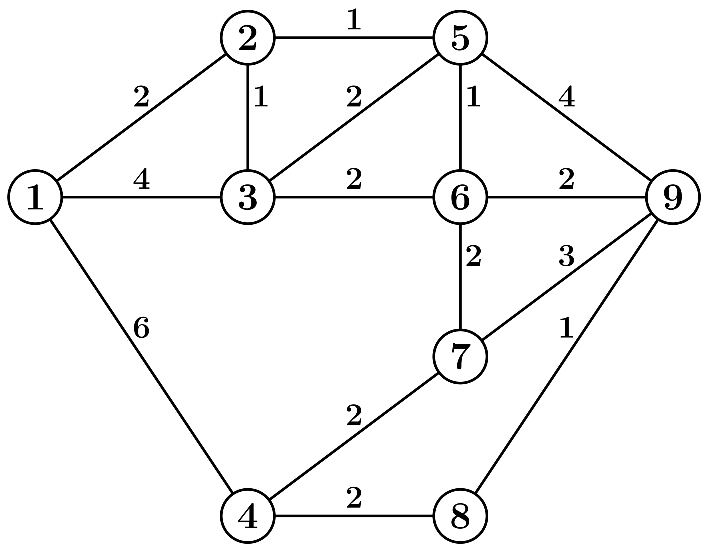
Graphs
— T.10 —

Summary:

- Introduction (definitions and examples)
- Data structures for graphs
- Graph traversal
- Connected components
- Connected components using the union-find data structure
- All paths
- All cycles
- Shortest path
- Minimum spanning tree

The examples on the second half part of this lecture use the following "test" graph.



Bibliography

Recommended bibliography for this lecture:

- **The Algorithm Design Manual**, Steven S. Skiena, second edition, Springer, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011

Useful external stuff:

- **Undirected graphs (slides)**, Sedgewick and Wayne, Princeton University, USA.
- **Directed graphs (slides)**, Sedgewick and Wayne, Princeton University, USA.
- **Union-find (slides)**, Sedgewick and Wayne, Princeton University, USA.
- **Shortest paths (slides)**, Sedgewick and Wayne, Princeton University, USA.
- **Minimum spanning trees (slides)**, Sedgewick and Wayne, Princeton University, USA.

Introduction (part 1, initial definitions)

A **graph** G is a pair (V, E), where V is a set of **vertices** and E is a set of **edges**.

It is usual to consider that a vertex is a point in a plane (or in space), but that is not strictly necessary; a vertex is merely an object (a point, a word, a number, a book, a country, *et cetera*). A vertex usually has one or more properties. For example, a point usually has coordinates. The number of vertices of G is the number of elements of V and is denoted by $|V|$. It is usual to give numbers to the vertices, so that $V = \{v_1, \ldots, v_{|V|}\}$.

An edge e is a pair (v_i, v_j) of two vertices. The set of edges is a binary relation on V (the binary relation can be considered to be "connected to," so that an edge connects two vertices). It is also common to associate a property to an edge. For example, the vertices may represent cities and the edges may represent distances between pairs of cities. The number of edges of G is the number of elements of E and is denoted by $|E|$. Like the vertices, it is usual to give numbers to the edges, so that $E = \{e_1, \ldots, e_{|E|}\}$. We also have $e_i = (v_{1i}, v_{2i})$, i.e., edge e_i connects the vertices v_{1i} and v_{2i}.

A graph is **directed** (called a **digraph**) when the edges are **ordered** pairs of vertices. For directed graphs, edge e_i **departs from** (or **leaves**, or is **incident from**) vertex v_{1i}, and **arrives at** (or **enters**, or is **incident to**) vertex v_{2i}. In drawings, vertices are usually represented by dots or circles, and edges are represented by line or curve segments terminated by an arrow (sometimes the arrow is drawn in the middle of the segment instead of at its end).

A graph is **undirected** when the edges are **unordered** pairs of vertices, i.e., (e_i, e_j) is considered to be the same as (e_j, e_i). For undirected graphs, edge e_i is **incident on** both v_{1i} and v_{2i}. In drawings the edges do not have arrows.

The **degree** of a vertex of an undirected graph is the number of edges incident on it. For directed graphs the **in-degree** and **out-degree** of a vertex are the number of edges entering and leaving it, and the **degree** is the sum of the two.

Introduction (part 2, more definitions)

A **weighted** graph is one where each edge e_i has a weight w_i (a property of the edge). In an **unweighted** graph that does not happen. In drawings, the weight is written near the edge (near its arrow for directed graphs).

A **labeled** graph is one where each vertex v_i has a label l_i (a property of the vertex) that uniquely identifies it. In an **unlabeled** graph that does not happen. In drawings, the label is written either inside the vertex or near it.

A graph is **simple** if there do not exist more that one edge connecting any pair of vertices and if there do not exist self-loops, edges of the form (v, v). If that does not happen, the graph is **non-simple**.

A graph is **dense** if "is has many edges" and is **sparse** if it "has few edges." These definitions are a bit vague (on purpose). Since the number of vertex pairs of a graph with n vertices can be as high as $n(n-1)/2$, one possible characterization of a dense graph is one in which the number of edges is a sizable fraction of $n(n-1)/2$, while a sparse graph might be one in which the number of edges is a reasonably small multiple of n.

A **path** of length L starting at vertex v_s and ending in vertex v_d is a sequence $v_s, v_i, v_j, \ldots, v_d$ of $L+1$ vertices such that there exists an edge between consecutive vertices of the sequence. The length of the path is then the number of edges it contains. By convention, 0-length paths (of course with $v_d = v_s$) always exist. The path is **simple** if the sequence of vertices does not have repeated vertices. A path is a **cycle** if $L > 0$ and if $v_d = v_s$ (the cycle is simple if it does nor contain repeated vertices). For a weighted graph, the weight of a path is the sum of the weights of its edges, The weight of a cycle is defined in a similar way.

The vertices v_s and v_d are **connected** if there exists a path that starts at v_s and ends at v_d. A graph is **connected** if all its pairs of vertices are connected.

A graph is **acyclic** if it does not contain any cycles. Otherwise it is a **cyclic** graph. A **tree** is a connected, acyclic, undirected graph.

