (i.e. a set)

*relationships*/connections between items

Examples:

maps: items are cities, connections are roads

scheduling: items are tasks, connections are dependencies

networks: items are sites, connections are routes

programs: items are functions, connections are calls

web: items are pages, connections are hyperlinks

|  |  |
| --- | --- |
| ... Graphs | 3/144 |

So far we have looked at several collection types:

lists ... linear sequence of items

trees ... branched hierachy of items

Both are special cases of a general collection type:

graphs ... arbitrarily connected set of items

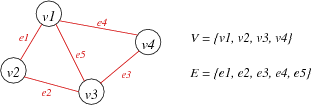
|  |  |
| --- | --- |
| ... Graphs | 4/144 |

A *graph* *G = (V,E)*

*V* is a set of *vertices*

*E* is a set of *edges* (subset of *V×V*)

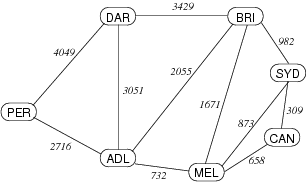
Example:



|  |  |
| --- | --- |
| ... Graphs | 5/144 |

A real example: Australian road distances

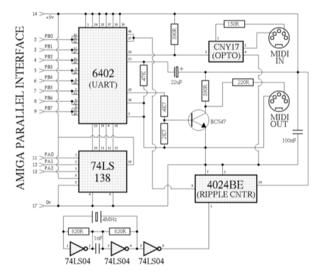
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Distance | Adelaide | Brisbane | Canberra | Darwin | Melbourne | Perth | Sydney |
| Adelaide | - | 2055 | 1390 | 3051 | 732 | 2716 | 1605 |
| Brisbane | 2055 | - | 1291 | 3429 | 1671 | 4771 | 982 |
| Canberra | 1390 | 1291 | - | 4441 | 658 | 4106 | 309 |
| Darwin | 3051 | 3429 | 4441 | - | 3783 | 4049 | 4411 |
| Melbourne | 732 | 1671 | 658 | 3783 | - | 3448 | 873 |
| Perth | 2716 | 4771 | 4106 | 4049 | 3448 | - | 3972 |
| Sydney | 1605 | 982 | 309 | 4411 | 873 | 3972 | - |



|  |  |
| --- | --- |
| ... Graphs | 6/144 |

Applications involving a set of items and connections between items suggest the use of a graph.

Examples: circuit diagrams, hyperlinked documents, ...



|  |  |
| --- | --- |
| ... Graphs | 7/144 |

Questions we might ask about a graph:

is there a way to get from item A to item B?

what is the best way to get from A to B?

which items are connected?

Graph algorithms are generally more complex than tree/list ones:

no implicit order of items

graphs may contain cycles

concrete representation is less obvious

algorithm complexity depends on connection complexity

|  |  |
| --- | --- |
| Properties of Graphs | 8/144 |

A graph with *|V|* vertices has at most *|V|(|V|-1)/2* edges.

(when talking about #vertices/nodes, we often omit |..|, e.g. *E ≤ V(V-1)/2*)

The ratio *V:E* can vary considerably.

if *E* is closer to *V2*, the graph is *dense*

if *E* is closer to *V*, the graph is *sparse*

Knowing whether a graph is sparse or dense is important

may affect choice of data structures to represent graph

may affect choice of algorithms to process graph

|  |  |
| --- | --- |
| ... Properties of Graphs | 9/144 |

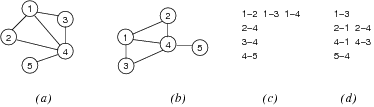
Defining graphs:

need some way of identifying vertices

could give diagram showing edges and vertices

could give a list of edges

E.g. four representations of the same graph:



|  |  |
| --- | --- |
| Graph Terminology | 10/144 |

*Graph*

set of vertices *V*   (e.g. {1, 2, 3, 4, 5})

set of edges *E*, involving all of *V*   (e.g. {1-2, 2-3, 2-4, 3-5})

*Subgraph*:

subset of edges *E'*   (e.g. {1-2, 2-4})

all vertices involved in edges from *E'*   (e.g. {1, 2, 4})

*Induced subgraph*:

subset of vertices *V'*   (e.g. {1, 2, 3, 5})

all edges involving a pair from *V'*   (e.g. {1-2, 2-3, 3-5})

If *|E| = 0*, then we have a *set*.

|  |  |
| --- | --- |
| ... Graph Terminology | 11/144 |

For an edge *e*, that connects vertices *v* and *w*

*v* and *w* are *adjacent*

*e* is *incident* on both *v* and *w*

*Degree* of a vertex *v*

number of edges incident on *e*

Synonyms:

vertex = node,   edge = arc = link

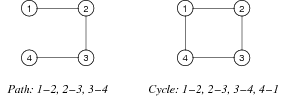
|  |  |
| --- | --- |
| ... Graph Terminology | 12/144 |

*Path*: a sequence of vertices where

each vertex has an edge to its predecessor

*Cycle*: a path where

last vertex in path is same as first vertex in path



|  |  |
| --- | --- |
| ... Graph Terminology | 13/144 |

A *list* is a special case of a graph with

exactly one path

A *tree* is a special case of a graph with

exactly one path between each pair of vertices

Neither a list nor a tree has any cycles.

|  |  |
| --- | --- |
| ... Graph Terminology | 14/144 |

*Connected graph*

there is a *path* from each vertex to every other vertex

if a graph is not connected, it has connected components

*Complete graph*

there is an *edge* from each vertex to every other vertex

in a complete graph, *E = V(V-1)/2*



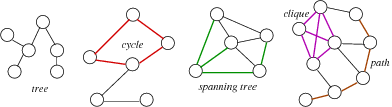
|  |  |
| --- | --- |
| ... Graph Terminology | 15/144 |

*Tree*: connected (sub)graph with no cycles

*Spanning tree*: tree containing all vertices

*Clique*: complete subgraph

Consider the following single graph:



This graph has 26 vertices, 32 edges, and 4 connected components

|  |  |
| --- | --- |
| ... Graph Terminology | 16/144 |

A *spanning tree* of connected graph *G = (V,E)*

is a subgraph of *G* containing all of *V*

and is a single tree (connected, no cycles)

A *spanning forest* of non-connected graph *G = (V,E)*

is a subgraph of *G* containing all of *V*

and is a set of trees (not connected, no cycles)

|  |  |
| --- | --- |
| ... Graph Terminology | 17/144 |

*Undirected graph*

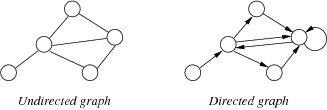
*edge(u,v) = edge(v,u)*,   no self-loops   (i.e. no *edge(v,v)*)

*Directed graph*

*edge(u,v) ≠ edge(v,u)*,   can have self-loops (i.e. *edge(v,v)*)

a directed graph (no parallel edges) has ≤ *V2* edges

Examples:



|  |  |
| --- | --- |
| ... Graph Terminology | 18/144 |

Other types of graphs ...

*Weighted graph*

each edge has an associated value (weight)

e.g. road map   (weights on edges are distances between cities)

*Multi-graph*

allow multiple edges between two vertices

e.g. function call graph   (f() calls g() in several places)

|  |  |
| --- | --- |
| Representing Graphs |  |

|  |  |
| --- | --- |
| Graph ADT | 20/144 |

Data:

set of edges, set of vertices

Operations:

building: create graph, create edge, add edge

deleting: remove edge, drop whole graph

scanning: get edges, copy, show

Notes:

set of vertices is fixed when graph initialised

we treat vertices as ints, but could be Items

|  |  |
| --- | --- |
| ... Graph ADT | 21/144 |

Graph ADT interface:

// graph representation is hidden

typedef struct GraphRep \*Graph;

// vertices denoted by integers 0..N-1

typedef int Vertex;

int validV(Graph,Vertex); //validity check

// edges are pairs of vertices (end-points)

typedef struct { Vertex v; Vertex w; } Edge;

Edge mkEdge(Graph, Vertex, Vertex);

// operations on graphs

Graph newGraph(int nV); // #vertices

void insertE(Graph, Edge);

void removeE(Graph, Edge);

// returns #vertices & array of edges

int edges(Graph, Edge \*, int);

Graph copy(Graph);

void destroy(Graph);

void show(Graph);

|  |  |
| --- | --- |
| ... Graph ADT | 22/144 |

Implementation of foundation operations:

// make an edge for a given Grpah

Edge mkEdge(Graph g, Vertex v, Vertex w)

{

assert(validV(g,v) && validV(g,w));

Edge e = {v,w}; // struct assignment

// or Edge e; e.v = v; e.w = w;

return e;

}

// is a vertex valid in a given Graph?

