JAVIER NAVARRO SORIANO
A20473222

HOMEWORK 5 - Combinatorial optimization

Problem 1

The problem for finding flows when even the nodes have capacities is so similar to the one we already know from the linear program of the max-flow problem. Only one more contraint concerning the capacity of the vertices has to be taken into accont.

Therefore, given a graph $G(V,E)$; the following constraint has to be added to the typical linear program of max-flow problem:

$$\underbrace{\sum_{i=1}^{n} f_{ij} \leq \text{Capacity of the vertex } j.}_{\substack{\text{sum of the flows throughout a vertex} \\ \text{must be less or equal to the capacities} \\ \text{of the corresponding vertex}}}$$

$n = |V|$ = number of vertices

Linear program

$$\max \quad f_s \quad \text{flow out of } s$$

$$\text{s.t} \quad f_{uv} \leq \underbrace{C_{uv}}_{\text{capacity of the edge } (u,v)}$$

Conservation contraint $\boxed{\sum_u f_{uv} = \sum_w f_{vw}}$

$\boxed{\sum_u f_{uv} \leq C_v}$ New constraint

$$f_{uv} \geq 0$$

$\Rightarrow$

$$\boxed{\begin{aligned} \max \quad & f_s \\ \text{s.t} \quad & f_{uv} \leq C_{uv} \\ & \sum_u f_{uv} = \sum_w {vw} \\ & \sum_u f_{uv} \leq C_v \\ & f_{uv} \geq 0 \end{aligned}}$$

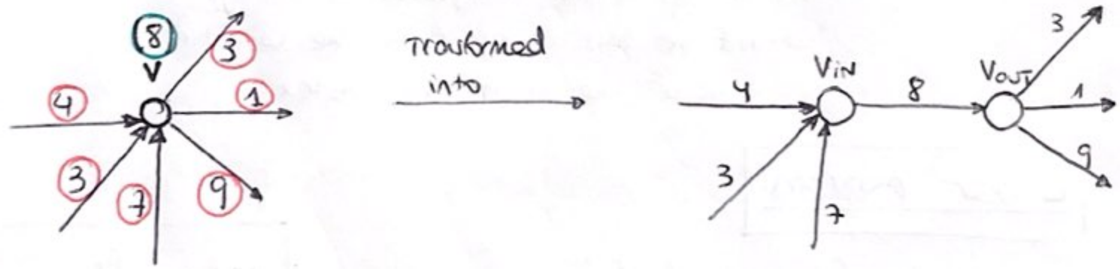
As we have said, this model is a linear program, so it can be solved in a polynomial time.

MORE DIRECT WAY OF SOLVING THE PROBLEM

There is a simple reduction from the max-flow problem with node capacities to a regular max-flow problem:

For every vertex v in your graph, replace with two vertices v_{in} and v_{out}. Every incoming edge to v should point to v_{in} and every outgoing edge from v should point from v_{out}. Then, create one additional edge from v_{in} to v_{out} with capacity c_v (the capacity of vertex v).

Example



edge capacities

node capacities

Thus, by doing this for each of the nodes, we would transform the problem into a simple max-flow problem with edges capacities, being a more direct way to solve the problem.

Problem 2

A minimum throughput algorithm for finding blocking flows that solve the maximum flow problem is Dinic's Algorithm.

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices: Source 's' and sink 't' in the graph; find the maximum flow from s to t with the following constraints:

a) Flow on an edge doesn't exceed the given capacity of the edge.

b) Incoming flow is equal to outgoing flow for every vertex except s and t.

To this type of problem we can use Ford-Fulkerson algorithm, but there is an algorithm which is a faster algorithm and takes $O(EV^2)$.

DINIC'S ALGORITHM

These are the concepts that follows:

a) A flow is maximum if there is no s to t path in residual graph.

b) BFS is used in a loop. There is a difference though in the way we use BFS in this algorithm.

In Dinic's algorithm, we use BFS to check if more flow is possible and to construct level graph. In level graph, we assign levels to all nodes (level of a node is shortest distance of the node from source).

Once level graph is constructed, we send multiple flows using this level graph.

In addition, a flow is blocking flow if no more flow can be sent using level graph.

Algorithm

Initialization 1: Let f be the zero flow ($f_e = 0$ for all $e \in E$);

ITERATIONS

2: while There exist augmenting ($s \to t$) paths do:

a) Perform a BFS search that stores the distance from s to each vertex;

b) Let l be the distance of t;

Find the blocking flow on each iteration

while there exist augmenting ($s \to t$) paths of length l do:

a) Get an augmenting path P with a DFS search using the vertices at distance l or less;

b) Send δ units of flow through P.

end

end

Hence, the algorithm consists of several phases. On each phase we construct the layered network of the residual network of G. Then, we find an arbitrary blocking flow in the layered network and add it to the current flow.