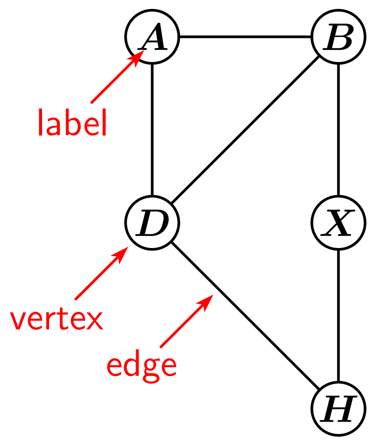
Introduction (part 3, even more definitions)

The set formed by the arrival vertices of the edges that depart from a vertex $\boldsymbol{v}$, excluding $\boldsymbol{v}$ itself, is called the neighborhood of $\boldsymbol{v}$. The definition can be extended to the neighborhood of a set $\boldsymbol{V'}$ of vertices: it is the set formed by the arrival vertices of the edges that depart from one of the vertices of $\boldsymbol{V'}$ and which do not arrive on a vertex of $\boldsymbol{V'}$.

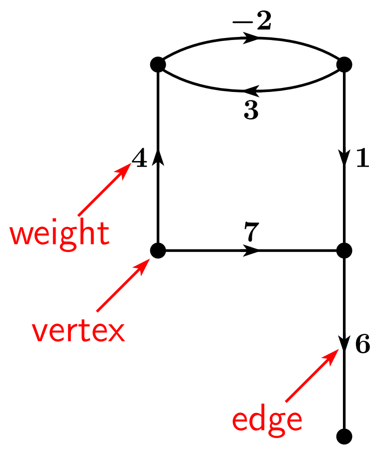
A spanning tree of a connected graph is a subgraph that is i) connected, ii) acyclic, and iii) includes all vertices. Conditions i) and ii) are the "tree" part and condition iii) is the spanning part of the name "spanning tree."

Introduction (part 4, examples)

Undirected and labeled

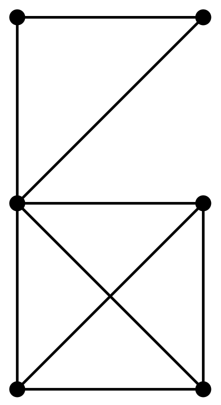


Directed and weighted

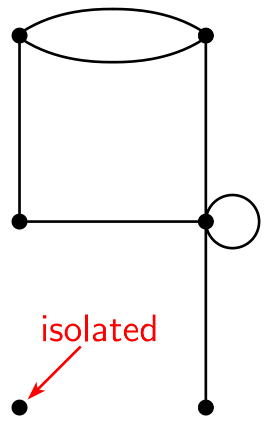


What is the degree of each vertex?

Simple

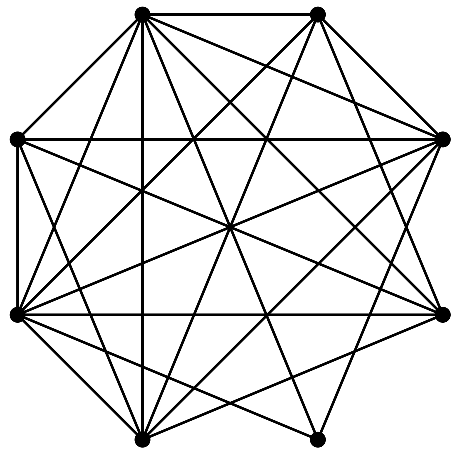


Non-simple

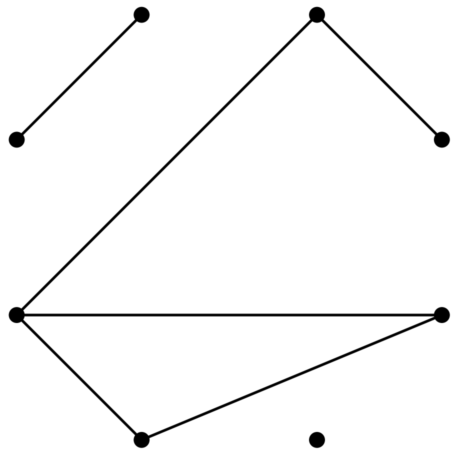


How many connected components?

Dense and connected

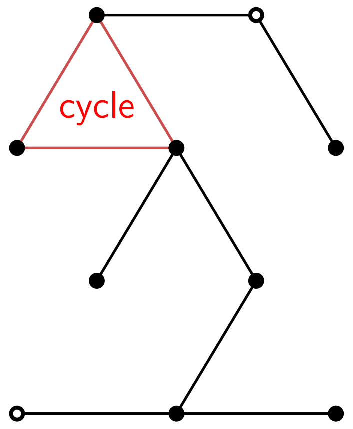


Sparse and unconnected

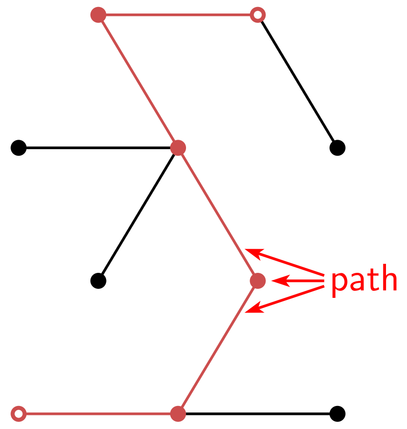


Average node degree?

Cyclic



Acyclic



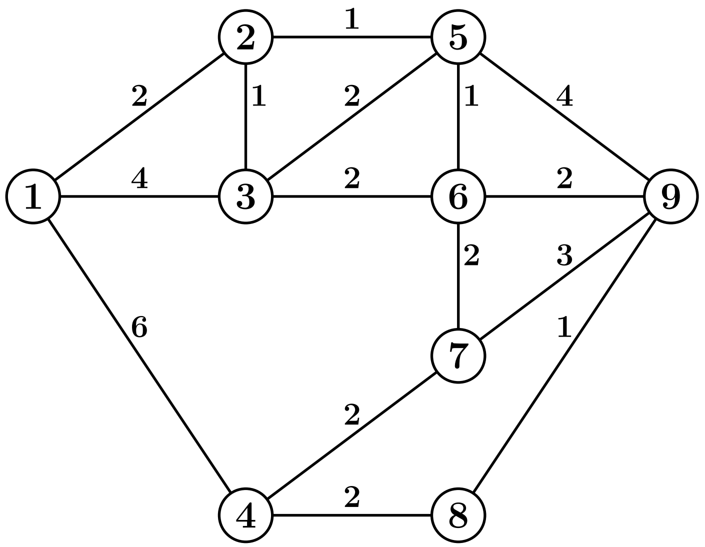
How many paths?

Data structures for graphs (part 1, adjacency matrix)

The adjacency matrix is a simple data structure (a $|V| \times |V|$ matrix) capable of storing all edge information of a simple graph (directed or undirected). It should be used when it is known that the graph has a small number of vertices, or when it is known that it is dense.

