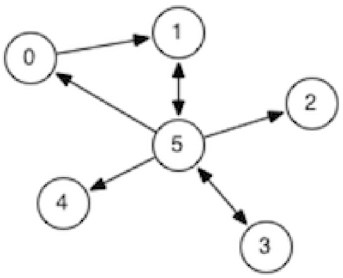
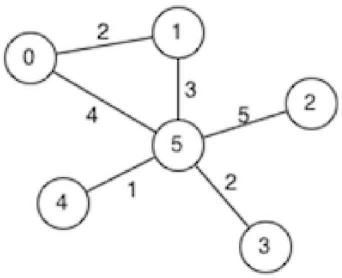
Week 08 Problem Set

Minimum Spanning Trees, Shortest Paths, Maximum Flows

1. (Graph representations)

   For each of the following graphs:

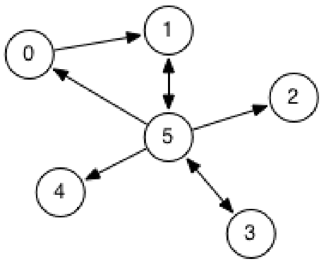




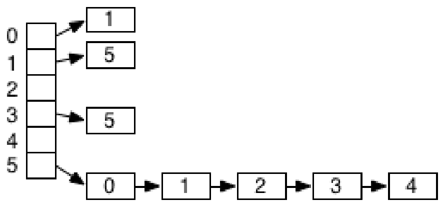
   Show the concrete data structures if the graph was implemented via:

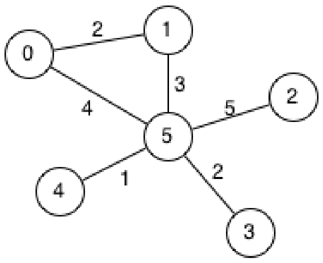
   a. adjacency matrix representation (assume full V×V matrix)
   b. adjacency list representation (if non-directional, include both (v,w) and (w,v))

   Answer:

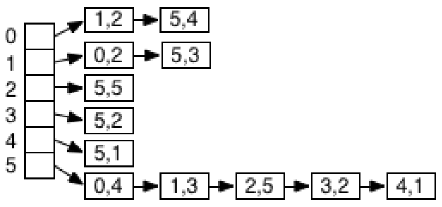


	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	0	0
3	0	0	0	0	0	1
4	0	0	0	0	0	0
5	1	1	1	1	1	0



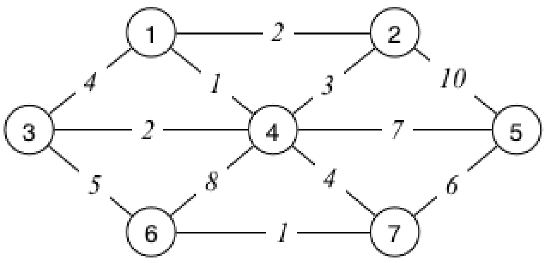


	0	1	2	3	4	5
0	0	2	0	0	0	4
1	2	0	0	0	0	3
2	0	0	0	0	0	5
3	0	0	0	0	0	2
4	0	0	0	0	0	1
5	4	3	5	2	1	0



2. (MST)

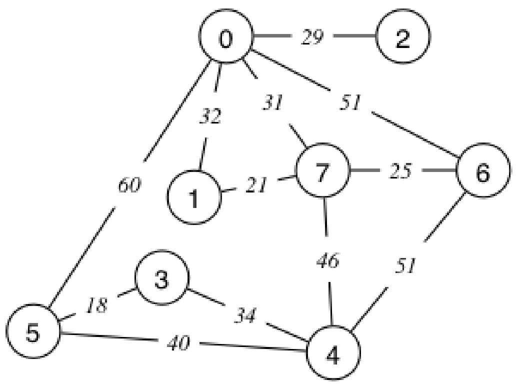
   a. Show how Kruskal's algorithm would construct the MST for the following graph:



   How many edges do you have to consider?

   b. For a graph *G=(V,E)*, what is the least number of edges that might need to be considered by Kruskal's algorithm, and what is the most number of edges? Add one vertex and edge to the above graph to force Kruskal's algorithm to the worst case.

