Exercise sessions #1 and #2
Graph theory

prof. B. Demoen W. Van Onsem

March 2015

Exercise 1 (adjacency matrix)**.**

1. Construct the adjacency matrix of the following graph.
2. Use this matrix to determine whether there a path between a and d?
3. If so, what is the minimum number of edges in such path? How can you determine this using the adjacency matrix?
4. How many matrix-multiplication operations do you need to calculate the connectivity between every two nodes for a given graph $G(V, E)$? What if the number of edges is very small?
5. Is there a specific property for adjacency matrices for undirected graphs that is not necessary true of adjacency matrices for directed graphs?

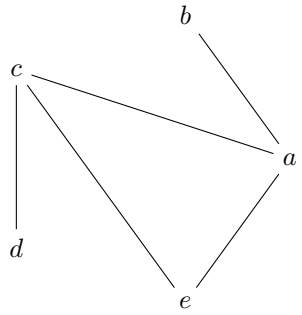

Figure 1: Graph for exercise 1.

Answer.

1. See matrix A. The indices of A correspond to the nodes in *alphabetical* order.

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix} \tag{1}$$

2. We can see that there is no direct edge between a and d because $A_{ad} = 0$, we thus determine whether there is a path of length 2 between a and d by calculating the square of A:

$$A^2 = \begin{pmatrix} 3 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 3 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 2 \end{pmatrix} \tag{2}$$

   As we can see $A^2_{ad} = 1$, there exists thus a path of length 2.

3. Since $A^0 = \mathbb{I}_5$ doesn't contain a path from a to d and neither does A, but A^2 does, we know that the shortest path uses two edges.

4. If we first perform a matrix-addition, $B = A + \mathbb{I}$. The resulting matrix B contains the edges as well as loops. If we now calculate B^2 we obtain all paths of length 0, 1 and 2. A naive algorithm can now perform n multiplications, thus calculate B^{n-1} with n the number of *nodes*. Since there are n nodes, it can take at most $n-1$ steps to go from one vertex to another one. A smarter approach hover is to reuse the result we obtain by B^2. Since $B^4 = \left(B^2\right)^2$, we can speed up the process. The algorithm does not calculate B^{n-1} using $n-2$ matrix multiplications, but $\log_2(n-1)$. If the number of edges is very small, we can make it even more efficient. The longest path is bounded by the number of edges e. If e is very small, we can calculate $B^{\min(n-1;e)}$. We can use our efficient algorithm for this.

◇

Exercise 2 (Euler's theorem)**.**

1. Generalize Euler's theorem with respect to *directed* graphs. Don't forget to take loops into account.

2. Prove that in a graph containing exactly two vertices u and v of odd degree, there exists a path from u to v.

Answer.

1. The new theorem states:

   Theorem 1 (Euler's modified theorem)**.** A directed graph $G(V, E)$ contains a Eulerian path if and only if the graph is connected[1] and for all vertices $v \in V$ is the number of incoming edges equal to the number of outgoing edges.

   Proof. Evidently the graph must be a connected graph: a Eulerian cycle contains each vertex so that means that we need to be able to traverse from every vertex to every vertex by following the Eulerian cycle.

   We construct a cycle P we do this by starting from a vertex v since the graph is connected, we know that there must be at least one outgoing edge. We now take one of the edges to a vertex w. Since w has at least one incoming edge, it has at least one outgoing edge as well, so we can repeat this operation until we reach v again. Now there are two possibilities:

   (a) We have visited every edge. In that case we've constructed a Eulerian cycle so we're done; or

[1] Note that the definition of connected differs a bit in the context of a directed graph.

(b) We haven't visited an edge. Optionally this means we haven't even visited every vertex. Since the graph is connected however, it means that there is at least one vertex we've visited where there exists an edge we haven't visited in our original path. Now since our original path always took one incoming and one outgoing edge per cycle, we know that this vertex must also contain an unused incoming edge. If we take the outgoing vertex, we're sure we can find a way to return to the vertex since the graph is connected. We can thus construct a cycle from that vertex that we can add to the path.