For unweighted graphs, the element in row i and column j of the adjacency matrix records the presence (1) or absence (0) of an edge starting at vertex i and ending at vertex j (for undirected graphs the adjacency matrix is a symmetric matrix). For weighted graphs it stores the weight of the edge starting at vertex i and ending at vertex j, if that edge exists, and stores an invalid weight value (for example, a NaN when the weight is a floating point number or zero if the weights are known to be positive) if it does not exist.

In the following example the adjacency matrix for the graph of the left is shown on the right (in this case the small zeros signal the absence of the edge):



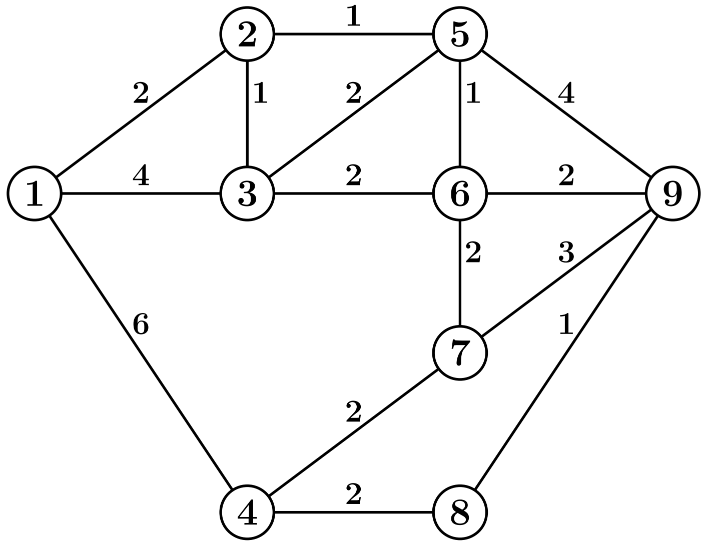
from\to	1	2	3	4	5	6	7	8	9
1	0	2	4	6	0	0	0	0	0
2	2	0	1	0	1	0	0	0	0
3	4	1	0	0	2	2	0	0	0
4	6	0	0	0	0	0	2	2	0
5	0	1	2	0	0	1	0	0	4
6	0	0	2	0	1	0	2	0	2
7	0	0	0	2	0	2	0	0	3
8	0	0	0	2	0	0	0	0	1
9	0	0	0	0	4	2	3	1	0

Data structures for graphs (part 2, adjacency lists)

The adjacency lists is another simple data structure (one linked list of edges per vertex) that is also capable of storing all edge information of a simple graph (directed or undirected). It should be used when it is known that the graph has a large number of vertices and that it is sparse (that is usually the case).

In this way to represent a graph, each vertex maintains a linked list of the edges that leave the vertex. For unweighted graphs, the nodes of the list only need to store the number of, or pointer to, the other vertex and, of course, the pointer to the next node of the linked list. For weighted graphs, they also store the weight.

In the following example the adjacency lists for the graph of the left are shown on the right (for each linked list node it is displayed first the other vertex number and then the edge weight):



1	$(4,6) \rightarrow (3,4) \rightarrow (2,2)$
2	$(5,1) \rightarrow (3,1) \rightarrow (1,2)$
3	$(6,2) \rightarrow (5,2) \rightarrow (2,1) \rightarrow (1,4)$
4	$(8,2) \rightarrow (7,2) \rightarrow (1,6)$
5	$(9,4) \rightarrow (6,1) \rightarrow (3,2) \rightarrow (2,1)$
6	$(9,2) \rightarrow (7,2) \rightarrow (5,1) \rightarrow (3,2)$
7	$(9,3) \rightarrow (6,2) \rightarrow (4,2)$
8	$(9,1) \rightarrow (4,2)$
9	$(8,1) \rightarrow (7,3) \rightarrow (6,2) \rightarrow (5,4)$

Note that the order of the edges in each list is arbitrary. Note also that for undirected graphs the size of a linked list is equal to the degree of its vertex (it is equal to the out-degree of the vertex for directed graphs).

Graph traversal (part 1, short description)

Traversing a graph is a fundamental operation. One starts at a given vertex and one moves along the edges until one visits all possible vertices. This can be done in a depth-first style or in a breadth-first style. Both are useful. The next slide explains with C code how to do this. That code uses the following data types and support functions.

```
deque *create_deque(int max_size); // deque creation
void destroy_deque(deque *dq);     // deque destruction
int get_lo(deque *dq);             // dequeue
int get_hi(deque *dq);             // pop
void put_hi(deque *dq,int v);      // push or enqueue

typedef struct edge
{
  struct edge *next; // next edge node
  int vertex_number; // vertex number
  int weight;        // edge weight
}
edge;
```

```
typedef struct vertex
{
  edge *out_edges;  // adjacency list head
  int mark;         // for graph traversals
}
vertex;

typedef struct graph
{
  vertex *vertices; // array of vertices (pointer)
  int n_vertices;   // number of vertices
}
graph;
```

To avoid visiting the same vertex more than once, a vertex is marked when it is touched for the first time (touching amounts to put the vertex in either a stack or a queue), and, later on, when it is visited and its edges are followed (to touch other vertices) the mark is given its final value (sequential visitation number). The following function is used to initialize the marks:

```
void mark_all_vertices(graph *g,int mark)
{
  for(int i = 0;i < g->n_vertices;i++)
    g->vertices[i].mark = mark;
}
```

Graph traversal (part 2, code)

Here is the code for a depth-first and a breadth-first traversal of a graph.

```c
void depth_first(graph *g,int initial_vertex)
{
  deque *dq;
  edge *e;
  int i,n;

  mark_all_vertices(g,-1);
  dq = create_deque(g->n_vertices);
  put_hi(dq,initial_vertex);
  n = 0;
  while(dq->size > 0)
  {
    i = get_hi(dq); // pop
    g->vertices[i].mark = ++n; // visit
    for(e = g->vertices[i].out_edges;e != NULL;e = e->next)
      if(g->vertices[e->vertex_number].mark == -1)
      { // touch
        g->vertices[e->vertex_number].mark = 0;
        put_hi(dq,e->vertex_number); // push
      }
  }
  destroy_deque(dq);
}
```

```c
void breadth_first(graph *g,int initial_vertex)
{
  deque *dq;
  edge *e;
  int i,n;

  mark_all_vertices(g,-1);
  dq = create_deque(g->n_vertices);
  put_hi(dq,initial_vertex);
  n = 0;
  while(dq->size > 0)
  {
    i = get_lo(dq); // dequeue
    g->vertices[i].mark = ++n; // visit
    for(e = g->vertices[i].out_edges;e != NULL;e = e->next)
      if(g->vertices[e->vertex_number].mark == -1)
      { // touch
        g->vertices[e->vertex_number].mark = 0;
        put_hi(dq,e->vertex_number); // enqueue
      }
  }
  destroy_deque(dq);
}
```

Notice how a small change makes a big difference in visiting order. For the graph used as example a few slides ago, starting at the vertex number $\mathbf{1}$ a depth-first traversal visits the vertices in the order $\mathbf{1, 2, 5, 6, 7, 9, 8, 3, 4}$, and a breadth-first traversal visits them in the order $\mathbf{1, 4, 3, 2, 8, 7, 6, 5, 9}$.