static int validV(Graph g, Vertex v)

{

return (g != NULL && v >= 0 && v < g->nV);

}

|  |  |
| --- | --- |
| Array-of-edges Representation | 23/144 |

Edges are represented as an array of Edge values

space efficient representation

adding and deleting edges is slightly complex

undirected: order of vertices in Edge doesn't matter

directed: order of vertices encodes direction of Edge



|  |  |
| --- | --- |
| ... Array-of-edges Representation | 24/144 |

Implementation of GraphRep

typedef struct GraphRep {

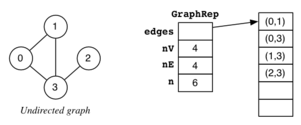
int nV; // #vertices (numbered 0..nV-1)

int nE; // #edges

int n; // size of edge array

Edge \*edges; // array of edges

} GraphRep;



|  |  |
| --- | --- |
| ... Array-of-edges Representation | 25/144 |

Implementation of graph initialisation:

Graph newGraph(int nV)

{

assert(nV >= 0);

int n = *Enough*;

Edge \*e = malloc(n\*sizeof(Edge));

assert(e != NULL);

Graph g = malloc(sizeof(GraphRep));

assert(g != NULL);

g->nV = nV; g->nE = 0;

g->n = n; g->edges = e;

return g;

}

How much is enough? ... No more than *(V2-V)/2*

|  |  |
| --- | --- |
| ... Array-of-edges Representation | 26/144 |

Implementation of edge insertion/removal:

void insertE(Graph g, Edge e)

{

assert(g != NULL && g->nE < g->n);

int i, nE = g->nE;

for (i = 0; i < nE; i++)

if (eq(e,g->edges[i])) break;

if (i == nE)

g->edges[g->nE++] = e;

}

void removeE(Graph g, Edge e)

{

assert(g != NULL);

int i, nE = g->nE;

for (i = 0; i < nE; i++)

if (eq(e,g->edges[i])) break;

if (i < nE) {

g->edges[i] = g->edges[nE-1];

g->nE--;

}

}

|  |  |
| --- | --- |
| ... Array-of-edges Representation | 27/144 |

Cost of operations:

initialisation: *O(1)*   (malloc GraphRep and edges array)

insert edge: *O(E)*   (assuming edge array has space)

delete edge: *O(E)*   (need to find edge in edge array)

If array is full on insert

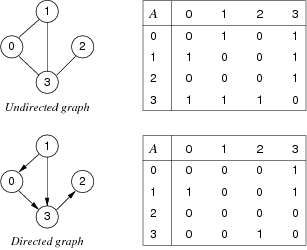
malloc a bigger array, copy edges across ⇒ still *O(E)*

If we maintain edges in order

use binary search to find edge ⇒ *O(logE)*

|  |  |
| --- | --- |
| Adjacency Matrix Representation | 28/144 |

Edges represented by a *V × V* matrix



|  |  |
| --- | --- |
| ... Adjacency Matrix Representation | 29/144 |

Advantages

easily implemented in C as 2-dimensional array

can represent graphs, digraphs and weighted graphs

graphs: symmetric boolean matrix

digraphs: non-symmetric boolean matrix

weighted: non-symmetric matrix of weight values

Disadvantages:

if few edges ⇒ sparse, memory-inefficient

|  |  |
| --- | --- |
| ... Adjacency Matrix Representation | 30/144 |

Implementation of GraphRep

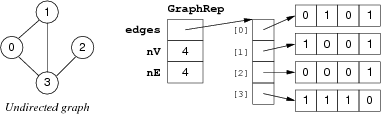
typedef struct GraphRep {

int nV; // #vertices

int nE; // #edges

Bool \*\*edges; // matrix of booleans

} GraphRep;



|  |  |
| --- | --- |
| ... Adjacency Matrix Representation | 31/144 |

Implementation of graph initialisation:

Graph newGraph(int nV)

{

assert(nV >= 0);

int i, j;

int \*\*e = malloc(nV \* sizeof(int \*));

assert(e != NULL);

for (i = 0; i < nV; i++) {

e[i] = malloc(nV \* sizeof(int));

assert(e[i] != NULL);

for (j = 0; j < nV; j++)

e[i][j] = 0; // FALSE

}

Graph g = malloc(sizeof(GraphRep));

assert(g != NULL);

g->nV = nV; g->nE = 0; g->edges = e;

return g;

}

|  |  |
| --- | --- |
| ... Adjacency Matrix Representation | 32/144 |

Implementation of edge insertion/removal:

void insertE(Graph g, Edge e)

{

assert(g != NULL);

assert(validV(g,e.v) && validV(g,e.w));

if (g->edges[e.v][e.w]) return;

g->edges[e.v][e.w] = 1;

g->edges[e.w][e.v] = 1;

g->nE++;

}

void removeE(Graph g, Edge e)

{

assert(g != NULL);

assert(validV(g,e.v) && validV(g,e.w));

if (!g->edges[e.v][e.w]) return;

g->edges[e.v][e.w] = 0;

g->edges[e.w][e.v] = 0;

g->nE--;

}

|  |  |
| --- | --- |
| ... Adjacency Matrix Representation | 33/144 |

Implementation of show()

void show(Graph g)

{

assert(g != NULL);

printf("V=%d, E=%d\n", g->nV, g->nE);

int i, j;

for (i = 0; i < g->nV; i++) {

int nshown = 0;

for (j = i+1; j < g->nV; j++) {

if (g->edges[i][j] != 0) {

printf("%d-%d ",i,j);

nshown++;

}

}

if (nshown > 0) printf("\n");

}

}

|  |  |
| --- | --- |
| ... Adjacency Matrix Representation | 34/144 |

Implementation of edges()

int edges(Graph g, Edge es[], int nE)

{

assert(g != NULL && es != NULL);

assert(nE >= g->nE);

int i, j, n = 0;

for (i = 0; i < g->nV; i++) {

for (j = i+1; j < g->nV; j++) {

if (g->edges[i][j] != 0) {

assert(n < nE);

es[n++] = mkEdge(g,i,j);

}

}

}

return n;

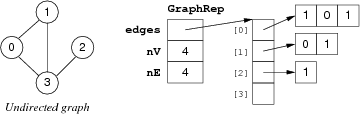
}

|  |  |
| --- | --- |
| ... Adjacency Matrix Representation | 35/144 |

Storage cost: *V* int ptrs + *V2* ints

If the graph is sparse, most storage is wasted.

A storage optimisation: store only top-right part of matrix.



New storage cost: *V-1* int ptrs + *V(V+1)/2* ints   (but still *O(V2)*)

Requires us to always use edges *(v,w)* such that *v < w*.

Cost of operations:

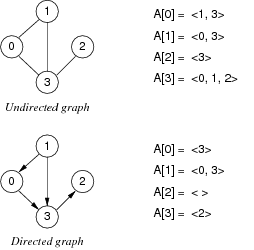
initialisation: *O(V2)*   (initialise *V×V* matrix)

insert edge: *O(1)*   (set two cells in matrix)

delete edge: *O(1)*   (unset two cells in matrix)

|  |  |
| --- | --- |
| Adjacency List Representation | 36/144 |

For each vertex, store linked list of adjacent vertices:



|  |  |
| --- | --- |
| ... Adjacency List Representation | 37/144 |

Advantages

relatively easy to implement in C

can represent graphs and digraphs

memory efficient if *E/V* relatively small

Disavantages:

one graph has many possible representations   
(unless lists are ordered by same criterion e.g. ascending)

|  |  |
| --- | --- |
| ... Adjacency List Representation | 38/144 |

Implementation of GraphRep

typedef struct vNode \*VList;

struct vNode { Vertex v; vList next; };

typedef struct graphRep GraphRep;

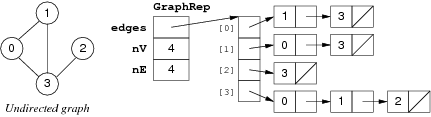
struct graphRep {

int nV; // #vertices

int nE; // #edges

VList \*edges; // array of lists

};



|  |  |
| --- | --- |
| ... Adjacency List Representation | 39/144 |

Implementation of core operations:

Edge mkEdge(Graph g, Vertex x, Vertex y)

{

assert(g != NULL);

assert(validV(g,x) && validV(g,y));

assert(x != y)

Edge e;

if (x < y)

{ e.v = x; e.w = y; }

else

{ e.v = y; e.w = x; }

return e;

}

Graph newGraph(int nV)

{

int i, j;

VList \*e = malloc(nV \* sizeof(VList));

assert(e != NULL);

for (i = 0; i < nV; i++) e[i] = NULL;

Graph g = malloc(sizeof(GraphRep));

assert(g != NULL);

g->nV = nV; g->nE = 0; g->edges = e;

return g;