By keeping to add cycles to the path, we will eventually end up with a Eulerian cycle because we can't add a cycle to the path anymore of no edges haven't been visited anymore. □

2. *Proof.* We first define two new functions: $l_G(v)$ the number of loops of a vertex v and $x_G(v)$ the number of edges to another vertex of v. By definition the degree of a vertex is equal to $d_G(v) = 2 \cdot l_G(v) + x_G(v)$. Since the degrees of both vertices are odd, we know that $x_G(u)$ and $x_G(v)$ are odd as well. Since the smallest odd integer is 1, we know there is at least one edge between u and v. The path is simply such edge. □

◇

Exercise 3 (Dijkstra's algorithm)**.** Describe an algorithm to carry out the following tasks:

1. Find the shortest path to *all* nodes from a given *source* node given the graph only contains positive weights.
2. Detect a *cycle* where the sum of the weights of the edges of that cycle is strictly negative. Evidently the graph can contain edges with negative weights.
3. The shortest path for a graph in which edges can have negative weights. The graph doesn't contain cycles such that the weight is strictly negative.
4. Given a graph containing only positive weights, find two paths: from a to b and from b to a such that the two paths don't have any edges in common and the *total* weight - the sum of the two paths - is minimized.
5. Find the *longest* path for a given graph with only positive weights.

Prove that your algorithm is correct.

Hint. In many cases a slight modification of *Dijkstra's algorithm* will be sufficient. ◇

Answer. First we will briefly sketch the working principle of Dijkstra's algorithm. Dijkstra's algorithm labels vertices considers a priority queue. Initially all vertices are labeled with $\langle +\infty, \textbf{false} \rangle$ except the source vertex s which is labeled with $\langle 0, \textbf{false} \rangle$. The label contains the shortest path to the corresponding vertex up to that point of the algorithm. Dijkstra's algorithm schedules the source vertex s in the priority.

As long as there are items in the priority queue, the first item is selected. If the node is already visited (the second element of the label of the vertex), we ignore the vertex. Otherwise we first set the second element of the label to **true**, the first element of the label is called the *weight* of the vertex w_v. Next we iterate over all outgoing edges $\langle v, u \rangle \in E(v)$ of v, we calculate the target weight w_u as $w_u = w_v + w_e$ with w the weight of the edge. If the target weight is lower than the weight of the vertex u at that moment, we modify the weight and schedule u in the priority queue. Otherwise, we leave u unmodified. The algorithm stops when the destination vertex d is scheduled first in the priority queue.

Proof. Dijkstra's algorithm works correct because it guarantees progression if the weights of all edges are positive: each time a node is expanded, the weight of that node is the weight shortest path between the source and that vertex. This is true, because the priority queue is dequeued by increasing weight. Given there is shorter path $\langle v_1, v_2, \ldots, v_n \rangle$ to vertex that is expanded, the the weight of vertex v_{n-1} has a weight lower than the weight of the vertex. But in that case v_{n-1} would have been expanded earlier and thus further have minimized the weight for v_n. □

1. We only need to modify the stop criterion of Dijkstra's algorithm. Now the algorithm stops if the priority queue is empty.

   Proof. This algorithm is correct because eventually all vertices connected with the given source index will be scheduled: if the vertex v is connected with the source s, there is a path $\langle s, v_1, v_2, \ldots, v \rangle$ between the the source vertex and that vertex. Since we definitely expand s, v_1 is placed on the priority queue, and each vertex v_i is scheduled because at least the previous vertex v_{i-1} is scheduled. So at least all vertices will eventually get scheduled. Since Dijkstra's algorithm only expands a node if there is no shorter path between the source and that node, all connected nodes are expanded and marked with the weight of the shortest path. □

2. One can use *Bellman-Ford's algorithm* this algorithm as well as it's proof are explained in the next item.

3. This is the well known *Bellman-Ford algorithm.* Dijkstra's algorithm works on a node-by-node basis where each node is placed in a queue and processed. Since Dijkstra's algorithm guarantees - based on the fact that all edges have positive weights - progression, we know for sure that the node expanded is optimal. This is no longer possible if the graph can contain negative edges. *Bellman-Ford's algorithm* still labels the vertices but only with the weights. All vertices are thus labeled with ∞ except the source node which is labeled with 0. The the following step is repeated n times with $n-1$ the number of vertices or until the weights of the vertices no longer change if that occurs earlier.

   > Iterate over all edges $e = \langle u, v \rangle \in E$. If u has a weight w_u and e weight w_e, if $w_u + w_e \leq w_v$, then set $w'_v = w_u + w_e$.