Connected components

Either version of the graph traversal code can be easily modified so that the new code

- finds all vertices connected to a given one,
  [Solution. The code of the previous slide already does this: vertices that at the end have a mark of -1 are not connected to the starting vertex. Marking the visited vertices with the vertex visiting order is not necessary!]
- subdivides a graph into connected components,
  [Solution. Mark initially all vertices with -1. Set a region number also to -1. Then, do the following until all vertices have a non-negative mark: Increase the region number by one; Find a vertex with a negative mark; Apply one of the algorithms to mark all vertices connected to it (when they are first touched) with the current region number.]
- determines if the graph is cyclic or acyclic.
  [Solution. Apply the method of the previous item but stop it saying the graph is cyclic if the algorithm touches a vertex that has already been touched and that has the same region number!]

The breadth-first variant can also be easily modified to find the smallest distance, in terms of number of edges traveled, between a starting vertex and all other vertices connected to it.
[Solution. Mark each touched vertex with one more than the mark of the vertex being visited.] (The depth-first variant can also do this, but in a more complex way.)

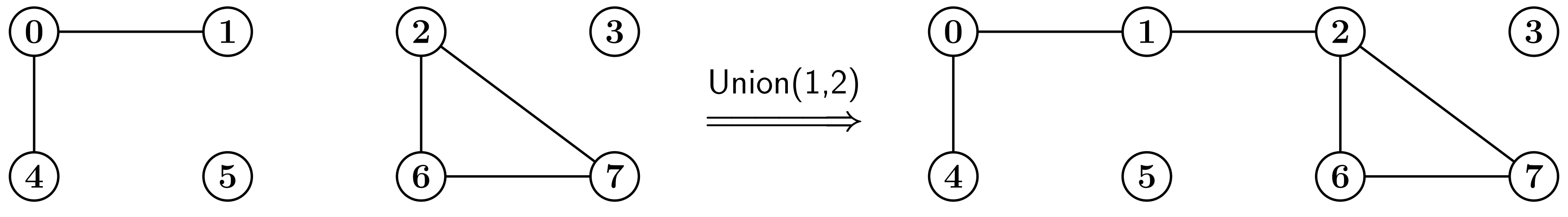
Connected components using the union-find data structure (part 1, problem)

The union-find data structure is an efficient way to keep track of the connected components of an **undirected** graph. (There are other application of this data structure.) Its main operations are

- **[Union]** replace the connected components containing vertices p and q with their union (if the vertices belong to the same component then there is nothing to to, otherwise join the two components), and
- **[Find]** given a vertex p, find the vertex q that represents the connected component that p belongs to.

To keep the connectivity information of a graph up-to-date for each edge insertion do an union operation of its incident vertices. Checking if two vertices are connected amounts to check if the representatives of their connected components are the same.

In the following example, an edge between vertices 1 and 2 is inserted in the graph. In this case the corresponding union operation joints two connected components.



Four connected components
$\{0,1,4\},\{2,6,7\},\{3\},\{5\}$

Three connected components
$\{0,1,2,4,6,7\},\{3\},\{5\}$

One of the vertices of each connected component will be the representative vertex of that component. For a graph without edges, each vertex is the representative vertex of its component (each component has only one vertex). Each union operation just makes the representative of the component of its second argument be the representative of the first.

Connected components using the union-find data structure (part 2a, code)

In order to keep track of the representative (vertex number) of each connected component and of the number of connected components, the `vertex` and `graph` structures need to be modified as follows.

```
typedef struct vertex
{
  edge *out_edges;    // adjacency list head
  int mark;           // for graph traversals
  int representative; // vertex number of representative
}
vertex;

typedef struct graph
{
  vertex *vertices;           // array of vertices (pointer)
  int n_vertices;             // number of vertices
  int n_connected_components; // number of connected components
}
graph;
```

The following extra code should be placed in the function that creates an empty graph.

```
  g->n_connected_components = n_vertices;
  for(int i = 0;i < n_vertices;i++)
    g->vertices[i].representative = i;
```

Connected components using the union-find data structure (part 2b, code)

As stated previously, the union operation amounts to change the representative of one connected region to be the representative of another connected region. Thus, in order to find the vertex number of the representative vertex of a connected region it is this necessary to follow the representative numbers until there is no change. To speed up subsequent queries, the representative numbers of all vertices that were visited in the query should be updated with the current representative number (that is called path compression). The following function does all this.

```
int find_representative(graph *g,int vertex_number)
{
  int i,j,k;

  // find (linked list done with index numbers!)
  for(i = vertex_number;i != g->vertices[i].representative;i = g->vertices[i].representative)
    ;
  // path compression
  for(j = vertex_number;j != i;j = k)
  {
    k = g->vertices[j].representative;
    g->vertices[j].representative = i;
  }
  return i;
}
```

Connected components using the union-find data structure (part 2c, code)

Finally, the function that adds an edge to the graph needs to be updated to maintain the union-find data.

```
int add_edge(graph *g,int from,int to,int weight)
{
  edge *e;

  assert(from >= 0 && from < g->n_vertices && to >= 0 && to < g->n_vertices && from != to);
  for(e = g->vertices[from].out_edges;e != NULL && e->vertex_number != to;e = e->next)
    ;
  if(e != NULL)
    return 0;
  e = create_edge();
  e->next = g->vertices[from].out_edges;
  g->vertices[from].out_edges = e;
  e->vertex_number = to;
  e->weight = weight;

  int fi = find_representative(g,from);
  int ti = find_representative(g,to);
  if(fi != ti)
  { // union
    g->vertices[ti].representative = fi;
    g->n_connected_components--;
  }

  return 1;
}
```

All paths (part 1, problem description and solution

Problem: given two distinct vertices of a graph, find all simple paths that begin at the first given vertex (the departure vertex) and end at the second given vertex (the destination vertex).