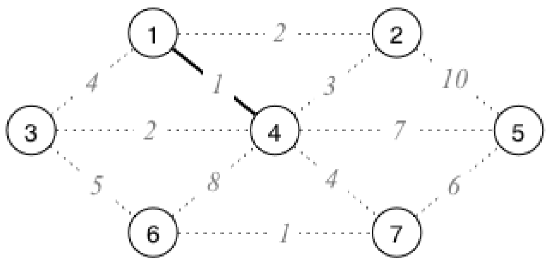
   c. Trace the execution of Prim's algorithm to compute a minimum spanning tree on the following graph:



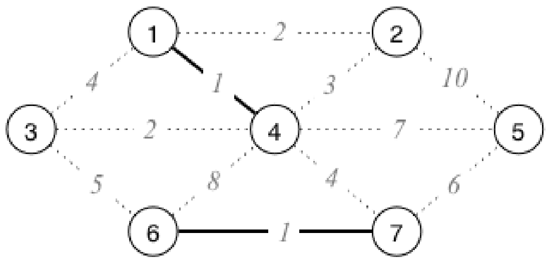
Choose a random vertex to start with. Draw the resulting minimum spanning tree.

Answer:

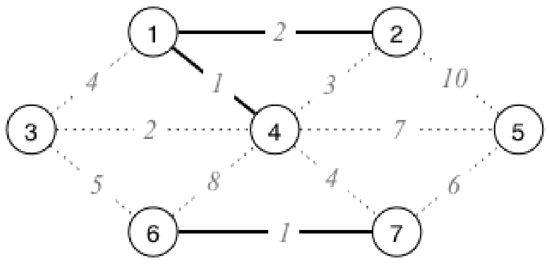
a. In the first iteration of Kruskal's algorithm, we could choose either 1-4 or 6-7, since both edges have weight 1. Assume we choose 1-4. Since its inclusion produces no cycles, we add it to the MST (non-existent edges are indicated by dotted lines):



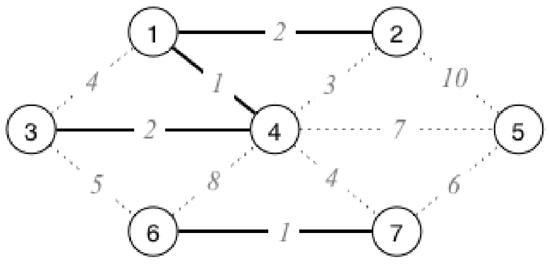
In the next iteration, we choose 6-7. Its inclusion produces no cycles, so we add it to the MST:



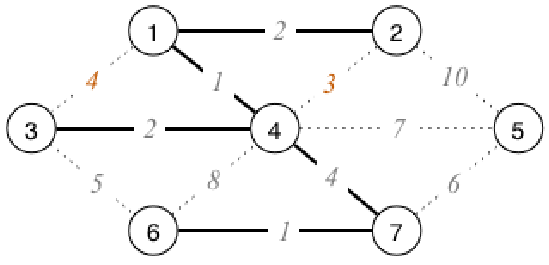
In the next iteration, we could choose either 1-2 or 3-4, since both edges have weight 2. Assume we choose 1-2. Since its inclusion produces no cycles, we add it to the MST:



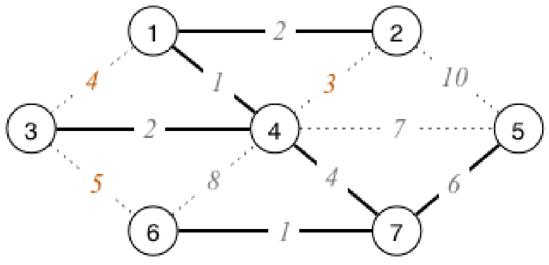
In the next iteration, we choose 3-4. Its inclusion produces no cycles, so we add it to the MST:



In the next iteration, we would first consider the lowest-cost unused edge. This is 2-4, but its inclusion would produce a cycle, so we ignore it. We then consider 1-3 and 4-7 which both have weight 4. If we choose 1-3, that produces a cycle so we ignore that edge. If we add 4-7 to the MST, there is no cycle and so we include it:



Now the lowest-cost unused edge is 3-6, but its inclusion would produce a cycle, so we ignore it. We then consider 5-7. If we add 5-7 to the MST, there is no cycle and so we include it:



At this stage, all vertices are connected and we have a MST.