   Now we perform the operation a final time. If the weights still change, we have obtained a negative cycle so there is no shortest path, if not we can reconstruct the path by backwards reasoning as we do for *Dijkstra's algorithm.*

   Proof. We will only prove the case of iterating the step $n-1$ times. One can easily modify the proof to show that it works if one iterates until the weights no longer modify. Say the optimal path between the source s and d is $\langle s, v_1, v_2, \ldots, v_m, d \rangle$. Since the number of vertices is n there cannot exists a longer (shortest) path than a path with $n-1$ edges. This thus means that $m \leq n-2$. The first time we call the subroutine, all vertices are labeled ∞ except the source. After applying the first step, all vertices to which an edge from the source are labeled with a value that is optimal for paths with *one* edge or lower (for zero edges, we can only reach the source node which has weight 0).

   Given we have iterated over the first i steps - thus obtain the shortest path to each vertex *with i edges* or lower - now we iterate over all edges we thus virtually "add" an edge to the optimal paths with i or less edges. If the constructed path is shorter (less weight), that means that the last vertex of the end has a weight that was higher. We thus replace is with the weight of the new path. After we have done this for every edge, the vertices are labeled with the shortest path with $i+1$ edges or less.

Since we do this $n-1$, times, the vertices are labeled with a path that contain $n-1$ or less edges. This is the largest path with respect to the number of edges possible. If by applying the step again weights change, this means that for at least one vertex, there exists a path of length n results in a shorter path than that of $n-1$. A path that contains n edges however contains at least one edge twice, this thus means there is a loop somewhere. In that case it is beneficial to keep looping over the graph, otherwise one would never have added the edge twice.

Given there is no such loop, we have calculated the weights for all the vertices for paths with at most $n-1$ edges. Since we know there are no cycles, it means we have iterated over all cycle-free paths and that the destination vertex v will be marked with the shortest path. □

4. This is the well known *Suurballe algorithm*. Suurballe's algorithm makes use of Dijkstra's algorithm twice. The algorithm works as follows:

   (a) First run Dijkstra's algorithm on the graph and return the label of all vertices $\vec{w}$ together with the shortest path $\langle v_1, v_2, \ldots, v_n \rangle$;

   (b) Now modify the weights of every edge $e = \langle u, v \rangle \in E$ with $w'_e = w_e - w_v + w_u$; and remove the *directed edges* $\langle v_i, v_{i+1} \rangle$ of the shortest path from the graph: if the edge $\langle v_i, v_{i+1} \rangle$ in G is directed, remove the edge; if the edge was undirected, make it directed such that it is outgoing from v_{i+1} and incoming from v_i;

   (c) Now run Dijkstra's algorithm again on the modified graph again from source to drain, resulting in an optimal path $\langle u_1, u_2, \ldots, u_m \rangle$;

   (d) Now we need to inspect the two paths. If the first path contains an edge $\langle v_i, v_{i+1} \rangle$ where the second path contains an edge $\langle u_j, u_j + 1 \rangle = \langle v_{i+1}, v_i \rangle$ - in other words they share an edge but in the different direction - one needs to perform a *swap*-operation. The swap operation modifies the paths to $\langle v_1, v_2, \ldots, v_i, u_{j+2}, \ldots, u_m \rangle$ and $\langle u_1, u_2, \ldots, u_j, v_{i+2}, \ldots, v_n \rangle$. In other words, you remove the edge from the graph by swapping the path ends. The operation is repeated until the paths are no longer modified;

   (e) By putting the second (or the first) path in reverse order, one obtains a cycle with no common edges that is optimal.

   Proof. The algorithm terminates because all substeps sterminate. Dijkstra's algorithm terminates and we call it twice so step (a) and (c) terminate. Modifying the weights can be done by iterating over the edges, since the number of edges is finite, this step terminates as well. Finally removing edges that are contained in both paths definitely terminates, since the paths only contain a finite amount of edges.

   Evidently the first path is the shortest path for the original graph. When we call Dijkstra's algorithm on the modified graph, the weight of each path $e = \langle s, v_1, \ldots, v_m, t \rangle$ will be equal to $w'_e = w_e + w_t - w_s$. So the weight of the path we obtain is the weight of that path for the original graph *minus* the weight of the shortest path of the original graph. We are thus reasoning about the extra weight of the "shortest path" compared to the. We thus obtain the shortest path, and the shortest path - with respect to the original graph - where we removed the edges. By merging them we obtain the shortest cycle containing the two points. □

5. If the graph does not contain positive edges, one can use the *Bellman-Ford algorithm* where we replace the weight of every edge with its negative counterpart. This is however not an assumption we can make. One can expect that if there is a positive cycle, we simply keep running through that cycle. The definition of a path however states that it can contain every edge at most once. If thus enter a positive cycle, we need to exit the cycle at last before we traverse an edge for the second time. You cannot incorporate this behavior in Dijkstra's algorithm because the edges it has already traversed is a

property that scales with the length of the path. There are thus an exponential amount of paths each with *different context*. As a result one has to implement a backtracking algorithm that considers all possible paths. The problem is NP-HARD and there is an easy reduction from the *Hamiltonian cycle problem*.