}

|  |  |
| --- | --- |
| ... Adjacency List Representation | 40/144 |

Implementation of edge insertion/removal:

void insertE(Graph g, Edge e)

{

assert(g != NULL);

assert(validV(g,e.v) && validV(g,e.w));

int orig = length(g->edges[e.v]);

g->edges[e.v] = insertVList(g->edges[e.v], e.w);

g->edges[e.w] = insertVList(g->edges[e.w], e.v);

if (length(g->edges[e.v]) > orig) g->nE++;

}

void removeE(Graph g, Edge e)

{

assert(g != NULL);

assert(validV(g,e.v) && validV(g,e.w));

int orig = length(g->edges[e.v]);

g->edges[e.v] = deleteVList(g->edges[e.v], e.w);

g->edges[e.w] = deleteVList(g->edges[e.w], e.v);

if (length(g->edges[e.v]) < orig) g->nE--;

}

|  |  |
| --- | --- |
| ... Adjacency List Representation | 41/144 |

Implementation of show()

void show(Graph g)

{

assert(g != NULL);

printf("V=%d, E=%d\n", g->nV, g->nE);

int i;

for (i = 0; i < g->nV; i++) {

vNode \*n = g->edges[i];

while (n != NULL) {

printf("%d-%d ",i,n->v);

n = n->next;

}

if (g->edges[i] != NULL) printf("\n");

}

}

|  |  |
| --- | --- |
| ... Adjacency List Representation | 42/144 |

Cost of operations:

initialisation: *O(V)*   (initialise *V* lists)

insert edge: *O(1)*   (insert one vertex into list)

delete edge: *O(E)*   (need to find vertex in list)

If vertex lists are sorted

insert requires search of list ⇒ *O(E)*

delete always requires a search, regardless of list order

|  |  |
| --- | --- |
| Comparison of Graph Representations | 43/144 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | array of edges | adjacency matrix | adjacency list |
| space usage | *E* | *V2* | *V+E* |
| initialise | *1* | *V2* | *V* |
| copy | *E* | *V2* | *V+E* |
| destroy | *1* | *V* | *V+E* |
| insert edge | *E* | *1* | *1* |
| remove edge | *E* | *1* | *E* |
| isolated(v)? | *E* | *V* | *1* |
| isPath(x,y)? | *ElgV* | *V2* | *V+E* |

|  |  |
| --- | --- |
| Graph Algorithms |  |

|  |  |
| --- | --- |
| Problems on Graphs | 45/144 |

Above ADT represents and manipulates graphs at low level.

What kind of problems do we want to solve on/via graphs?

is the graph fully-connected?

can we remove an edge and keep it fully-connected?

is one vertex reachable starting from some other vertex?

what is the cheapest cost path from *v* to *w*?

which vertices are reachable from *v*? (transitive closure)

is there a cycle that passes through all *V*? (tour)

is there a tree that links all vertices? (spanning tree)

what is the minimum spanning tree?

can a graph be drawn in a plane with no crossing edges?

are two graphs "equivalent"? (isomorphism)

|  |  |
| --- | --- |
| Complexity Classes | 46/144 |

Many of the above problems are solved, but some are not.

For the solved problems ...

some can be solved efficiently via a simple algorithm

some have *polynomial* worst-case performance (e.g. *V2*)

some have *exponential* worst-case performance (e.g. *2V*)

Classes of problems:

*P* = an algorithm can compute answer in polynomial time

*NP* = no *P* algorithm is known for solving the problem

|  |  |
| --- | --- |
| ... Complexity Classes | 47/144 |

The *P* and *NP* classes suggest "difficulty" of a problem.

*P* problems can be solved

by a known algorithm whose complexity has been analysed

the algorithm can solve the problem in e.g. *O(N2), O(N3)*, etc

*NP* problems have no known *P* algorithm

can potentially be solved via "brute force" techniques

evaluate all possibilities; check whether each is a solution

the space of possibilites is too big to feasibly use this approach

|  |  |
| --- | --- |
| ... Complexity Classes | 48/144 |

A problem is said to be *NP-complete* if

it is an *NP* problem

it is as difficult as the most difficult *NP* problems

Another characterisation of difficulty:

easy ... have a polynomial-time algorithm (useful in practice)

tractable ... have an algorithm, feasible only for small *N*

intractable ... no tractable algorithm is known (*NP*-hard)

non-computable ... no algorithm can exist

|  |  |
| --- | --- |
| Algorithms on Graphs | 49/144 |

Examples of graph problems in *P*:

shortest path connecting two vertices

minimum spanning tree

transitive closure

Euler tour

two-colourability

planarity (tractable)

Examples of graph problems in *NP*:

longest path connecting two vertices

three-colourability

boolean satisfiability

travelling salesman

Examples of graph problems with unknown difficulty:

graph isomorphism

|  |  |
| --- | --- |
| ... Algorithms on Graphs | 50/144 |

In the remainder of this set of notes we examine algorithms for

connectivity   (simple graphs)

path finding   (simple/directed graphs)

minimal spanning trees   (weighted graphs)

shortest path   (weighted graphs)

and look at generic methods for traversing graphs.

We begin with one of the simplest graph traversals ...

|  |  |
| --- | --- |
| Finding a Path | 51/144 |

Problem: is there a path from vertex *v* to vertex *w* ?

Approach to solving problem:

examine vertices adjacent to *v*

if any of them is *w*, then done

otherwise try vertices two edges from *v*

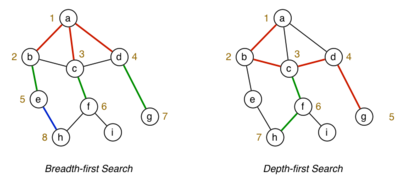
repeat looking further and further from *v*

Two different approaches to order of searching:

*breadth-first* search (BFS), *depth-first* search (DFS)

|  |  |
| --- | --- |
| ... Finding a Path | 52/144 |

Comparison of BFS/DFS search for *isPath(a,h)* ...



Both approaches ignore some edges by remembering previously visited vertices.

|  |  |
| --- | --- |
| ... Finding a Path | 53/144 |

We consider generic approaches to BFS/DFS later.

For now, we look at simple algorithms for path finding

BFS implemented via a queue of to-be-visited vertices

DFS implemented via a stack of to-be-visited vertices

also need to remember previously visited vertices

don't remember path; just determine that path exists

|  |  |
| --- | --- |
| ... Finding a Path | 54/144 |

BFS algorithm:

int isPath(Graph g, Vertex v, Vertex w)

{

int \*visited = calloc(g->nV,sizeof(int));

Queue q = newQueue();

QueueJoin(q, v);

while (!QueueEmpty(q)) {

Vertex y, x = QueueLeave(q);

if (visited[x]) continue;

visited[x] = 1;

foreach (y in neighbours(x)) {

if (y == w) return TRUE;

if (!visited[y]) QueueJoin(q, y);

}

}

return FALSE;

}

|  |  |
| --- | --- |
| ... Finding a Path | 55/144 |

DFS algorithm:

int isPath(Graph g, Vertex v, Vertex w)

{

int \*visited = calloc(g->nV,sizeof(int));

Stack s = newStack();

StackPush(s, v);

while (!StackEmpty(s)) {

Vertex y, x = StackPop(s);

if (visited[x]) continue;

visited[x] = 1;

foreach (y in neighbours(x)) {

if (y == w) return TRUE;

if (!visited[y]) StackPush(s, y);

}

}

return FALSE;

}

|  |  |
| --- | --- |
| Graph Search | 56/144 |

A common class of graph algorithms involves

walking along edges and visiting vertices

e.g. to find (shortest) path from one vertex to another

e.g. to compute reachability

We consider "graph walking"

first, from an abstract perspective

then, by looking at its applications

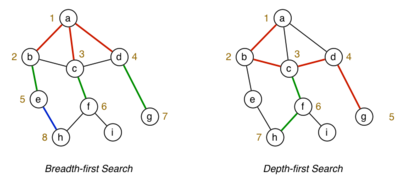
Problem: avoiding infinite loops around cyclic paths.

|  |  |
| --- | --- |
| ... Graph Search | 57/144 |

Two basic approaches to traversing a graph:

depth-first search   (follow paths first)

breadth-first search   (consider neighbours first)



|  |  |
| --- | --- |
| Depth-first Search | 58/144 |

Basic approach to depth-first search (DFS):

visit and mark current vertex

traverse each outgoing edge (if end-point not marked)

Notes:

need a mechanism for "marking" vertices

in fact, we number them as we visit them

treat the second step as a recursive DFS

|  |  |
| --- | --- |
| ... Depth-first Search | 59/144 |

Depth-first search (DFS) algorithm:

int order; // visiting order

int \*visited; // array of visiting orders

// indexed by vertex 0..V-1

void dfs(Graph g)