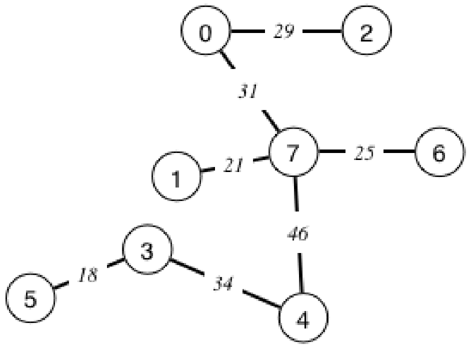
For this graph, we considered 9 of the 12 possible edges in determining the MST.

b. For a graph with *V* vertices and *E* edges, the best case would be when the first *V-1* edges we consider are the lowest cost edges and none of these edges leads to a cycle. The worst case would be when we had to consider all *E* edges. If we added a vertex 8 to the above graph, and connected it to vertex 5 with edge cost 11 (or any cost larger than all the other edge costs in the graph), we would need to consider all edges to construct the MST.

c. If we start at node 5, for example, edges would be found in the following order:

- 5-3
- 3-4
- 4-7
- 1-7
- 6-7
- 0-7
- 0-2

The minimum spanning tree:



3. (Priority queue for Dijkstra's algorithm)

Assume that a priority queue for Dijkstra's algorithm is represented by a global variable `PQueue` of the following structure:

```
#define MAX_NODES 1000
typedef struct {
   Vertex item[MAX_NODES];  // array of vertices currently in queue
   int    length;           // #values currently stored in item[] array
} PQueueT;

PQueueT PQueue;
```

Further assume that vertices are stored in the `item[]` array in the priority queue in *no* particular order. Rather, the `item[]` array acts like a set, and the priority is determined by reference to a `priority[0..nV-1]` array.

If the priority queue `PQueue` is initialised as follows:

```
void PQueueInit() {
   PQueue.length = 0;
}
```

then give the implementation of the following functions in C:

```
// insert vertex v into priority queue PQueue
// no effect if v is already in the queue
void joinPQueue(Vertex v) { ... }

// remove the highest priority vertex from PQueue
// remember: highest priority = lowest value priority[v]
// returns the removed vertex
Vertex leavePQueue(int priority[]) { ... }

// check if priority queue PQueue is empty
bool PQueueIsEmpty() { ... }
```

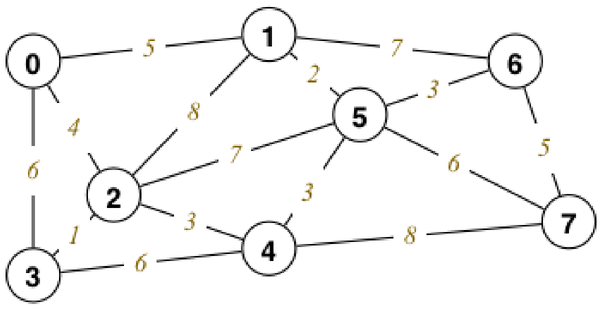
Answer:

The following are possible implementations for joining and leaving the priority queue, and checking whether it's empty:

```
void joinPQueue(Vertex v) {
   assert(PQueue.length < MAX_NODES);
   int i = 0;
   while (i < PQueue.length && PQueue.item[i] != v)  // check if v already in queue
      i++;
   if (i == PQueue.length) {                         // v not found => add it at the end
      PQueue.item[PQueue.length] = v;
      PQueue.length++;
   }
}

#define VERY_HIGH_VALUE 999999

Vertex leavePQueue(int priority[]) {
   assert(PQueue.length > 0);

   int i, bestIndex = 0, bestVertex = PQueue.item[0], bestWeight = VERY_HIGH_VALUE;
   for (i = 0; i < PQueue.length; i++) {             // find i with min priority[item[i]]
      if (priority[PQueue.item[i]] < bestWeight) {
         bestIndex = i;
         bestWeight = priority[PQueue.item[i]];
         bestVertex = PQueue.item[i];                // vertex with lowest value so far
      }
   }
   PQueue.length--;
   PQueue.item[bestIndex] = PQueue.item[PQueue.length];  // replace dequeued node
                                                         // by last element in array
   return bestVertex;
}

bool PQueueIsEmpty() {
   return (PQueue.length == 0);
}
```

4. (Dijkstra's algorithm)

   a. Trace the execution of Dijkstra's algorithm on the following graph to compute the minimum distances from source node 0 to all other vertices:



   Show the values of `vSet`, `dist[]` and `pred[]` after each iteration.

   b. Implement Dijkstra's algorithm in C using your priority queue functions for Exercise 3 and the Weighted Graph ADT (`WGraph.h`, `WGraph.c`) from the lecture. Your program should

   - prompt the user for the number of nodes in a graph,
   - prompt the user for a source node,
   - build a weighted undirected graph from user input,

- compute and output the distance and a shortest path from the source to every vertex of the graph.