Proof. Since the backtracking algorithm iterates over all possible paths up to length $n-1$ - thus all real paths - and evaluates the weight of these paths, evidently the algorithm will come up with the longest path. □

◇

Exercise 4 (Assignment problem)**.** 1. Say you have n employees $\vec{e} = \langle e_1, e_2, \ldots, e_n \rangle$. On a certain day, there are m tasks $\vec{t} = \langle t_1, t_2, \ldots, t_m \rangle$ that must be carried out. Each employee can perform a given maximum number of tasks $\vec{b} = \langle b_1, b_2, \ldots, b_n \rangle$ that day and furthermore some tasks require the employee to be certified. A matrix C describes the certificates of the employees: an employee e_i is certified to carry out task t_j if and only if $c_{i\,j} = \textbf{true}$. Describe a way to convert this problem into a graph and that - using a graph algorithm - you can come up with a solution for this problem.

2. Show - by running your algorithm - an assignment for:

$$n = 3 \tag{3}$$
$$m = 4 \tag{4}$$
$$\vec{b} = \langle 1, 2, 1 \rangle \tag{5}$$
$$C = \begin{pmatrix} \textbf{true} & \textbf{false} & \textbf{false} & \textbf{true} \\ \textbf{true} & \textbf{false} & \textbf{true} & \textbf{false} \\ \textbf{false} & \textbf{true} & \textbf{false} & \textbf{true} \end{pmatrix} \tag{6}$$

Answer. You can represent the problem as a graph with four kinds of nodes:

1. the source node w;
2. the sink node z;
3. the employee nodes x_i; and
4. the task nodes y_i.

For each employee e_i there exists an edge between the source w and the employee node x_i with capacity the number of tasks that employee can handle b_i. Furthermore for each task t_i we consider a node y_i that is connected with the sink node z with capacity 1. If an employee e_i is certified to carry out task t_j (thus $c_{ij} = \textbf{true}$), we add an edge with capacity 1 between employee node x_i and task node y_j. The graph for the given values is shown on Figure 2.

We can now generate an assignment by running the *Maximum Flow algorithm* on the generated graph.

We will now demonstrate the *Maximum Flow algorithm* on the graph (Figure 2) as shown in Figure 4. ◇

Exercise 5 (Graph properties)**.**

1. Give a formula that describes the number of simple paths of length k between two different vertices in a graph $K_{n;n}$?
2. Can one draw a graph with n vertices such that every vertex has a distinct degree?

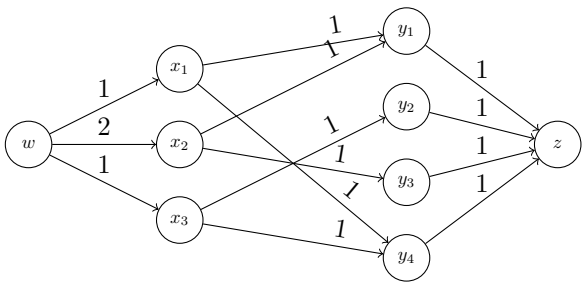

Figure 2: The graph produced after applying the discussed transformation.

3. Show that for every n, $K_{n;n}$ has a subgraph isomorphic to C_k - a cycle of length k - for every $k \in \{4, 6, \ldots, 2 \cdot n\}$.

4. Show that for every $n \geq 2$, $K_{n;n}$ is a Hamiltonian graph.

Answer.

1. We will denote the first set of vertices with A and the second set of vertices with B. By definition, there exists only edges between every two vertices $a_i \in A$ and $b_j \in B$. The two "marked" vertices for which we want to count the number of paths are denoted with u and v.

   If $k \geq 2 \cdot n$, then evidently there exists no such path: in that case the path has visited every single vertex at least once. Since a simple path visits every vertex at most once, there is no edge that can be added to the path. If $u \in A \Leftrightarrow v \in A$, then the number of edges k must be even, otherwise the number of paths is zero. The same holds for $u \in A \Leftrightarrow v \in B$: k must be odd, otherwise the number of paths is zero as well.