{

int i;

visited = malloc(g->nV\*sizeof(int));

for (i = 0; i < g->nV; i++) visited[i] = -1;

order = 0;

dfsR(g, 0);

}

void dfsR(Graph g, Vertex v)

{

visited[v] = order++;

Vertex w;

for (w = 0; w < g->nV; w++) {

if (g->edges[v][w] && visited[w] == -1)

dfsR(g, w);

}

}

|  |  |
| --- | --- |
| ... Depth-first Search | 60/144 |

DFS can also be described non-recursively (via a stack):

int \*visited; // array [0..V-1] of visiting order

void dfs(Graph g, Vertex v)

{

int i, order = 0;

visited = malloc(g->nV \* sizeof(int));

for (i = 0; i < g->nV; i++) visited[i] = -1;

Stack s = newStack();

StackPush(s,v);

while (!StackIsEmpty(s)) {

Vertex y, x = StackPop(s);

if (visited[x] != -1) continue;

visited[x] = order++;

for (y = g->nV-1; y >= 0; y--) {

if (!g->edges[x][y]) continue;

if (visited[y] == -1) StackPush(s,y);

}

}

}

|  |  |
| --- | --- |
| ... Depth-first Search | 61/144 |

The above approach yields a *DFS tree*

all vertices reachable from starting vertex

the edges to reach those vertices

effectively, gives G without cycles and alternative paths

What if the graph is not fully connected?

need to perform dfsR for each component

results in a *DFS forest*

|  |  |
| --- | --- |
| ... Depth-first Search | 62/144 |

Modified dfs to handle forests:

void dfs(Graph g)

{

int i;

for (i = 0; i < g->nV; i++)

visited[i] = -1;

order = 0;

while (order < g->nV) {

Vertex v;

for (v = 0; v < g->nV; v++)

if (visited[v] == -1) break;

dfsR(g, v);

}

}

Note: no starting vertex; always starts at vertex 0.

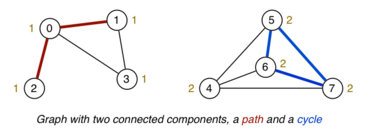
|  |  |
| --- | --- |
| DFS Examples | 63/144 |

Some problems to solve via DFS graph search

checking for the existence of a cycle

finding a path between two vertices

determining which connected component each vertex is in



|  |  |
| --- | --- |
| ... DFS Examples | 64/144 |

Check for cycles in graph:

int \*visited; // array of booleans

// indexed by vertex 0..V-1

int hasCycle(Graph g)

{

int i;

visited = malloc(g->nV\*sizeof(int));

for (i = 0; i < g->nV; i++) visited[i] = 0;

return dfsCycleCheck(g, 0);

}

int dfsCycleCheck(Graph g, Vertex v)

{

visited[v] = 1;

Vertex w;

for (w = 0; w < g->nV; w++) {

if (g->edges[v][w] && visited[w])

return 1; // found cycle

if (g->edges[v][w] && !visited[w])

return dfsCycleCheck(g, w);

}

return 0; // no cycle at v

}

|  |  |
| --- | --- |
| ... DFS Examples | 65/144 |

Check for path between two vertices *src* and *dest*:

int \*visited; // array of booleans

// indexed by vertex 0..V-1

int hasPath(Graph g, Vertex src, Vertex dest)

{

int i;

visited = malloc(g->nV\*sizeof(int));

for (i = 0; i < g->nV; i++) visited[i] = 0;

return dfsPathCheck(g, src, dest);

}

int dfsPathCheck(Graph g, Vertex v, Vertex dest)

{

visited[v] = 1;

Vertex w;

for (w = 0; w < g->nV; w++) {

if (g->edges[v][w] && w == dest)

return 1; // found path

if (g->edges[v][w] && !visited[w])

return dfsPathCheck(g, w);

}

return 0; // no path from src to dest

}

Note: we saw a non-recursive algorithm for this task earlier

|  |  |
| --- | --- |
| ... DFS Examples | 66/144 |

Assign vertices to connected components:

int \*componentOf; // array of component ids

// indexed by vertex 0..V-1

void components(Graph g)

{

int i, comp = 0;

visited = malloc(g->nV\*sizeof(int));

for (i = 0; i < g->nV; i++) componentOf[i] = -1;

order = 0;

while (order < g->nV) {

Vertex v;

for (v = 0; v < g->nV; v++)

if (componentOf[v] == -1) break;

dfsComponents(g, v, comp);

comp++;

}

// componentOf[] is now set

}

void dfsComponents(Graph g, Vertex v, int c)

{

componentOf[v] = c;

Vertex w;

for (w = 0; w < g->nV; w++) {

if (g->edges[v][w] && componentOf[w] == -1)

dfsComponents(g, w, c);

}

}

|  |  |
| --- | --- |
| Breadth-first Search | 67/144 |

Basic approach to breadth-first search (DFS):

visit and mark current vertex

visit all neighbours of current vertex

then consider neighbours of neighbours

Notes:

tricky to describe recursively

a minor variation on non-recursive DFS search works

switch the stack for a queue

|  |  |
| --- | --- |
| ... Breadth-first Search | 68/144 |

BFS algorithm (records visiting order):

int \*visited; // array [0..V-1] of visiting order

void bfs(Graph g, Vertex v)

{

int i, order = 0;

visited = malloc(g->nV \* sizeof(int));

for (i = 0; i < g->nV; i++) visited[i] = -1;

Queue q = newQueue();

QueueJoin(q,v);

while (!QueueIsEmpty(q)) {

Vertex y, x = QueueLeave(q);

if (visited[x] != -1) continue;

visited[x] = order++;

for (y = 0; y < g->nV; y++) {

if (!g->edges[x][y]) continue;

if (visited[y] == -1) QueueJoin(q,y);

}

}

}

|  |  |
| --- | --- |
| BFS Examples | 69/144 |

BFS algorithm for path checking:

int hasPath(Graph g, Vertex src, Vertex dest)

{

int \*visited = calloc(g->nV,sizeof(int));

Queue q = newQueue();

QueueJoin(q,src);

int isFound = 0;

while (!QueueIsEmpty(q) && !isFound) {

Vertex y, x = QueueLeave(q);

if (visited[x]) continue;

for (y = 0; y < g->nV; y++) {

if (!g->edges[x][y]) continue;

if (y == dest) { isFound = 1; break; }

if (!visited[y]) { QueueJoin(q,y); }

}

}

free(visited);

return isFound;

}

|  |  |
| --- | --- |
| ... BFS Examples | 70/144 |

BFS algorithm for finding shortest path

int \*visited; // array [0..V-1] of boolean

int findPath(Graph g, Vertex src, Vertex dest)

{

int i;

visited = malloc(g->nV \* sizeof(int));

for (i = 0; i < g->nV; i++) visited[i] = 0;

Vertex \*path = malloc(g->nV \* sizeof(Vertex));

Queue q = newQueue();

QueueJoin(q, src); visited[src] = 1;

int isFound = 0;

while (!emptyQ(q) && !isFound) {

Vertex y, x = QueueLeave(q);

for (y = 0; y < g->nV; y++) {

if (!g->edges[x][y]) continue;

path[y] = x;

if (y == dest) { isFound = 1; break; }

if (!visited[y]) {

QueueJoin(q, y);

visited[y] = 1;

}

}

}

if (isFound) {

// display path in dest..src order

Vertex v;

for (v = dest; v != src; v = path[v])

printf("%d-", v);

printf("%d\n", src);

}

}

|  |  |
| --- | --- |
| ... BFS Examples | 71/144 |

The above algorithm finds a "shortest" path

based on minimum # edges between *src* and *dest*.

stops with first-found path, if there are multiple ones

In many applications, edges are weighted and we want

based on minimum sum-of-weights along path *src* .. *dest*

We discuss weighted/directed graphs in the next section.

|  |  |
| --- | --- |
| Hamilton Path and Tour | 72/144 |

Hamilton path problem:

find a simple path connecting two vertices *v,w* in graph *G*

such that the path includes each *vertex* exactly once

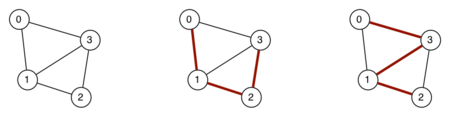
If *v = w*, then we have a Hamilton Tour.

Simple to state, but difficult to solve (*NP*-hard).