An example of the program executing is shown below. The input graph consists of a simple triangle along with a fourth, disconnected node.

```
prompt$ ./dijkstra
Enter the number of vertices: 4
Enter the source node: 0
Enter an edge (from): 0
Enter an edge (to): 1
Enter the weight: 42
Enter an edge (from): 0
Enter an edge (to): 2
Enter the weight: 25
Enter an edge (from): 1
Enter an edge (to): 2
Enter the weight: 14
Enter an edge (from): done
Finished.
0: distance = 0, shortest path: 0
1: distance = 39, shortest path: 0-2-1
2: distance = 25, shortest path: 0-2
3: no path
```

Note that any non-numeric data can be used to 'finish' the interaction. Test your algorithm with the graph from Exercise 4a.

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to the problem set and week. It expects to find a program named* `dijkstra.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ ~cs9024/bin/dryrun prob08
```

Answer:

a. Initialisation:

```
vSet = { 0, 1, 2, 3, 4, 5, 6, 7 }
dist = [ 0, ∞, ∞, ∞, ∞, ∞, ∞, ∞ ]
pred = [ -1, -1, -1, -1, -1, -1, -1, -1 ]
```

The vertex in vSet with minimum dist[] is 0. Relaxation along the edges (0,1,5), (0,2,4) and (0,3,6) results in:

```
vSet = { 1, 2, 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 6, ∞, ∞, ∞, ∞ ]
pred = [ -1, 0, 0, 0, -1, -1, -1, -1 ]
```

Now the vertex in vSet with minimum dist[] is 2. Considering all edges from 2 to nodes still in vSet:

- relaxation along (2,1,8) does not give us a shorter distance to node 1
- relaxation along (2,3,1) yields a smaller value (4+1 = 5) for dist[3], and pred[3] is updated to 2
- relaxation along (2,4,3) yields a smaller value (4+3 = 7) for dist[4], and pred[4] is updated to 2
- relaxation along (2,5,7) yields a smaller value (4+7 = 11) for dist[5], and pred[5] is updated to 2

```
vSet = { 1, 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 11, ∞, ∞ ]
pred = [ -1, 0, 0, 2, 2, 2, -1, -1 ]
```

Next, we could choose either 1 or 3, since both vertices have minimum distance 5. Suppose we choose 1. Relaxation along (1,5,2) and (1,6,7) results in new values for nodes 5 and 6:

```
vSet = { 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, ∞ ]
pred = [ -1, 0, 0, 2, 2, 1, 1, -1 ]
```

Now we consider vertex 3. The only adjacent node still in vSet is 4, but there is no shorter path to 4 through 3. Hence no update to dist[] or pred[]:

```
vSet = { 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, ∞ ]
pred = [ -1, 0, 0, 2, 2, 1, 1, -1 ]
```

Next we could choose either vertex 4 or 5. Suppose we choose 4. Edge (4,7,8) is the only one that leads to an update:

```
vSet = { 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, 15 ]
pred = [ -1, 0, 0, 2, 2, 1, 1, 4 ]
```

Vertex 5 is next. Relaxation along edges (5,6,3) and (5,7,6) results in:

```
vSet = { 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 10, 13 ]
pred = [ -1, 0, 0, 2, 2, 1, 5, 5 ]
```

Of the two vertices left in vSet, 6 has the shorter distance. Edge (6,7,5) does not update the values for node 7 since dist[7]=13<dist[6]+5=15. Hence:

```
vSet = { 7 }
dist = [ 0, 5, 4, 5, 7, 7, 10, 13 ]
pred = [ -1, 0, 0, 2, 2, 1, 5, 5 ]
```

Processing the last remaining vertex in vSet will obviously not change anything. The values in pred[] determine shortest paths to all nodes as follows:

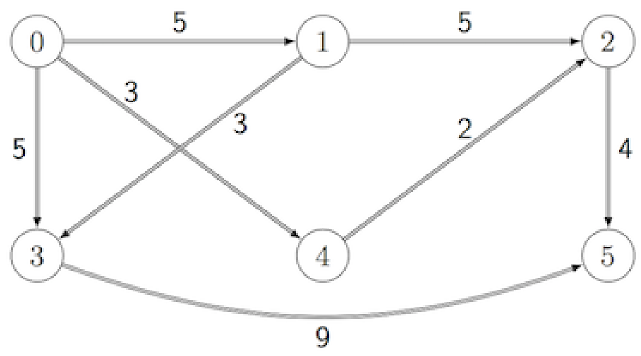
```
0: distance = 0, shortest path: 0
1: distance = 5, shortest path: 0-1
2: distance = 4, shortest path: 0-2
3: distance = 5, shortest path: 0-2-3
4: distance = 7, shortest path: 0-2-4
5: distance = 7, shortest path: 0-1-5
6: distance = 10, shortest path: 0-1-5-6
7: distance = 13, shortest path: 0-1-5-7
```

b. Sample implementation of Dijkstra's algorithm, including a recursive function to display a path via `pred[]`:

```
void showPath(int v, int pred[]) {
   if (pred[v] == -1) {
      printf("%d", v);
   } else {
      showPath(pred[v], pred);
      printf("-%d", v);
   }
}

void dijkstraSSSP(Graph g, Vertex source) {
   int  dist[MAX_NODES];
   int  pred[MAX_NODES];
   bool vSet[MAX_NODES];  // vSet[v] = true <=> v has not been processed
   int s, t;

   PQueueInit();
   int nV = numOfVertices(g);
   for (s = 0; s < nV; s++) {
      joinPQueue(s);
      dist[s] = VERY_HIGH_VALUE;
      pred[s] = -1;
      vSet[s] = true;
   }
   dist[source] = 0;
   while (!PQueueIsEmpty()) {
      s = leavePQueue(dist);
      vSet[s] = false;
      for (t = 0; t < nV; t++) {
         if (vSet[t]) {
            int weight = adjacent(g,s,t);
            if (weight > 0 && dist[s]+weight < dist[t]) {  // relax along (s,t,weight)
               dist[t] = dist[s] + weight;
               pred[t] = s;
            }
         }
      }
   }
   for (s = 0; s < nV; s++) {
      printf("%d: ", s);
      if (dist[s] < VERY_HIGH_VALUE) {
         printf("distance = %d, shortest path: ", dist[s]);
         showPath(s, pred);
         putchar('\n');
      } else {
         printf("no path\n");
      }
   }
}

void reverseEdge(Edge *e) {
   Vertex temp = e->v;
   e->v = e->w;
   e->w = temp;
}

int main(void) {
   Edge e;
   int n, source;

   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   Graph g = newGraph(n);

   printf("Enter the source node: ");
   scanf("%d", &source);
   printf("Enter an edge (from): ");
   while (scanf("%d", &e.v) == 1) {
      printf("Enter an edge (to): ");
      scanf("%d", &e.w);
      printf("Enter the weight: ");
      scanf("%d", &e.weight);
      insertEdge(g, e);
      reverseEdge(&e);              // ensure to add edge in both directions
      insertEdge(g, e);
      printf("Enter an edge (from): ");
   }
   printf("Finished.\n");