   Given u and v are in different sets and k is odd and not too long (see previous paragraphs), we can calculate the number of paths by looking at the number of ways we can move to the right and left. The first time we can move to $n-1$ vertices. Indeed: we can move to all vertices in the right set except the final vertex v. Next we move to the left and have $n-1$ options as well. If we move again to the right, we now have $n-2$ options, the same holds if we move again to the left. The i-th time we move to the right and back, we have thus $n-i$ options. And we perform $k-1/2$ such movements. The last movement to the right, we have only one option: reaching v. The formula is thus:

$$\prod_{i=1}^{k-1=2} (n-i)^2 = \left(\frac{(n-1)!}{(k-1/2)!}\right)^2 \tag{7}$$

   In the case both vertices originate from the same set, k must be even. We make the assumption the path is not too long. We will assume that both vertices are located in the left set, the proof is completely analogue if the two vertices are placed in the right set. If we first move to the right we have n possible vertices we can select. If we move back to the left, we have $n-2$ possible vertices, except if the end of the path is reached, in that case we have only one vertex to pick: v. We keep iterating over moves to the right and left until we reach the length k. For convenience, we split the formula into two different products:

$$\left(\prod_{i=1}^{k=2} (n-i+1)\right) \cdot \left(\prod_{i=1}^{k=2-1} (n-i-1)\right) = \frac{n!}{(k/2)!} \cdot \frac{(n-2)!}{(k/2-1)!} \tag{8}$$