Correctness

Let's show that if the algorithm terminates, it finds the maximum flow.

If the algorithm terminated, it couldn't find a blocking flow in the layered network. It means that the layered network doesn't have any path from s to t. It means that the residual network doesn't have any path from s to t. It means that the flow is maximum. (*)

Moreover, the algorithm terminates in less than V phases. To prove this, we must firstly prove that the distance from s to each vertex don't decrease after each iteration (for example, $level_{i+1}(v) \geq level_i(v)$).

<u>Proof.</u>

Fix a phase i and a vertex v. Consider any shortest path P from s to v in the residual graph (G^R_{i+1}). The length of P equals $level_{i+1}(v)$. Note that G^R_{i+1} can only contain edges from G^R_i and back edges from G^R_i. If P has no back edges from G^R_i, then $level_{i+1}(v) \geq level_i(v)$ because P is also a path in G^R_i. Now, suppose that P has at least one back edge. Let the first such edge be (u, w). Then $level_{i+1}(u) \geq level_i(u)$. The edge (u, w) doesn't belong to G^R_i, so the edge (w, u) was affected by the blocking flow on the previous iteration. It means that $level_i(u) = level_i(w) + 1$. Also, $level_{i+1}(w) = level_{i+1}(u)$ $+ 1$. From these two equations and $level_{i+1}(u) \geq level_i(u)$ we obtain $level_{i+1}(w) \geq level_{i+1}(w) + 2$. Now we can use the same idea for the rest of the path.

It's also needed to prove that $level_{i+1}(t) > level_i(t)$. To prove this suppose that $level_{i+1}(t) = level_i(t)$. Note that G^R_{i+1} can only contain edges from G_i^R and back edges for edges from G_i^R. It means the there is a shortest path in G_i^R which wasn't blocked by the blocking flow. It's a contradiction.

Hence, I have showed the correctness of this minimum throughput algorithm for finding blocking flows to solve the maximum flow with this two proofs and (*).

TIME COMPLEXITY

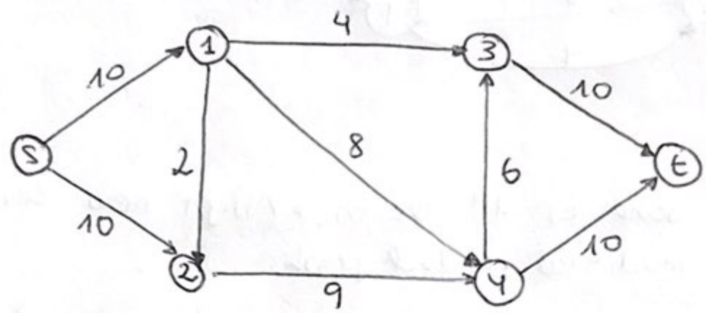
In order to find the blocking flow on each iteration, we may simply try pushing flow with DFS from s to t in the layered network while it can be pushed. In order to do it more quickly, we must remove the edges which can't be used to push more. To do this we can keep a pointer in each vertex which points to the next edge which can be used. Each pointer can be moved at most E times, so each phase works in $O(VE)$.

In addition, there are less than V phases, so the total complexity is $\boxed{O(V^2E)}$

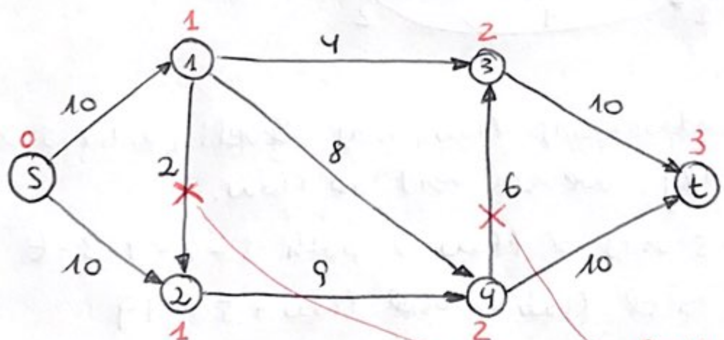
Finally, let's show an example to understand what we have explained.

EXAMPLE

Initial residual graph = given graph.



1^{st} iteration. We assign levels to all nodes using BFS. We also check if more flow is possible (or there is a s-t path in residual graph).