Solution: Do a depth-first traversal starting at the departure vertex, using the vertex marks to keep track of the path being built. In particular,

- the depth-first traversal can be done using recursion (that is not possible in a breadth-first traversal),
- the depth-first traversal should backtrack when the destination vertex is reached,
- initially mark all vertices with, say, -1 (an invalid vertex number),
- do not follow an edge if it arrives at an already marked vertex, and
- when following an edge, mark the vertex it departs from with the number of the vertex it arrives at (do not forget to unmark it when backtracking!).

Marking with vertex numbers is quite useful because these marks can be used to follow each complete path from its departing vertex to its destination vertex (a linked list!).

Task: For our test graph, enumerate all paths starting at vertex 1 and ending at vertex 9. Identify the paths with the smallest and largest weights (the weight of a path is the sum of the weights of its edges).

Curiosity: In a complete undirected graph with n vertices there are $f(n)$ paths between any two vertices, where $f(1) = 0$, $f(2) = 1$, and, for $n > 2$, $f(n) = (n-2)f(n-1) + 1$. Note that $f(n) \approx e \cdot (n-2)!$

n	$f(n)$
1	0
2	1
3	2
4	5

n	$f(n)$
5	16
6	65
7	326
8	1957

n	$f(n)$
9	13700
10	109601
11	986410
12	9864101

All paths (part 2, code)

The following C functions implement one possible solution to the all paths problem posed in the previous slide.

```
void all_paths_r(graph *g,int initial_vertex,int final_vertex,int current_vertex,int current_weight)
{
  int i;

  if(current_vertex == final_vertex)
  { // found one!
    printf("  %d",(i = initial_vertex) + 1);
    do
      printf(" -> %d",(i = g->vertices[i].mark) + 1);
    while(i != final_vertex);
    printf(" [%d]\n",current_weight);
  }
  else
    for(edge *e = g->vertices[current_vertex].out_edges;e != NULL;e = e->next)
      if(g->vertices[e->vertex_number].mark < 0)
      {
        g->vertices[current_vertex].mark = e->vertex_number;
        all_paths_r(g,initial_vertex,final_vertex,e->vertex_number,current_weight + e->weight);
        g->vertices[current_vertex].mark = -1;
      }
}

void all_paths(graph *g,int initial_vertex,int final_vertex)
{
  assert(initial_vertex != final_vertex);
  printf("all paths between vertices %d and %d\n",initial_vertex + 1,final_vertex + 1);
  mark_all_vertices(g,-1);
  all_paths_r(g,initial_vertex,final_vertex,initial_vertex,0);
}
```

All paths (part 3, example)

There are 33 paths starting at vertex 1 and ending at vertex 9:

$1 \to 4 \to 8 \to 9$ (weight 9)
$1 \to 4 \to 7 \to 9$ (weight 11)
$1 \to 4 \to 7 \to 6 \to 9$ (weight 12)
$1 \to 4 \to 7 \to 6 \to 5 \to 9$ (weight 15)
$1 \to 4 \to 7 \to 6 \to 3 \to 5 \to 9$ (weight 18, **largest**)
$1 \to 4 \to 7 \to 6 \to 3 \to 2 \to 5 \to 9$ (weight 18, **largest**)
$1 \to 3 \to 6 \to 9$ (weight 8)
$1 \to 3 \to 6 \to 7 \to 9$ (weight 11)
$1 \to 3 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 13)
$1 \to 3 \to 6 \to 5 \to 9$ (weight 11)
$1 \to 3 \to 5 \to 9$ (weight 10)
$1 \to 3 \to 5 \to 6 \to 9$ (weight 9)
$1 \to 3 \to 5 \to 6 \to 7 \to 9$ (weight 12)
$1 \to 3 \to 5 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 14)
$1 \to 3 \to 2 \to 5 \to 9$ (weight 10)
$1 \to 3 \to 2 \to 5 \to 6 \to 9$ (weight 9)
$1 \to 3 \to 2 \to 5 \to 6 \to 7 \to 9$ (weight 12)
$1 \to 3 \to 2 \to 5 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 14)
$1 \to 2 \to 5 \to 9$ (weight 7)
$1 \to 2 \to 5 \to 6 \to 9$ (weight 6, **smallest**)
$1 \to 2 \to 5 \to 6 \to 7 \to 9$ (weight 9)
$1 \to 2 \to 5 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 11)
$1 \to 2 \to 5 \to 3 \to 6 \to 9$ (weight 9)
$1 \to 2 \to 5 \to 3 \to 6 \to 7 \to 9$ (weight 12)
$1 \to 2 \to 5 \to 3 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 14)
$1 \to 2 \to 3 \to 6 \to 9$ (weight 7)
$1 \to 2 \to 3 \to 6 \to 7 \to 9$ (weight 10)
$1 \to 2 \to 3 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 12)
$1 \to 2 \to 3 \to 6 \to 5 \to 9$ (weight 10)
$1 \to 2 \to 3 \to 5 \to 9$ (weight 9)
$1 \to 2 \to 3 \to 5 \to 6 \to 9$ (weight 8)
$1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 9$ (weight 11)
$1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 13)

The path with the smallest weight is, in this case, not one of the paths with the smallest number of edges.

All cycles

Problem: Find all simple cycles (closed paths) of a graph.

Solution: Do a depth-first traversal starting at each vertex, never move to a vertex with a number (index) smaller than that of the starting vertex, and using the vertex marks to keep track of the cycle being built. In particular,

- the depth-first traversal can be done using recursion (that is not possible in a breadth-first traversal),
- the depth-first traversal should backtrack when a cycle is found,
- initially mark all vertices with, say, -1 (an invalid vertex number),
- do not follow an edge if it arrives at an already marked vertex or if that vertex has a number (index) smaller than that of the starting vertex , and
- when following an edge, mark the vertex it departs from with the number of the vertex it arrives at (do not forget to unmark it when backtracking!).

Task: Enumerate all cycles of our test graph. Identify the cycles with the smallest and largest weights (the weight of a cycle is the sum of the weights of its edges).