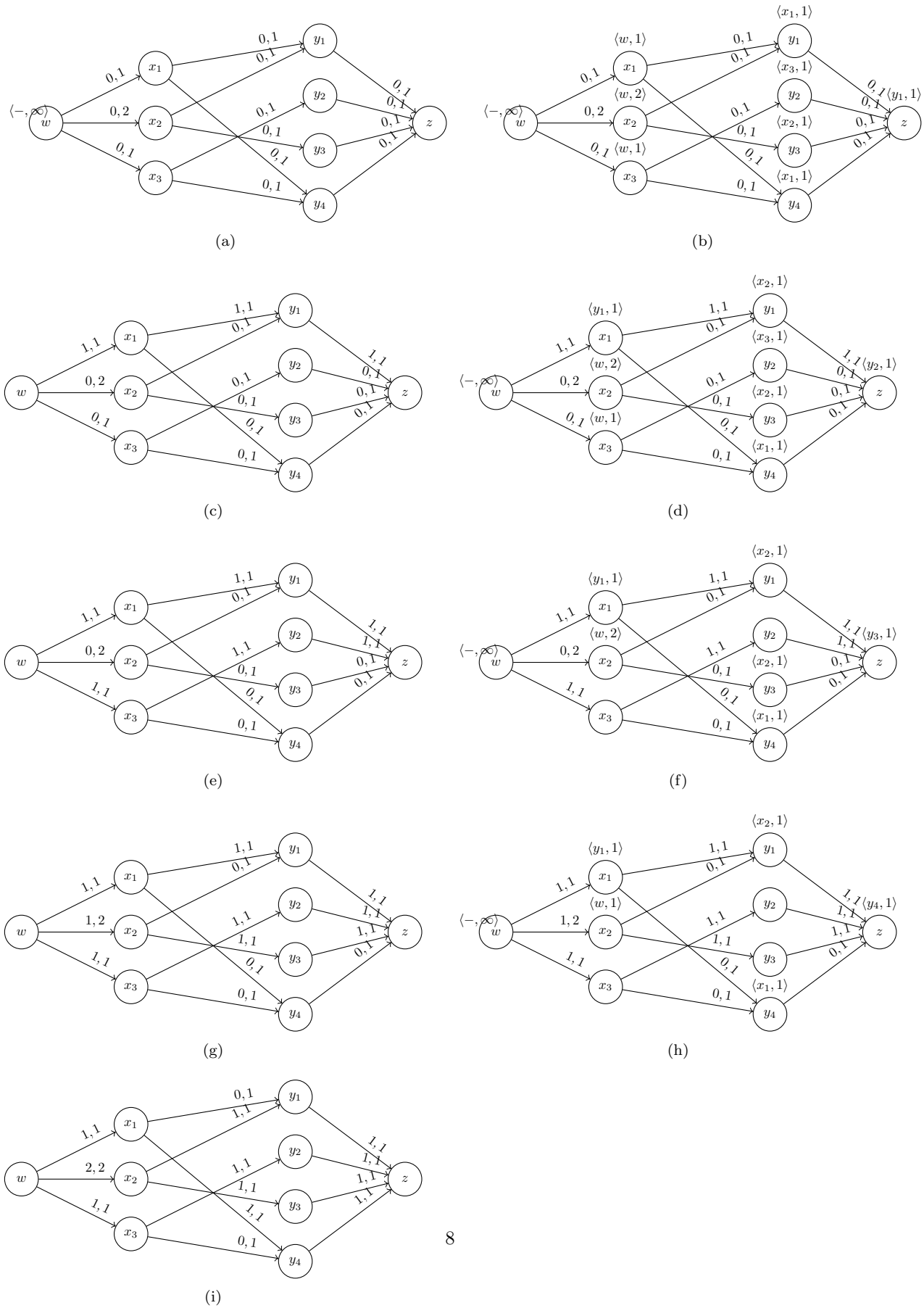

Figure 3: Evolution of the maxflow algorithm for the given graph.

2. Not for simple graphs. Since the property must hold for every $n > 0$, we consider a simple graph with two vertices v_1 and v_2. There are only two possible graphs: a graph where there is no edge between v_1 and v_2 and the graph where such edge exists. In the first case both vertices have degree 0, in the other case both vertices have degree 1.

   For general graphs, one can construct such a graph. Simple consider a graph with n vertices with no edges between the vertices. Add i loops to the i-th vertex. Now vertex i has a degree of $2 \cdot i$.

3. Given a graph $K_{n;n}$ we will label the vertices of the first set with $a_i \in A$ and the vertices of the second set with $b_j \in B$. By definition there only exist edges between every $a_i \in A$ and every $b_j \in B$. Now for a given k in $\{4, 6, \ldots, 2 \cdot n\}$, we can define a cycle graph C_k as $a_1 \to b_1 \to a_2 \to b_2 \to \ldots \to a_{k=2} \to b_{k=2} \to a_1$. It is clear that all the edges in the path are edges from $K_{n;n}$ since we alternate between a_i and b_j. Furthermore we never use an edge twice since each time an edge starts from a vertex in A, the index has incremented.

4. If we take the previously described path and set $k = 2 \cdot n$ $a_1 \to b_1 \to a_2 \to b_2 \to \ldots \to a_n \to b_n \to a_1$, we've visited every vertex exactly once and didn't visit an edge more than once. This works for every $n \geq 2$. For $n = 1$, there exists no cycle since there is only one edge between a_1 and b_1 and we would need to traverse it twice in order to generate a cycle.

◇

Exercise 6 (Eulerian and Hamiltonian paths)**.**

1. Can a graph contain both a Eulerian and Hamiltonian path? If so, draw such graph.

2. Can a graph contain a path that is both Eulerian and Hamiltonian? If so, what can you say about the shape of such graphs? Prove this.

3. An n-cube can be defined inductively as follows:

   > *Definition* 1 (n-cube). A 1-cube is a graph that contains exactly one vertex and no edges. An n-cube consists of two $n-1$ cubes such that the corresponding vertices of the two cubes are connected.

   For which values for n does an n-cube contain a Hamiltonian path? Prove your answer. Show paths for the 3- and 4-cube.

Answer.

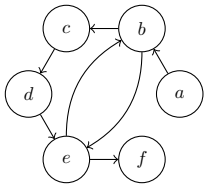

Figure 4: A graph that contains both a Hamiltonian and Eulerian path.

1. Yes. See Figure 4. The Hamiltonian path is $a \to b \to c \to d \to e \to f$. The Eulerian is $a \to b \to e \to b \to c \to d \to e \to f$.

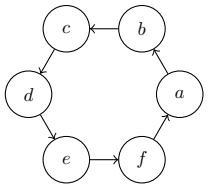

Figure 5: A graph that contains a path that is both Hamiltonian and Eulerian.

2. Yes. See Figure 5. There are only two possible graphs-families that contain such path, a cycle graph C_n and a path graph P_n[2].

   Proof. A path that is both Eulerian and Hamiltonian visits all vertices and edges exactly once. Given the path $v_1 \to v_2$, if $v_1 = v_2$, then we speak of a cycle. This means the graph contains an loop for v_1. If $v_1 \neq v_2$, then the only possible graph is a graph with two vertices v_1 and v_2 with one edge in between. Now we add a vertex in between the path. If the path is $v_1 \to v_3 \to v_2$, since every vertex is visited exactly once, we know for sure that $v_1 \neq v_3 \neq v_2$. Again in the case $v_1 = v_2$, we speak of a cycle. Since the we visit every vertex exactly once, the graph consists in that case out of 2 vertices v_1 and v_3 and there are two edges: one from v_1 to v_3 and vice versa. If $v_1 \neq v_2$ we can only construct a path graph. By induction where we each time add a vertex in between, we can proof that we can only generate graphs that belong to the C_n or P_n graph family. □

3. From $n \geq 2$ one can generate such Hamiltonian path.

   Proof. We proof this by induction.