   dijkstraSSSP(g, source);
   freeGraph(g);
   return 0;
}
```

5. (Maximum Flow)

   a. Identify a maximum flow from source=0 to sink=5 in the following network (without applying the algorithm from the lecture):



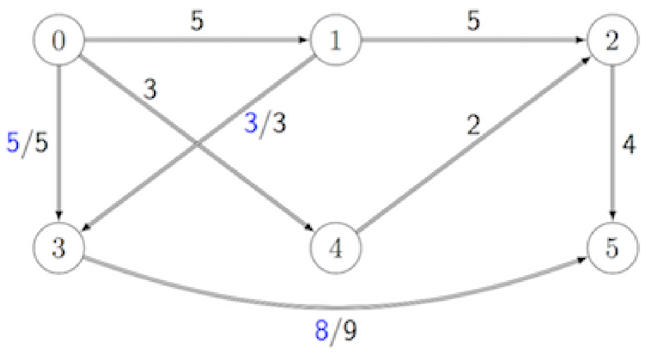
   What approach did you use in finding the maxflow?

   b. Show how Edmonds and Karp's algorithm would construct a maximum flow from 0 to 5 for the above network. How many augmenting paths do you have to consider?

Answer:

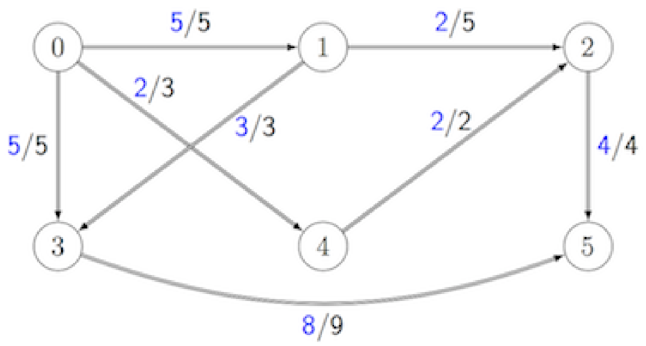
   a. I suspect (no proof) that most people would use a strategy like:

      1. Choose an edge from a vertex into the sink and attempt to see if and how the maximum flow through this edge can be achieved, for example edge (3,5) with capacity 9.
      2. Attempt to maximise the flow into this vertex 3 to check if the required capacity 9 can be reached.
      3. Realise that the maximum flow into 3 is actually 8, which would get you to a partial solution like:



      4. Inspect the other edge into the sink – (2,5) with capacity 4 – and attempt to max out its capacity.

5. Realise that the maximum flow that can be sent into node 2 is indeed 4, which results in the maximum flow shown below.



b. The shortest augmenting path found by BFS is 0-3-5, along which we can send `df=5`. The resulting `flow[][]` matrix is:

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	0	0	5	0	0
[1]	0	0	0	0	0	0
[2]	0	0	0	0	0	0
[3]	-5	0	0	0	0	5
[4]	0	0	0	0	0	0
[5]	0	0	0	-5	0	0

The negative values encode the "reversed" edges in the residual graph.

If we choose neighbours in ascending order, then the next augmenting path found by BFS is 0-1-2-5 with `df=4`. We update the `flow[][]` matrix to:

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	4	0	5	0	0
[1]	-4	0	4	0	0	0
[2]	0	-4	0	0	0	4
[3]	-5	0	0	0	0	5
[4]	0	0	0	0	0	0
[5]	0	0	-4	-5	0	0

Now the shortest augmenting path is 0-1-3-5 with `df=1`, so we update the `flow[][]` matrix as follows:

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	5	0	5	0	0
[1]	-5	0	4	1	0	0
[2]	0	-4	0	0	0	4
[3]	-5	-1	0	0	0	6
[4]	0	0	0	0	0	0
[5]	0	0	-4	-6	0	0

The residual graph admits one more augmenting path: 0-4-2-1-3-5 with `df=2`, by which some of the current flow from 1 to 2 is "redirected" to 3 and replaced by the same amount from 4 to 2. The resulting `flow[][]` matrix is:

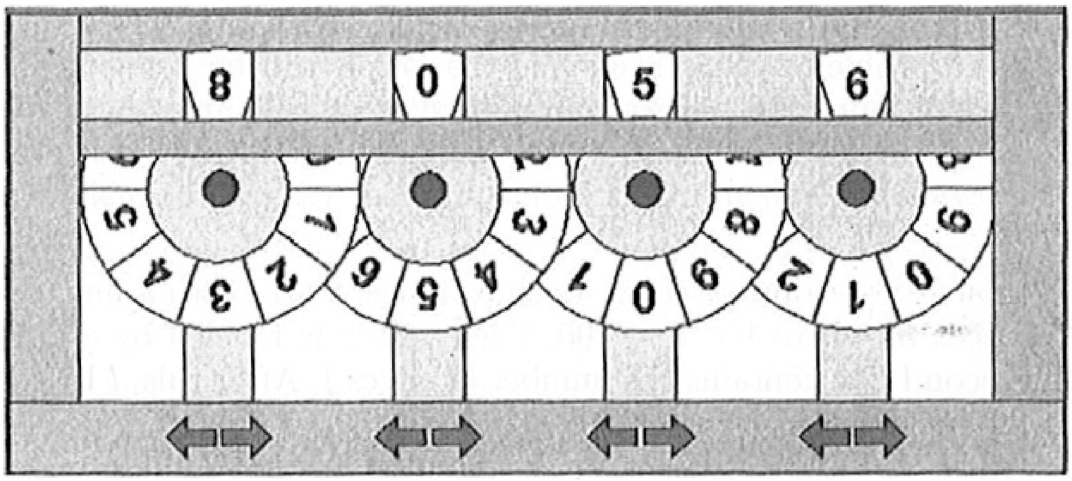
	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	5	0	5	2	0
[1]	-5	0	2	3	0	0
[2]	0	-2	0	0	-2	4

[3]	-5	-3	0	0	0	8
[4]	-2	0	2	0	0	0
[5]	0	0	-4	-8	0	0

No more augmenting path exists. The maximum flow is 5+4+1+2 = 12, the sum of what we could maximally send along the 4 augmenting paths that were found.

6. **Challenge Exercise**

Consider a machine with four wheels. Each wheel has the digits 0…9 printed clockwise on it. The current *state* of the machine is given by the four topmost digits `abcd`, e.g. 8056 in the picture below.



Each wheel can be controlled by two buttons: Pressing the button labelled with "←" turns the corresponding wheel clockwise one digit ahead, whereas presssing "→" turns it anticlockwise one digit back.

Write a C-program to determine the *minimum* number of button presses required to transform

- a given initial state, `abcd`
- to a given goal state, `efgh`
- without passing through any of *n≥0* given "forbidden" states, $\texttt{forbidden[]=\{w}_1\texttt{x}_1\texttt{y}_1\texttt{z}_1\texttt{,…,w}_n\texttt{x}_n\texttt{y}_n\texttt{z}_n\texttt{\}}$.

For example, the state 8056 as depicted can be transformed into 0056 in 2 steps if `forbidden[]={}`, whereas a minimum of 4 steps is needed for the same task if `forbidden[]={9056}`. (Why?)

Use your program to compute the least number of button presses required to transform 8056 to 7012 if

a. there are no forbidden states
b. you are not permitted to pass through any state 7055–8055 (i.e., 7055, 7056, …, 8055 all are forbidden)
c. you are not permitted to pass through any state 0000–0999 or 7055–8055

Answer:

The problem can be solved by a breadth-first search on an undirected graph:

- nodes 0…9999 correspond to the possible configurations 0000 – 9999 of the machine;
- edges connect nodes v and w if, and only if, one button press takes the machine from v to w and vice versa.

The following code implements BFS on this graph with the help of the integer queue ADT from the lecture (`queue.h`, `queue.c`). Note that the graph is built *implicitly* : nodes are generated as they are encountered during the search.