Problem named after Irish mathematician, physicist and astronomer Sir William Rowan Hamilton (1805 - 1865)

|  |  |
| --- | --- |
| ... Hamilton Path and Tour | 73/144 |

Graph and two possible Hamilton paths:



|  |  |
| --- | --- |
| ... Hamilton Path and Tour | 74/144 |

Approach:

generate all possible simple paths (using e.g. DFS)

keep a counter of vertices visited in current path

stop when find a path containing *V* vertices

Can be expressed via a recursive DFS algorithm

similar to simple path finding approach, except

keeps track of path length; succeeds if length = *v*

resets "visited" marker after unsuccessful path

|  |  |
| --- | --- |
| ... Hamilton Path and Tour | 75/144 |

Function to find Hamilton path:

int \*visited; // array [0..V-1] of bools

int hasHamiltonPath(Graph g, Vertex src, Vertex dest)

{

visited = calloc(g->nV,sizeof(int));

int res = HamiltonR(g, src, dest, g->nV-1);

free(visited);

return res;

}

int HamiltonR(Graph g, Vertex v, Vertex w, int d)

{

int t;

if (v == w) return (d == 0) ? 1 : 0;

visited[v] = 1;

for (t = 0; t < g->nV; t++) {

if (!neighbours(v,t)) continue;

if (visited[v] == 1) continue;

if (HamiltonR(g,t,w,d-1) return 1;

}

visited[v] = 0;

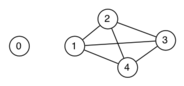
return 0;

}

|  |  |
| --- | --- |
| ... Hamilton Path and Tour | 76/144 |

Analysis: worst case requires *(V-1)!* paths to be examined

Consider a graph with isolated vertex and the rest fully-connected



Checking hasHamiltonPath(g,0,*x*) for any *x*

requires us to consider every possible path

e.g 1-2-3-4, 1-2-4-3, 1-3-2-4, 1-3-4-2, 1-4-2-3, ...

starting from any *x*, there are 3! paths ⇒ 4! total paths

there is no path of length 5 in these *(V-1)!* possibilities

There is no simpler algorithm for this task ⇒ NP-hard.

Note, however, that the above case could be solved in constant time if   
we had a fast check for 0 and *x* being in the same connected component

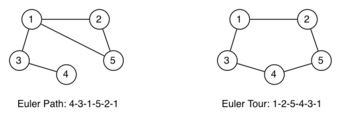
|  |  |
| --- | --- |
| Euler Path and Tour | 77/144 |

Euler path problem:

find a path connecting two vertices *v,w* in graph *G*

such that the path includes each *edge* exactly once   
(note: the path does not have to be simple ⇒ can visit vertices more than once)

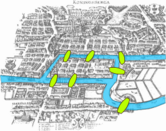
If *v = w*, the we have a Euler tour.



|  |  |
| --- | --- |
| ... Euler Path and Tour | 78/144 |

Problem first discussed by mathematician Leonard Euler (1736).

Based on a tour via bridges in Konigsberg

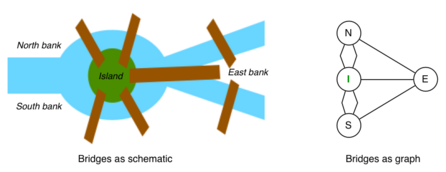


|  |  |
| --- | --- |
| ... Euler Path and Tour | 79/144 |

Restatement of problem:

is there a way to cross all the bridges of Konigsberg exactly once on a walk through the town?

treat land as nodes; bridges as edges



|  |  |
| --- | --- |
| ... Euler Path and Tour | 80/144 |

One possible "brute-force" approach:

check for each path if it's a Euler path

would result in factorial time performance

Can develop a better algorithm by exploiting:

*a graph has a Euler path* if and only if   
it is connected and exactly two vertices have odd degree

*a graph has a Euler tour* if and only if   
it is connected and all vertices have even degree

|  |  |
| --- | --- |
| ... Euler Path and Tour | 81/144 |

Assume the existence of degree(G,v) (degree of a vertex)

Function to check whether a graph has a Euler path:

int hasEulerPath(Graph g, Vertex v, Vertex w)

{

int t = degree(g,v) + degree(g,w);

if ((t % 2) != 0) return 0;

Vertex x;

for (x = 0; x < g->nV; x++) {

if (x != v && x != w) {

if ((degree(g,x) % 2) != 0)

return 0;

}

}

return 1;

}

Analysis:

assume that connectivity is already checked

assume that degree is available via *O(1)* lookup

single loop over all vertices ⇒ *O(V)*

|  |  |
| --- | --- |
| Connected Components | 82/144 |

Problems:

how many connected subgraphs are there?

are two vertices in the same connected subgraph?

Both of the above can be solved if we can

build an array, one element for each vertex V

indicating which connected component V is in

Function components(g) above does this for us.

|  |  |
| --- | --- |
| ... Connected Components | 83/144 |

Consider an application where connectivity is critical

we frequently ask questions of the kind above

but we cannot afford to run components() each time

Add a new fields to the GraphRep structure:

struct GraphRep {

...

int nC; // # connected components

int \*cc; // which component contains each vertex

... // i.e. array [0..nV-1] of 0..nC-1

}

|  |  |
| --- | --- |
| ... Connected Components | 84/144 |

With this structure, the above tasks become trivial:

// How many connected subgraphs are there?

int nConnected(Graph g)

{

return g->nC;

}

// Are two vertices in the same connected subgraph?

int inSameComponent(Graph g, Vertex v, Vertex w)

{

return (g->cc[v] == g->cc[w]);

}

|  |  |
| --- | --- |
| ... Connected Components | 85/144 |

Consider maintenance of such a graph representation:

initially, (nC == nV)   (because no edges)

adding an edge may reduce nC

removing an edge may increase nC

cc[] can simplify path checking   
(ensure v,w are in same component before starting search)

Additional maintenance cost amortised by reduced cost for nConnected() and inSameComponent()

|  |  |
| --- | --- |
| Directed Graphs |  |

|  |  |
| --- | --- |
| Directed Graphs (Digraphs) | 87/144 |

In our previous discussion of graphs:

an edge indicates a relationship between two vertices

an edge indicates nothing more than a relationship

In most real-world applications of graphs:

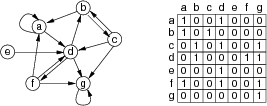
edges are directional   (*v → w ≠ w → v*)

edges have a *weight*   (cost to go from *v → w*)

In this lecture, we include direction.   
In the next lecture, we include weight.

|  |  |
| --- | --- |
| ... Directed Graphs (Digraphs) | 88/144 |

Example digraph and adjacency matrix representation:



Non-directional *⇒* symmetric matrix; directional *⇒* non-symmetric matrix   
Non-directional can also be viewed as each "edge" = two directional edges.

|  |  |
| --- | --- |
| ... Directed Graphs (Digraphs) | 89/144 |

Terminology for digraphs ...

*Directed path*:   list of *n ≥ 2* vertices *v1, v2, ... vn*

where *(vi,vi+1) ∈ edges(G)* for all *(vi,vi+1)* in list

if *v1 = vn*, we have a *directed cycle*

*Degree of vertex*:   *d(v)* = number of edges like *(v, \_)*

*Indegree* of vertex: *d-1(v)* = number of edges like *(\_, v)*

*Reachability*:   *w* is reachable from *v* if *&exists;* directed path *v,...,w*

*Strong connectivity*:   every vertex is reachable from every other vertex

*Directed acyclic graph* (DAG):   graph containing no directed cycles

|  |  |
| --- | --- |
| Digraph Applications | 90/144 |

Potential application areas:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Domain |  | Vertex |  | Edge |
| Web |  | web page |  | hyperlink |
| scheduling |  | task |  | precedence |
| chess |  | board position |  | legal move |
| science |  | journal article |  | citation |
| dynamic data |  | malloc'd object |  | pointer |
| programs |  | function |  | function call |
| make |  | file |  | dependency |

|  |  |
| --- | --- |
| ... Digraph Applications | 91/144 |

Problems to solve on digraphs:

is there a directed path from *s* to *t*? (transitive closure)

what is the shortest path from *s* to *t*? (shortest path)

are all vertices mutually reachable? (strong connectivity)

how to organise a set of tasks? (topological sort)

which web pages are "important"? (PageRank)

how to build a web crawler? (graph traversal)

|  |  |
| --- | --- |
| Digraph Representation | 92/144 |

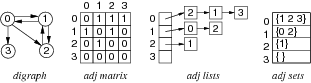
Similar set of choices as for non-directional graphs:

*V* vertices identified by *0 .. V-1*

vertex-indexed adjacency matrix   (non-symmetric)

vertex-indexed adjacency lists   (not ordered)

vertex-indexed adjacency *sets*   (some set ADT)



|  |  |
| --- | --- |
| ... Digraph Representation | 93/144 |

Costs of representations:    (where degree *d(v)* = #edges leaving *v*)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Repres. |  | Storage |  | Add edge |  | Exists edge *(v,w)*? |  | Get edges leaving *v* |
| Adj *matrix* |  | *V + V2* |  | *1* |  | *1* |  | *V* |  |
| Adj *lists* |  | *V + kE* |  | *1* |  | *d(v)* |  | *d(v)* |  |
| Adj *sets* |  | *V + sE* |  | *log(d(v))* |  | *log(d(v))* |  | *d(v)* |  |

Overall, adjacency sets representation is best

real graphs tend to be sparse   
(large number of vertices, small average degree *d(v)*)

algorithms frequently iterate over edges from *v*

|  |  |
| --- | --- |
| Counter-Example: PageRank | 94/144 |

Goal: determine which Web pages are important

Approach: ignore page contents; focus on hyperlinks

treat Web as graph: page = vertex, hyperlink = di-edge

pages with many incoming hyperlinks are important

need to computing "incoming degree" for vertices

Problem: the Web is a *very* large graph

approx. *1010* pages,   *1012* hyperlinks

Assume for the moment that we could build a graph ...