   Base case Given a 2-cube with two vertices v_1 and v_2 a trivial Hamiltonian path is $v_1 \to v_2$. Since a 2-cube contains two vertices with exactly one edge in between, evidently this is a Hamiltonian path.

   Induction Given two n-cubes, the $n+1$-cube that consists out of the two cubes contains a Hamiltonian path as well. Since the n-cubes have the same shape, evidently they have the same Hamiltonian path(s). We will denote this path for the first n-cube as $v_1 \to v_2 \to \ldots \to v_{2^{n-1}}$. The equivalent path for the second cube is denoted by $v'_1 \to v'_2 \to \ldots \to v'_{2^{n-1}}$. Since the edges are not directed, we can reverse the path, and the resulting path is still a Hamiltonian path: $v'_{2^{n-1}} \to v'_{2^{n-1}-1} \to \ldots \to v'_1$. Since v_i has an edge to v'_i we can connect $v_{2^{n-1}}$ with $v'_{2^{n-1}}$. The resulting path is $v_1 \to v_2 \to \ldots \to v_{2^{n-1}} \to v'_{2^{n-1}} \to v'_{2^{n-1}-1} \to \ldots \to v'_1$. Since the first two paths visited every vertex once, and we simply concatenated the paths, the resulting path visits every vertex in both cubes. So the resulting path is Hamiltonian as well. □

◇

Exercise 7 (Trees)**.**

1. How many non-isomorphic trees can be constructed with n vertices?
2. Draw the 11 non-isomorphic trees with 7 vertices.

Answer.

[2]A path graph is graph that consists out of n vertices such that v_i is connected to v_{i-1} and v_{i+1}, there is no wrap-around.

1. One can first determine the number of *labeled* trees one can draw with n vertices using double counting:

$$\frac{\prod_{i=1}^{n-1} i \cdot n}{n!} = \frac{(n-1)! \cdot n^{n-1}}{n!} = n^{n-2} \tag{9}$$

The problem is that these trees are labeled: we don't take isomorphism into account. Now the algorithm can (randomly) construct a tree. Given a tree, one can calculate the automorphism of the tree: given we can swap the nodes, how many of such trees can be generated. An upper bound one can take into account is that if there are k_i nodes with degree d_i the upperbound is:

$$\prod_i k_i! \tag{10}$$

Furthermore it's a matter of looking at the structure of the tree. The number of automorphic trees for a given tree t is labeled A_t. The number of isomorphic trees is than defined by:

$$I_t = \frac{n!}{A_t} \tag{11}$$

Each time we succeed in finding a new tree that is not isomorphic with previous discovered trees, we can calculate the number of isomorphic trees it represents and subtract this from the total. The moment we reach zero, we know we've found all trees.

2. See Figure 6.

◇

Exercise 8 (Independent set)**.**

1. Construct an algorithm that given a graph $G(V, E)$ you subdivide the set of vertices into $\mathcal{V} = \{V_i\}_{i=1}^{n}$ such that $\forall i, j : V_i \cap V_j = \emptyset$, $\cup_{i=1}^{n} V_i = V$ and for each $\langle a, b\rangle \in E$, $a \in V_i \wedge b \in V_j \Rightarrow V_i \neq V_j$.
2. The *edge cover problem* is a problem where given a simple connected graph $G(V, E)$ one aims to find a set of edges E' such that each vertex $v \in V$ is adjacent to at least one edge $e \in E$. For the *minimum edge cover problem* the number of edges $|E|$ must be minimal.
   (a) Describe a polynomial algorithm that calculates an edge cover.
   (b) Describe a polynomial algorithm that finds a minimum edge cover.
   (c) Is it harder to find a minimal edge cover if we work with weighted edges?

Answer.