```
#include "queue.h"

int shortestPath(int source, int target, int forbidden[], int n) {
   int visited[10000];

   int i;
   for (i = 0; i <= 9999; i++)
      visited[i] = -1;             // mark all nodes as unvisited
   for (i = 0; i < n; i++)
      visited[forbidden[i]] = -2;  // mark forbidden nodes as visited => they won't be selected
   visited[source] = source;

   queue Q = newQueue();
   QueueEnqueue(Q, source);
   bool found = (target == source);
   while (!found && !QueueIsEmpty(Q)) {
      int v = QueueDequeue(Q);
      if (v == target) {
            found = true;
         } else {
            int wheel, turn;
            for (wheel = 10; wheel <= 10000; wheel *= 10) { // fancy way of generating the
               for (turn = 1; turn <= 9; turn += 8) {       // eight neighbour configurations of v
                  int w = wheel * (v / wheel) + (v % wheel + (wheel/10) * turn) % wheel;
                  if (visited[w] == -1) {
                     visited[w] = v;
                     QueueEnqueue(Q, w);
                  }
               }
            }
         }
      }
      dropQueue(Q);
      if (found) {                        // unwind path to determine length
         int length = 0;
         while (target != source) {
            target = visited[target];     // move to predecessor on path
            length++;
         }
         return length;
      } else {
         return -1;                       // no solution
      }
}
```

a. 9
b. 17
c. ∞ (unreachable)