Curiosity: In a complete undirected graph with n vertices there are $g(n)$ cycles, where $g(1) = g(2) = 0$, $g(3) = 1$, and, for $n > 3$, $g(n) = \frac{(n-1)(n-2)}{2} + ng(n-1) - (n-1)g(n-2)$. Note that

$$g(n) = \frac{n!}{2}\sum_{k=3}^{n}\frac{1}{k(n-k)!} \approx \frac{e}{2}\frac{(n+1)!}{n^2}.$$

n	$f(n)$	n	$f(n)$	n	$f(n)$
1	0	5	37	9	62814
2	0	6	197	10	556014
3	1	7	1172	11	5488059
4	7	8	8018	12	59740609

All cycles (part 2, code)

The following C functions implement one possible solution to the all cycles problem posed in the previous slide.

```c
void all_cycles_r(graph *g,int initial_vertex,int current_vertex,int current_weight)
{
  int i;

  for(edge *e = g->vertices[current_vertex].out_edges;e != NULL;e = e->next)
    if(e->vertex_number == initial_vertex)
    { // found one!
      if(g->vertices[initial_vertex].mark < current_vertex)
      { // make sure each cycle is reported only once (and ignore cycles with only two edges)
        printf("  %d",(i = initial_vertex) + 1);
        do
          printf(" -> %d",(i = g->vertices[i].mark) + 1);
        while(i != current_vertex);
        printf(" -> %d [%d]\n",initial_vertex + 1,current_weight + e->weight);
      }
    }
    else if(e->vertex_number > initial_vertex && g->vertices[e->vertex_number].mark < 0)
    {
      g->vertices[current_vertex].mark = e->vertex_number;
      all_cycles_r(g,initial_vertex,e->vertex_number,current_weight + e->weight);
      g->vertices[current_vertex].mark = -1;
    }
}

void all_cycles(graph *g)
{
  printf("all cycles\n");
  mark_all_vertices(g,-1);
  for(int i = 0;i < g->n_vertices;i++)
    all_cycles_r(g,i,i,0);
}
```

All cycles (part 3, example)

$1 \to 3 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 17)
$1 \to 3 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 19)
$1 \to 3 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 20)
$1 \to 3 \to 6 \to 7 \to 4 \to 1$ (weight 16)
$1 \to 3 \to 6 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 20)
$1 \to 3 \to 6 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 22, **largest**)
$1 \to 3 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 19)
$1 \to 3 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 21)
$1 \to 3 \to 5 \to 9 \to 6 \to 7 \to 4 \to 1$ (weight 22, **largest**)
$1 \to 3 \to 5 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 3 \to 5 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 20)
$1 \to 3 \to 5 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 21)
$1 \to 3 \to 5 \to 6 \to 7 \to 4 \to 1$ (weight 17)
$1 \to 3 \to 2 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 19)
$1 \to 3 \to 2 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 21)
$1 \to 3 \to 2 \to 5 \to 9 \to 6 \to 7 \to 4 \to 1$ (weight 22, **largest**)
$1 \to 3 \to 2 \to 5 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 3 \to 2 \to 5 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 20)
$1 \to 3 \to 2 \to 5 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 21)
$1 \to 3 \to 2 \to 5 \to 6 \to 7 \to 4 \to 1$ (weight 17)
$1 \to 2 \to 5 \to 9 \to 8 \to 4 \to 7 \to 6 \to 3 \to 1$ (weight 20)
$1 \to 2 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 16)
$1 \to 2 \to 5 \to 9 \to 7 \to 6 \to 3 \to 1$ (weight 18)
$1 \to 2 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 5 \to 9 \to 6 \to 7 \to 4 \to 1$ (weight 19)
$1 \to 2 \to 5 \to 9 \to 6 \to 3 \to 1$ (weight 15)
$1 \to 2 \to 5 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 15)
$1 \to 2 \to 5 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 17)
$1 \to 2 \to 5 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 5 \to 6 \to 7 \to 4 \to 1$ (weight 14)
$1 \to 2 \to 5 \to 6 \to 3 \to 1$ (weight 10)
$1 \to 2 \to 5 \to 3 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 5 \to 3 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 20)
$1 \to 2 \to 5 \to 3 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 21)
$1 \to 2 \to 5 \to 3 \to 6 \to 7 \to 4 \to 1$ (weight 17)
$1 \to 2 \to 5 \to 3 \to 1$ (weight 9)
$1 \to 2 \to 3 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 16)
$1 \to 2 \to 3 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 3 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 19)
$1 \to 2 \to 3 \to 6 \to 7 \to 4 \to 1$ (weight 15)
$1 \to 2 \to 3 \to 6 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 19)
$1 \to 2 \to 3 \to 6 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 21)
$1 \to 2 \to 3 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 3 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 20)
$1 \to 2 \to 3 \to 5 \to 9 \to 6 \to 7 \to 4 \to 1$ (weight 21)
$1 \to 2 \to 3 \to 5 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 17)
$1 \to 2 \to 3 \to 5 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 19)
$1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 20)
$1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 4 \to 1$ (weight 16)
$1 \to 2 \to 3 \to 1$ (weight 7)
$2 \to 3 \to 6 \to 9 \to 5 \to 2$ (weight 10)
$2 \to 3 \to 6 \to 7 \to 9 \to 5 \to 2$ (weight 13)
$2 \to 3 \to 6 \to 7 \to 4 \to 8 \to 9 \to 5 \to 2$ (weight 15)
$2 \to 3 \to 6 \to 5 \to 2$ (weight 5)
$2 \to 3 \to 5 \to 2$ (weight 4, **smallest**)
$3 \to 5 \to 9 \to 8 \to 4 \to 7 \to 6 \to 3$ (weight 15)
$3 \to 5 \to 9 \to 7 \to 6 \to 3$ (weight 13)
$3 \to 5 \to 9 \to 6 \to 3$ (weight 10)
$3 \to 5 \to 6 \to 3$ (weight 5)
$4 \to 7 \to 9 \to 8 \to 4$ (weight 8)
$4 \to 7 \to 6 \to 9 \to 8 \to 4$ (weight 9)
$4 \to 7 \to 6 \to 5 \to 9 \to 8 \to 4$ (weight 12)
$5 \to 6 \to 9 \to 5$ (weight 7)
$5 \to 6 \to 7 \to 9 \to 5$ (weight 10)
$6 \to 7 \to 9 \to 6$ (weight 7)
[There are 65 cycles.]

Shortest path (part 1, problem formulation)

Problem: Given a weighted directed graph, find the path with the smallest weight between a given source vertex s and a given destination vertex d (connected to s). Assume that all weights are nonnegative.

Similar problem: Given a weighted directed graph and a source vertex s, find the paths with smallest weight to all other vertices of the graph (in the same connected component). Assume that all weights are nonnegative.

Related problem: Given a weighted directed graph, find the path with the smallest weight between a given source vertex and a given destination vertex (connected to s). Assume that the weights can take positive, zero, or negative values, and that all cycles have a positive weight.

Possible solution: Use brute force. Generate all possible paths between s and d and choose one with the smallest weight. Bad idea! The number of paths that may need to be checked can be very large.