1. Since the number of sets is not supposed to be minimal, we can consider an algorithm that initially uses 0 sets of vertices. Then it iterates over all vertices. For each vertex v_i it looks whether there is a set such that none of the constraints are harmed. If no such set exists, we construct a new singleton set with this vertex. Otherwise we add the vertex to one of the already existing sets.
2.
   (a) One can simply return the full set of edges. Since the graph is connected this means that all vertices are adjacent to at least one edge. By returning the set of edges, all vertices are adjacent.
   (b) The minimum edge cover can be constructed by first constructing the *maximum matching* (or *maximal matching*) this can be done by adding each time an arbitrary edge such that they don't share vertices. Next we simple again iterate over the set of edges and add all edges that are connected to a vertex that wasn't part of the edge cover yet.
   (c) By sorting the set of edges first, one can generated the minim edge cover in polynomial time. Depending on the implementation, this can result in an additional logarithmic time complexity.

◇

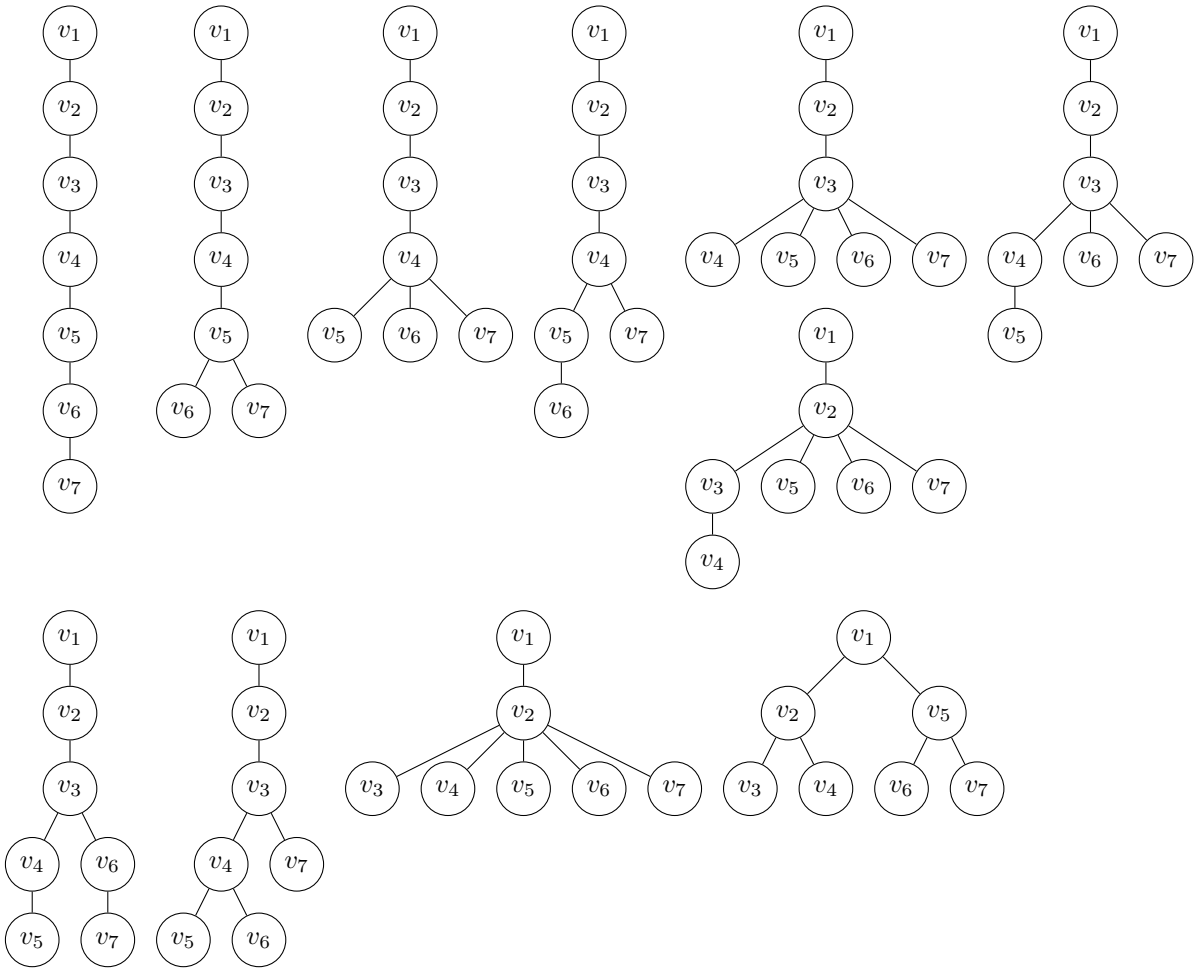

Figure 6: The 11 non-isomorphic trees with 7 vertices.