Most frequent operation in algorithm "Does edge (v,w) exist?"

|  |  |
| --- | --- |
| ... Counter-Example: PageRank | 95/144 |

Simple PageRank algorithm:

PageRank(myPage):

rank = 0;

foreach (page in the Web)

if (linkExists(page,myPage)) rank++;

Costs for computing PageRank for each representation:

adjacency matrix: *Cost = V.1*

linkExists(v,w): edge[v][w]

adjacency lists: *Cost = V.sum(d(v)) = V.E*

linkExists(v,w): search(w,list[v])

adjacency sets:   *Cost = V.sum(log(d(v))) < V.E*

linkExists(v,w): isElem(w,set[v])

But not quite feasible ... e.g. adj matrix has *1020* entries

So how to really do it?

|  |  |
| --- | --- |
| ... Counter-Example: PageRank | 96/144 |

Approach: the random web surfer

if we randomly follow links in the web ...

... more likely to re-discover pages with many inbound links

foreach (page in the Web) rank[page] = 0;

curr = random page

for (a long time) {

if (!isElem(seen, (prev,curr)))

rank[curr]++

seen = include(seen, (prev,curr))

if (random(0,100) < 85)

prev = curr

curr = choose hyperlink from curr

else

// avoids getting stuck

prev = null

curr = random page

}

Could be accomplished while we crawl web to build search index

|  |  |
| --- | --- |
| Digraph Traversal | 97/144 |

Same algorithms as for non-directed graphs:

depth-first searching (DFS)

traverse(G, *v*):

mark *v* as visited

for each vertex *w* in set *(v,w)*

traverse(G, *w*)

breadth-first searching (DFS)

traverse(G, *v*):

mark *v* as visited

enqueue(Q, *v*)

while not empty(Q) {

*n* = dequeue(Q)

for each vertex *w* in set *(n,w)*

if *w* is not already visited

mark *w* as visited

enqueue(Q, *w*)

}

|  |  |
| --- | --- |
| Example: Web Crawling | 98/144 |

Goal: visit every page on the web

Solution: breadth-first search with "implicit" graph

webCrawl(startingURL):

mark startingURL as alreadySeen

enqueue(Q, startingURL)

while not empty(Q) {

nextPage = dequeue(Q)

*visit* nextPage

foreach (hyperLink in nextPage) {

if (hyperLink not alreadySeen)

mark hyperLink as alreadySeen

enqueue(Q, hyperLink)

}

}

visit operation scans page and collects e.g. keywords and links

Assumption: web is fully connected

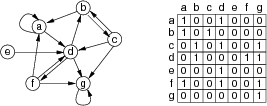
|  |  |
| --- | --- |
| Transitive Closure | 99/144 |

Goal: produce a matrix of reachability values

if *tc[s][t]* is 1, then *t* is reachable from *s*

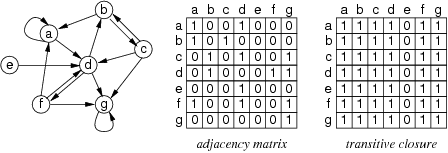
if *tc[s][t]* is 0, then *t* is not reachable from *s*

Example: which reachable *s .. t* exist in the following graph?



|  |  |
| --- | --- |
| ... Transitive Closure | 100/144 |

Transitive closure matrix for example graph:



|  |  |
| --- | --- |
| ... Transitive Closure | 101/144 |

How to compute transitive closure?

One possibility: don't compute it in advance

use DFS from start vertex *s*

check whether *t* is in visited set

ok if *reachable(s,t)* is infrequent operation

What if we have an algorithm that frequently needs it?

Would be very convenient/efficient to have:

int reachable(Graph g, Vertex s, Vertex t)

{

return (g->tc[s][t]);

}

Of course, if *V* is very large, then this is not feasible.

|  |  |
| --- | --- |
| ... Transitive Closure | 102/144 |

So, how to create a *tc[s][t]* matrix?

Observation:

// V is # vertices, A is adjacency matrix

for (s = 0; s < V; s++) {

for (t = 0; t < V; t++) {

C[s][t] = 0;

for (i = 0; i < V; i++)

if (A[s][i] && A[i][t]) C[s][t] = 1;

}

}

C[s][t] == 1 if there is a path of length 2 from *s→i→t*

|  |  |
| --- | --- |
| ... Transitive Closure | 103/144 |

If we modify the above to:

for (i = 0; i < V; i++) {

for (s = 0; s < V; s++) {

for (t = 0; t < V; t++)

if (A[s][i] && A[i][t]) A[s][t] = 1;

}

}

then we get an algorithm to convert A into a *tc*

After iteration 1, A[s][t] is 1 if

either *s→t* exists or *s→0→t* exists

After iteration 2, A[s][t] is 1 if any of the following exist

*s→t*   or   *s→0→t*   or   *s→1→t*   or   *s→0→1→t*   or   *s→1→0→t*

Etc. ... so after *Vth* iteration, A[s][t] is 1 if

there is any directed path in the graph from *s* to *t*

This technique is know as *Warshall's algorithm*

|  |  |
| --- | --- |
| ... Transitive Closure | 104/144 |

Could be added to a graph ADT as follows:

int reachable(Graph g, Vertex src, Vertex dest)

{

if (g->tc == NULL) makeClosure(g);

return g->tc[src][dest];

}

void makeClosure(Graph g)

{

int i, s, t, V = g->nV;

int \*\*tc = makeMatrix(V, V, 0);

for (s = 0; s < V; s++) {

for (t = 0; t < V; t++)

tc[s][t] = g->edges[s][t];

}

for (i = 0; i < V; i++) {

for (s = 0; s < V; s++) {

if (tc[s][i] == 0) continue;

for (t = 0; t < V; t++)

if (tc[i][t] == 1) tc[s][t] = 1;

}

}

g->tc = tc;

}

|  |  |
| --- | --- |
| ... Transitive Closure | 105/144 |

Cost analysis:

storage: additional *V2* items (each item may be 1 bit)

computation of makeClosure(): *V3* on first call to reachable()

computation of reachable(): *O(1)* after first call to reachable()

Amortization: would need many calls to reachable to justify other costs

Alternative: use DFS in each call to reachable()

Cost analysis

storage: cost of queue and set during reachable

computation of reachable(): cost of DFS: *O(V2)*   (for adj matrix)

|  |  |
| --- | --- |
| Graph Algorithms: MST, SP |  |

|  |  |
| --- | --- |
| Weighted Graphs | 107/144 |

Graphs so far have considered

edge = an association bewteen two vertices/nodes

may be a precedence in the association (directed)

Some applications require us to consider

a *cost* or *weight* of an association

modelled by assigning weights/costs to edges

Example: "map" of airline flight routes

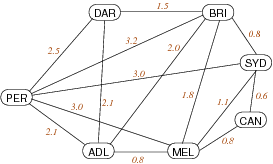
vertices = airports;   edges = flight routes

weights = distance  or  travel time  or  fare cost

Weights may apply in both directed and non-directed graphs.

|  |  |
| --- | --- |
| ... Weighted Graphs | 108/144 |

Example: major airline flight routes in Australia



Representation:   edge = direct flight;   weight = approx flying time (hours)

|  |  |
| --- | --- |
| ... Weighted Graphs | 109/144 |

Often use a geometric interpretation of weights:

edge with low weight/cost is a "short" edge

edge with hight weight/cost is a "long" edge

However, weights can have negative values   
(so the geometric interpretation is not always appropriate)

Weights typically represented as floating point values

often as values in range 0.0 *≤ w ≤* 1.0

other ranges can be mapped to this by scaling

|  |  |
| --- | --- |
| ... Weighted Graphs | 110/144 |

Weights naturally lead to minimisation-type questions, e.g.