Solution: Exploit the property that if a path of smallest weight passes through vertex v then its sub-path from s to v has the smallest possible weight of all paths starting at s and ending at v. (If that were not the case then that sub-path could be replaced by a better one.)

Let $W(s,v)$ be the weight of the best path that starts at s and ends at v. Our problem is to find $W(s,d)$ and to find a path with this weight. Let $w_{p,q}$ be the weight of the edge that connects p to q (if there is no edge $w_{p,q}$ is $+\infty$). We have

$$W(s,d) = \min_{v \in V}\bigl(W(s,v) + w_{v,d}\bigr).$$

To solve the problem it is thus necessary to solve smaller problems of the same type. As explained in the next slide, it is better to organize the computation so that larger and larger problems are solved (instead of smaller and smaller as this equation implies).

Shortest path (part 2, solution [Dijkstra's algorithm])

To implement the equation at the bottom of the previous slide it is necessary to find all edges that arrive at $\boldsymbol{d}$. That information is not stored in the graph data structures that we have been using in these slides (for each vertex only the edges that leave the vertex are stored). Although this can be solved easily, it remains the problem of how to organize the computation (in a dynamic programming style) so that cycles in the graph do not give rise to cycles in the algorithm.

It turns out that it is easier to construct the solution, one vertex at a time, starting at vertex $\boldsymbol{s}$. Suppose that $\boldsymbol{V'}$ is a subset of $\boldsymbol{V}$ that contains $\boldsymbol{s}$, that $\boldsymbol{W(s,p)}$ is already known for all $\boldsymbol{p \in V'}$, and that $\boldsymbol{V'}$ contains the $|\boldsymbol{V'}|$ vertices with the smallest values of $\boldsymbol{W(s,v)}$. Initially $\boldsymbol{V' = \{s\}}$ and $\boldsymbol{W(s,s) = 0}$. Each step of the algorithm selects an edge departing from a vertex $\boldsymbol{p \in V'}$ and arriving at a vertex $\boldsymbol{q \in V - V'}$ (in the neighborhood of $\boldsymbol{V'}$), so that $\boldsymbol{W(s,p) + w_{p,q}}$ is minimized. (Break ties arbitrarily.) This ensures, for the vertices $\boldsymbol{p}$ and $\boldsymbol{q}$ of the selected edge, that $\boldsymbol{W(s,q) = W(s,p) + w_{p,q}}$, because since edge weights are nonnegative any other path would not be better. If $\boldsymbol{q}$ is $\boldsymbol{d}$ we are done. Otherwise, add $\boldsymbol{q}$ to $\boldsymbol{V'}$ and repeat the procedure. Best paths can be reconstructed by recording the edges that were selected (store in vertex $\boldsymbol{q}$ the corresponding $\boldsymbol{p}$, so that paths can be traced back from $\boldsymbol{d}$ to $\boldsymbol{s}$).

Shortest path (part 3, implementation details)

For each vertex v keep

- the weight $Z(s,v)$ of the current best path from s to v — if $v \in V'$ then $W(s,v) = Z(s,v)$,
- the number of the previous vertex, $P(v)$, of the current best path ending at the vertex.

Initially, set $Z(s,s) = 0$ and set $Z(s,v) = \infty$ for the other vertices. When a vertex p is added to V' (the first one will be s itself), for all vertices q that p is connected to check if $Z(s,q)$ is larger than $Z(s,p) + w_{p,q}$. If it is, set $Z(s,q)$ to $Z(s,p) + w_{p,q}$ and set $P(q) = p$.

The next vertex to be added to V' will be the one, not already there, with the smallest value of $W(s,v)$. To do this efficiently, the values of $W(s,v)$ can be kept in a min-heap. This is not entirely trivial to do, because it is necessary to keep track of the position where each $Z(s,v)$ value is stored in the heap. This is necessary because it may be necessary to decrease its value. It is thus also necessary, for each vertex v, to keep

- the heap position (an index) where $Z(s,v)$ is stored.

It can be shown that this algorithm takes $O\bigl(|E| \log |V|\bigr)$ time to finish.

Shortest path (part 4a, code – data structures)

To simplify the code the min heap data can be placed inside the data structures used to represent a graph (the vertex `mark` field is not needed here).

```
typedef struct edge   edge;
typedef struct vertex vertex;
typedef struct graph  graph;

struct edge
{
  edge *next;           // next edge node
  int vertex_number;    // vertex number
  int weight;           // edge weight
};

struct vertex
{
  edge *out_edges;      // adjacency list head

  vertex *trace_back; // "follow-path-back" vertex
  int weight;           // shortest path weight
  int heap_index;       // min heap index of this vertex

};

struct graph
{
  vertex *vertices;     // the vertices
  int n_vertices;       // number of vertices

  vertex **heap;        // heap of vertex pointers
  int heap_size;        // current heap size

};
```

Shortest path (part 4b, code – update vertex data)

The following function updates $Z(s,q)$ given p and $Z(s,p)+w_{p,q}$. The function argument from is a pointer to the vertex p, the argument to is a pointer to q, and the argument weight is $Z(s,p)+w_{p,q}$. The function also keeps vertices in their proper places in the min-heap.

```c
static void update_weight(graph *g,vertex *from,vertex *to,int weight)
{
  int i;

  if(to->trace_back != NULL && to->weight <= weight)
    return; // not better
  to->weight = weight;
  to->trace_back = from;
  if(to->heap_index > 0)
    i = to->heap_index; // already in the heap
  else
    i = ++g->heap_size; // augment heap
  for(;i > 1 && g->heap[i / 2]->weight > weight;i /= 2)
    (g->heap[i] = g->heap[i / 2])->heap_index = i;
  (g->heap[i] = to)->heap_index = i;
}
```

The trace_back fields are initialized to NULL, so that is used to deal correctly with the first update of the to data. The heap_index fields are also initialized to 0, so that is used to check if to is already in the min-heap (if so, it says where it is).

Shortest path (part 4c, code – get vertex with the smallest path weight)

The following function used the min-heap data to retrieve the vertex with the smallest path weight. It returns `NULL` when the min-heap is empty.

```
static vertex *get_vertex_with_min_weight(graph *g)
{
  vertex *v,*t;
  int i,j;

  if(g->heap_size == 0)
    return NULL;
  v = g->heap[1];
  t = g->heap[g->heap_size--];
  for(i = 1;2 * i <= g->heap_size;i = j)
  {
    j = (2 * i + 1 <= g->heap_size && g->heap[2 * i + 1]->weight < g->heap[2 * i]->weight) ? 2 * i + 1 : 2 * i;
    (g->heap[i] = g->heap[j])->heap_index = i;
  }
  (g->heap[i] = t)->heap_index = i;
  v->heap_index = 0; // do this last bacause t = v when the heap becomes empty
  return v;
}
```

Shortest path (part 4d, code – the algorithm)

Finally, the Dijkstra algorithm. First, the vertex data and the min-heap are initialized. Then, vertices are extracted from the min-heap one at a time, all their outgoing edges are followed, that the information of each destination vertex is updated. The function computes the path of smallest weight from the starting vertex to all other vertices. It would be trivial to modify the code to stop the algorithm when the best path to a given destination vertex becomes known.