1. What is the cheapest way to get from *A* to *B*?

i.e. what is the least cost path from *A* to *B*?

also known as the *shortest path* problem

assumes that edges are weighted and directed

2. How can I visit all towns and travel the least distance?

i.e. what set of edges covers the graph with least cost?

also known as the *minimum spanning tree* problem

assumes that edges are weighted and non-directed

|  |  |
| --- | --- |
| Weighted Graph Representation | 111/144 |

Weights can easily be added to:

adjacency matrix representation   (bool *→* float)

adjacency lists representation   (add float to each node)

But does not fit naturally with adjacency sets.  
(unless you want to introduce e.g. fuzzy sets)

An alternative representation useful in this context:

edge list representation   (list of *(s,t,w)* triples)

All representations work whether edges are directed or not.

|  |  |
| --- | --- |
| ... Weighted Graph Representation | 112/144 |

Representation of vertices and edges with weights:

typedef int Vertex; // unchanged from before

typedef struct edgeRep {

Vertex src; // start point

Vertex dest; // end point

float weight; // cost of edge

} Edge;

Edge mkEdge(Vertex src, Vertex dest, float weight)

{

Edge new;

new.src = src;

new.dest = dest;

new.weight = weight;

return new;

}

int compareE(Edge e1, Edge e2)

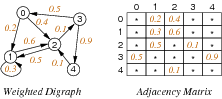
{

return e1.weight - e2.weight;

}

|  |  |
| --- | --- |
| ... Weighted Graph Representation | 113/144 |

Adjacency matrix representation with weights:



|  |  |
| --- | --- |
| ... Weighted Graph Representation | 114/144 |

Adjacency matrix implementation (if negative weights allowed):

// since 0 is a valid weight, can't use it for "no edge"

// need a distinguished value to indicate "no edge"

#define NO\_EDGE MAXFLOAT

typedef struct graphRep GraphRep;

struct graphRep {

int nV; // #vertices

int nE; // #edges

float \*\*edges; // matrix of floats

};

Graph newGraph(int nV) // asserts omitted for brevity

{

float \*\*e = malloc(nV\*sizeof(float \*));

int i, j;

for (i = 0; i < nV; i++) {

e[i] = malloc(nV\*sizeof(float));

for (j = 0; j < nV; j++)

e[i][j] = NO\_EDGE;

}

Graph g = malloc(sizeof(struct graphRep));

g->nV = nV; g->nE = 0; g->edges = e;

return g;

}

void insert(Graph g, Edge e)

{

Vertex v = e.src, w = e.dest;

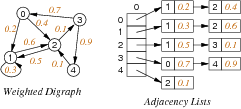
if (G->edges[v][w] == NO\_EDGE) g->nE++;

g->edges[v][w] = e.weight;

}

|  |  |
| --- | --- |
| ... Weighted Graph Representation | 115/144 |

Adjacency lists representation with weights:



|  |  |
| --- | --- |
| ... Weighted Graph Representation | 116/144 |

Adjacency lists implementation:

typedef struct vNode \*VList;

struct vNode { Vertex v; float weight; vList next; };

typedef struct graphRep GraphRep;

struct graphRep {

int nV; // #vertices

int nE; // #edges

VList \*edges; // array of lists

};

Graph newGraph(int nV)

{

VList \*e = malloc(nV \* sizeof(VList));

int i;

for (i = 0; i < nV; i++) e[i] = NULL;

Graph g = malloc(sizeof(struct graphRep));

g->nV = nV; g->nE = 0; g->edges = e;

return g;

}

void insertE(Graph g, Edge e)

{

Vertex v = e.src, w = e.dest;

int orig = length(g->edges[v]);

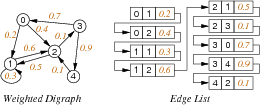
g->edges[v] = insertVList(g->edges[v], w, e.weight);

if (length(g->edges[v]) > orig) g->nE++;

}

|  |  |
| --- | --- |
| ... Weighted Graph Representation | 117/144 |

Edge list representation with weights:



Note: not very efficient for use in processing algorithms, but does   
give a possible representation for min spanning trees or shortest paths

|  |  |
| --- | --- |
| ... Weighted Graph Representation | 118/144 |

Edge list implementation:

typedef struct eNode \*EList;

struct eNode { Edge e; vList next; };

typedef struct graphRep GraphRep;

struct graphRep {

int nV; // #vertices

int nE; // #edges

EList edges; // list of edges

};

Graph newGraph(int nV)

{

Graph g = malloc(sizeof(struct graphRep));

g->nV = nV; g->nE = 0; g->edges = NULL;

return g;

}

void insertE(Graph g, Edge e)

{

int orig = length(g->edges);

g->edges = insertEList(g->edges, e);

if (length(g->edges) > orig) g->nE++;

}

|  |  |
| --- | --- |
| Minimum Spanning Trees | 119/144 |

Reminder: *Spanning tree* *ST* of graph *G(V,E)*

*ST* is a subgraph of *G*   (*G'(V,E')* where *E' ⊂ E*)

*ST* is *connected* and *acyclic*

*Minimum spanning tree* *MST* of graph *G*

*MST* is a spanning tree of *G*

sum of edge weights is no larger than any other ST

Problem: how to (efficiently) find MST for graph *G*?

|  |  |
| --- | --- |
| ... Minimum Spanning Trees | 120/144 |

Origins of MST problem:

Otakar Boruvka, electrical engineer in 1926

most economical construction of electric power network

Some modern applications of MST:

network layout: telephone, electric, computer, road

cluster analysis in data mining applications

MST is an archetype for problem-solving in combinatorial optimisation.

We consider two algorithms that are examples of *greedy* approaches.   
(in this context "Greed is good. Greed is right. Greed works." *G.Gecko*)

|  |  |
| --- | --- |
| ... Minimum Spanning Trees | 121/144 |

Brute force solution:

MSTree findMST(Graph g)

{

MSTree t, best; float bestCost = MAXFLOAT;

foreach (t in AllSpanningTrees(g)) {

if (cost(t) < bestCost) {

best = t; bestCost = cost(t);

}

}

return best;

}

Example of *generate-and-test* approach; not appropriate here.

Not appropriate because [#spanning trees](http://en.wikipedia.org/wiki/Kirchhoff%27s_theorem) is potentially large.

|  |  |
| --- | --- |
| ... Minimum Spanning Trees | 122/144 |

Simplifying assumptions:

edges in *G* are not directed   (MST for digraphs is hard)

no edge weights are negative

all edge weights are distinct

If edges may have same weight:

MST may not be unique   (not necessarily a problem)

if we have to choose between edges with same weight ...

greedy algorithms might make wrong choice

might not end up with minimum cost solution

|  |  |
| --- | --- |
| ... Minimum Spanning Trees | 123/144 |

If we phrase MST problem as function:

MSTree findMST(Graph g) { ... }

what is the return type (MSTree)?

Ultimately, MSTree contains a set of edges.

MST algorithms allow choices of intermediate representation.

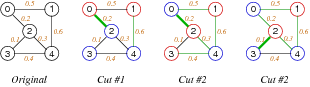
Appropriate representation can make a huge efficiency difference.

|  |  |
| --- | --- |
| ... Minimum Spanning Trees | 124/144 |

*Cut*: partition of vertices into two disjoint sets

*Crossing edge*: any edge between vertices from diff sets

Examples:



*Cut property*: any minimal crossing edge is part of MST

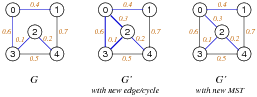
|  |  |
| --- | --- |
| ... Minimum Spanning Trees | 125/144 |

*Cycle property*

consider a graph *G* with MST

*G'* = add an edge to *G* to include cycle in "MST"

remove maximal edge from cycle *⇒* MST for *G'*



|  |  |
| --- | --- |
| Kruskal's Algorithm | 126/144 |

One approach to computing MST for graph *G(V,E)*:

start with empty MST

consider edges in increasing weight order

add edge if it does not form a cycle in MST

repeat until *V-1* edges are added

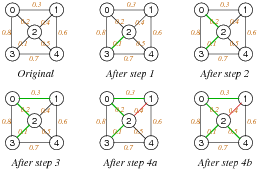
Critical operations:

iterating over edges in weight order

checking for cycles in a graph

|  |  |
| --- | --- |
| ... Kruskal's Algorithm | 127/144 |

Execution trace of Kruskal's algorithm:



|  |  |
| --- | --- |
| Implementation of Kruskal's Algorithm | 128/144 |

C-like description of algorithm:

MSTree kruskalFindMST(Graph g)

{

Graph mst = newGraph(); //MST initially empty

EdgeList eList; //sorted list of edges

int i; Edge e; int eSize = sizeof(Edge);

edges(eList, g->nE, g);

eList = qsort(sorted, g->nE, eSize, compareE);

for (i = 0; mst->nE < g->nV-1; i++) {

e = eList[i];

insertE(mst, e);

if (hasCycle(mst)) removeE(mst, e);

}

}

|  |  |
| --- | --- |
| ... Implementation of Kruskal's Algorithm | 129/144 |

Rough cost analysis:

sorting edge list is *O(E log E)*

at least *V* iterations over sorted edges

on each iteration ...

getting next lowest cost edge is *O(1)*

checking whether adding it forms a cycle has cost ??

Possibilities for cycle checking:

use DFS ... but too expensive

use Union-Find data structure from Sedgewick ch.1!

|  |  |
| --- | --- |
| Prim's Algorithm | 130/144 |

Another approach to computing MST for graph *G(V,E)*:

start from any vertex *s* and empty MST

choose edge not already in MST to add to MST

must not contain a self-loop

must connect to a vertex already on MST

must have minimal weight of all such edges

repeat until MST covers all vertices

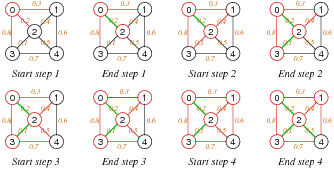
Critical operations:

checking for vertex being connected in a graph

finding min weight edge in a set of edges

|  |  |
| --- | --- |
| ... Prim's Algorithm | 131/144 |

Execution trace of Prim's algorithm:



|  |  |
| --- | --- |
| Implementation of Prim's Algorithm | 132/144 |

C-like description of algorithm:

MSTree primFindMST(Graph g)

{

EdgeSet mst = {}; //MST initially empty

VertexSet vSet = {0}; // start vertex

EdgeSet fringe = {}; //edges at "fringe"

Vertex curr, s, t; Edge e; float w;

fringe = edgesAt(0);

while (card(vSet) < g->nV) {

find e in fringe with minimum cost

fringe = exclude(fringe, e)

(s,curr,w) = e

vSet = include(vSet, curr)

mst = include(mst, e)

foreach (e in edgesAt(curr)) {

(s,t,w) = e // s == curr

if (!isElem(t,vSet))

fringe = include(fringe,e)

}

}

}

|  |  |
| --- | --- |
| ... Implementation of Prim's Algorithm | 133/144 |

find e in fringe with minimum cost

can be done efficiently via a priority queue

foreach (e in edgesAt(curr))

straightforward for adjacency matrix/list

Need efficient implementation of sets

In fact, this algorithm is a variation of graph search

variation is use of priority queue   (cf. LIFO or FIFO)

|  |  |
| --- | --- |
| ... Implementation of Prim's Algorithm | 134/144 |

Rough cost analysis:

*V* iterations of outer loop

in each iteration ...

find min cost node, is *O(1)* with e.g. PQueue

set operations could be *O(1)* with e.g. bit-map

iteration over edges is *O(d(curr))* e.g. max *V*

for each outgoing edge ...

check membership in vSet could be *O(1)* e.g. bit-map

overall, with the right data structures, could be *O(V2)*

|  |  |
| --- | --- |
| Shortest Path | 135/144 |

*Weight* of a path *p* in graph *G*

sum of weights on edges along path (*weight(p)*)

*Shortest path* between vertices *s* and *t*

a simple path *p* where *s = first(p), t = last(p)*

no other simple path *q* has *weight(q) < weight(p)*

Problem: how to (efficiently) find *shortestPath(G,s,t)*?

Assumptions: weighted digraph, no negative weights.

|  |  |
| --- | --- |
| ... Shortest Path | 136/144 |

Shortest-path is useful in a wide range of applications

robot navigation

finding routes in maps

optimal pipelining of VLSI chip

routing in data/computer networks

Flavours of shortest-path

source-target   (shortest path from *s* to *t*)

single-source   (shortest paths from *s* to all other *V*)

all-pairs (shortest paths for all *(s,t)* pairs)

|  |  |
| --- | --- |
| Single-source Shortest Path | 137/144 |

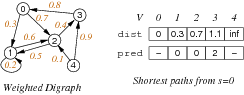
Given: weighted digraph *G*, source vertex *s*

Result: shortest paths from *s* to all other vertices

dist[] *V*-indexed array of distances from *s*

pred[] *V*-indexed array of predecessors in shortest path

Example:



Note: shortest paths can be viewed as tree rooted at *s*

|  |  |
| --- | --- |
| Edge Relaxation | 138/144 |

Assume: dist[] and pred[] as above   
  (but not necessarily containing data for shortest paths)

*Relaxation* along edge *e* from *v* to *w*

dist[v] is length of some path from *s* to *v*

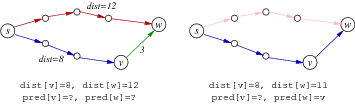
dist[w] is length of some path from *s* to *w*

if *e* gives shorter path *s* to *w* via *v*,   
   then update dist[w] and pred[w]

Relaxation updates data on *w* if we find a shorter path to *s*.

|  |  |
| --- | --- |
| ... Edge Relaxation | 139/144 |

Example:



Implementation of edge relaxation:

if (dist[v] + e.weight < dist[w]) {

dist[w] = dist[v] + e.weight;

pred[w] = v;

}

|  |  |
| --- | --- |
| Dijkstra's Algorithm | 140/144 |

One approach to solving single-source shortest path ...

Data: *G*, *s*, dist[], pred[], and ...

*vSet*: set of vertices whose shortest path from *s* is known

Algorithm:

initialise *vSet* to *{s}*

initialise dist[] to all *∞*, except dist[s]=0

choose an edge *e* where

*e* connects *v* in *vSet* to *w* not in *vSet*

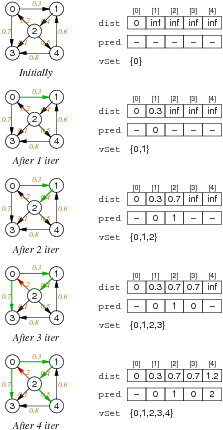
*e* is the minimum weight such edge

relax along *e* and add *w* to *vSet*

repeat until *vSet* contains all vertices connected to *s*

|  |  |
| --- | --- |
| ... Dijkstra's Algorithm | 141/144 |

Execution trace of Dijkstra's algorithm:



Produces a "spreading tree" of shortest paths from *s*

|  |  |
| --- | --- |
| Implementation of Dijkstra's Algorithm | 142/144 |

C-like description of algorithm:

MSTree shortestPath(Graph g, Vertex s)

{

VertexSet vSet = {s}; // start vertex

float dist[g->nV] = {inf,};

Vertex pred[g->nV] = {NULL,};

Vertex v, w; Edge e; float wt;

while (card(vSet) < g->nV) {

find e=(v,w,wt) satisfying

isElem(v,vSet)

&& !isElem(w,vSet)

&& minimum(dist[v]+wt)

dist[w] = dist[v] + wt; pred[w] = v;

vSet = include(vSet, w)

}

}

|  |  |
| --- | --- |
| ... Implementation of Dijkstra's Algorithm | 143/144 |

Implementations of "find e=(v,w,wt) satisfying ..."

try all *e ∈ E* ... cost = *O(VE)*

(Djiskstra) maintain the following:

for *v ∈ vSet*, dist[v] is shortest path *s* to *v*

for *w∉vSet*, dist[w] minimises dist[p]+weight[p-w]

which means ... cost = *O(V2)*

can find e in *V* steps   (scan dist[])

update dist[] in max *V* steps   (update all neighbours)

use a PQueue to find edge to relax   (see next page)

|  |  |
| --- | --- |
| ... Implementation of Dijkstra's Algorithm | 144/144 |

C-like description of more efficient algortihm:

MSTree shortestPath(Graph g, Vertex s)

{

VertexSet vSet = {}; // visited vertices

PQueue todo = newPQueue(); // edges to be considered

float dist[g->nV] = {inf,}; dist[s] = 0.0;

Vertex pred[g->nV] = {NULL,};

Vertex v, w, s, t; Edge e; float wt;

PQueueJoin(todo, s, dist[s]);

while (!PQueueIsEmpty(todo)) {

v = PQueueLeave(todo);

if (isElem(vSet,v)) continue;

vSet = include(vSet, v)

foreach (e=(v,w,wt) in edgesAt(v)) {

if (dist[v] + wt < dist[w]) {

dist[w] = dist[v] + wt;

pred[w] = v;

PQueueJoin(todo, w, dist[w]);

}

}

}

}

Produced: 17 Aug 2014