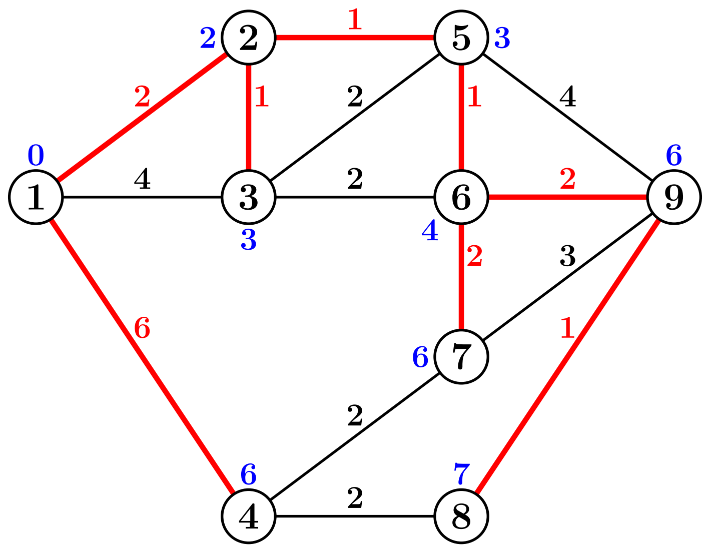
```
static void shortest_paths(graph *g,int starting_vertex)
{
  vertex *v;
  edge *e;
  int i;

  assert(starting_vertex >= 0 && starting_vertex < g->n_vertices);
  for(i = 0;i < g->n_vertices;i++)
  {
    g->vertices[i].weight = 0;
    g->vertices[i].trace_back = NULL;
    g->vertices[i].heap_index = 0;
  }
  update_weight(g,&g->vertices[starting_vertex],&g->vertices[starting_vertex],0);
  while((v = get_vertex_with_min_weight(g)) != NULL)
    for(e = v->out_edges;e != NULL;e = e->next)
      update_weight(g,v,&g->vertices[e->vertex_number],v->weight + e->weight);
}
```

The entire code can be found in the `P13.tgz` archive.

Shortest path (part 5, example)

The following figure presents the best paths from vertex 1 to all other vertices. The smallest weight of each path is shown in blue near the destination vertex. Edges belonging to at least one best path are shown in red. Note that they form a tree.



The progress of the algorithm is displayed on the table on the right. Each stage of the algorithm has two phases. In the first, <u>one</u> of the vertices with the smallest distance that has not yet been selected (not blue) is selected; this is signaled by a small circle placed between columns. In the second, the distances of the paths that pass through the selected vertex are updated (best current distances are presented in black, discarded distances are presented in gray). The algorithm terminates either when there are no more vertices to select or when the destination vertex has been selected.

1	0 ●	0	0	0	0	0	0	0	0	0
2	∞	∞,2 ●	2	2	2	2	2	2	2	2
3	∞	∞,4	4,3 ●	3	3	3	3	3	3	3
4	∞	∞,6	6	6	6	6	6,8 ●	6	6	6
5	∞	∞	∞,3	3,5 ●	3	3	3	3	3	3
6	∞	∞	∞	∞,5	5,4 ●	4	4	4	4	4
7	∞	∞	∞	∞	∞	∞,6 ●	6	6	6	6
8	∞	∞	∞	∞	∞	∞	∞	∞,8	8,7 ●	7
9	∞	∞	∞	∞	∞,7	7,6	6,9	6 ●	6	6

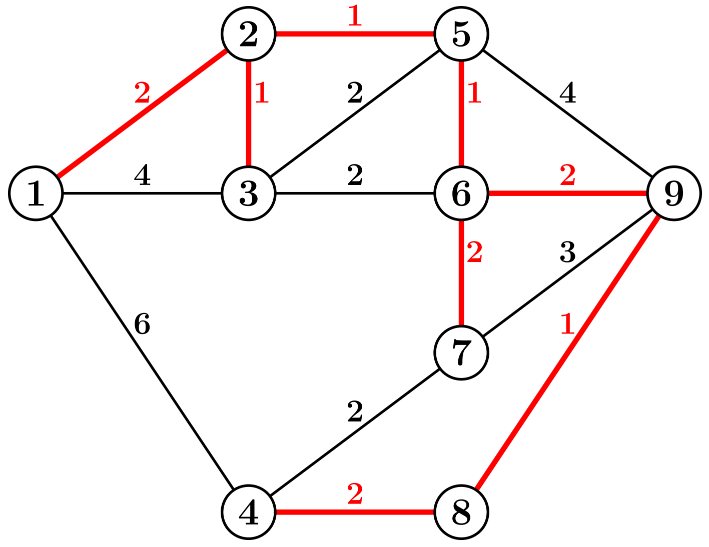
Minimum spanning tree (part 1, Kruskal's algorithm and example)

Problem: Given a weighted undirected connected graph, find the spanning tree with the smallest weight. (The weight of a tree is the sum of the weights of its edges.)

Solution [Kruskal]: Sort the edges in increasing order. While the number of accepted edges is smaller than the number of vertices minus one, get the next edge (start with the smallest). If its two endpoints belong to different connected components, accept the edge (and update the connected components information using the union-find data structure). Otherwise reject the edge.

It can be shown that this algorithm takes $O\big(|E| \log |E|\big)$ time to finish.

When the edge weights are distinct, the minimum spanning tree is unique. Otherwise, many spanning trees with the same weight can exist.



Minimum spanning tree (part 2, Prim's algorithm)

Solution [Prim]: Let V' be a set of vertices having initially only one vertex (chosen arbitrarily). Let E' be the set of edges that connect one vertex of V' to a vertex in the neighborhood of V'. Repeat the following procedure until E' becomes empty. Select the edge of E' with the smallest weight. That edge becomes part of the minimum spanning tree. Add its arriving vertex to V'.

It can be shown that this algorithm takes $O\big(|E| \log |V|\big)$ time to finish when the edge selection is done using a priority queue (min-heap).