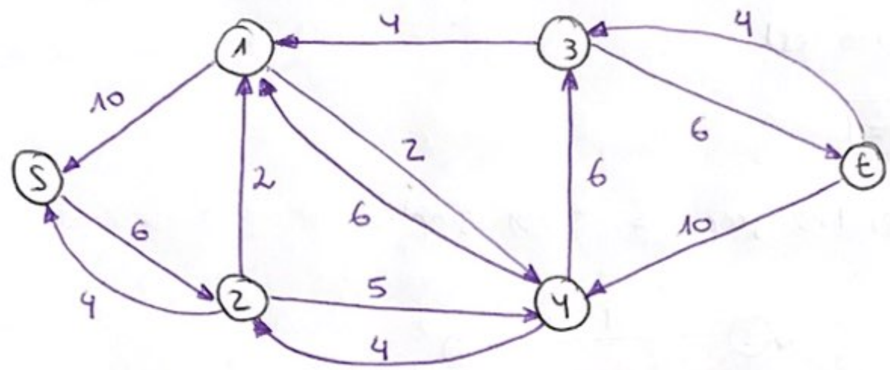
Cross edge can't be used to send more flow in this iteration because these edges don't connect node from i th to i+1 th level.

Now, we find blocking flow using levels. We send three flows together.

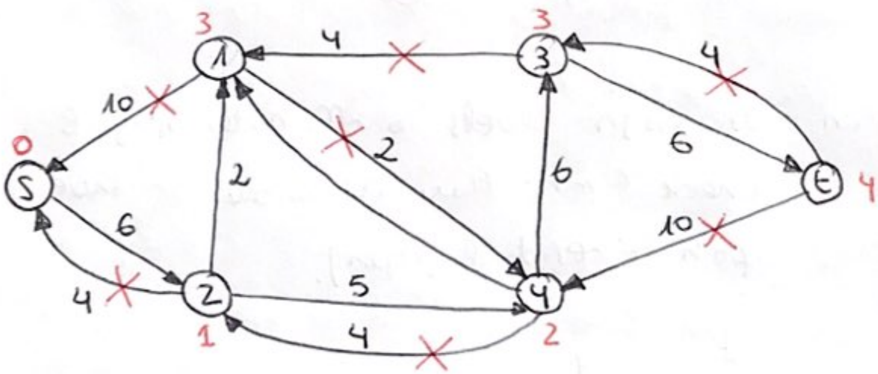
- 4 units of flow on path s - 1 - 3 - t
- 6 units of flow on path s - 1 - 4 - t
- 4 units of flow on path s - 2 - 4 - t

} Total Flow = 14

New Residual graph



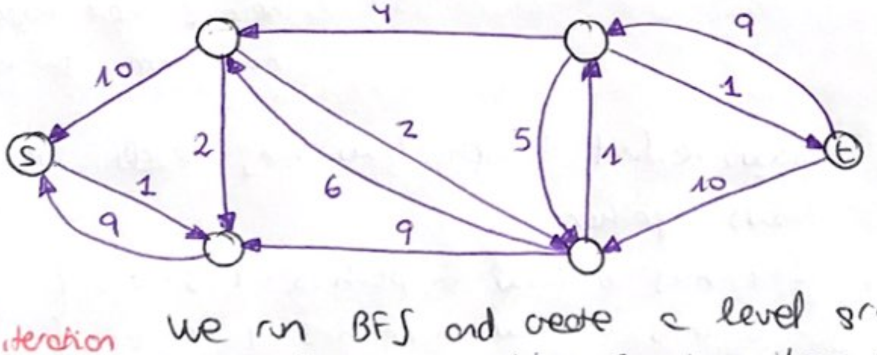
2nd Iteration Same as 1st iteration → Assign new levels to the modified residual graph.



Those edges that are not going from ith level to (i+1)th level are crossed.

Find blocking flow using levels (0, 1, 2, 3 or 4).
Only, we can send one flow.
- 5 units of flow on path s - 2 - 4 - 3 - t
Total flow = total flow + 5 = 19

New residual graph



3rd iteration We run BFS and create a level graph. We also check if more flow is possible. This time there is no s-t path in residual graph, so we terminate the algorithm. $\boxed{\text{TOTAL FLOW} = 19}$

Problem 3

a) Let G be a directed graph with n vertices, and let s and t be two vertices in G. The s-t connectivity of G is the maximum number of simple vertex-disjoint paths from s to t in G.

To find the maximum number of s-t vertex disjoint path in the directed graph G, let us formulate given problem to a max flow problem as follows:

ALGORITHM

For each node in G:
    make it two nodes v_{in} and v_{out}.
    add an edge from v_{in} to v_{out} with capacity 1.

For each edge (u,v) in G:
    put the edge from u_{out} to v_{in} with capacity 1

Then, run the Ford-Fulkerson algorithm to find the max-flow from s_{out} to t_{in}. Now, the value of the max-flow is the number of vertex-disjoint paths.

Maxflow value at t is the maximum number of the vertex-disjoint paths from s to t. Because each edge's capacity is one, at each node there will be one outgoing path.

TIME COMPLEXITY: the time complexity of the Ford-Fulkerson algorithm is $O(VE) = O(nn^2) = O(n^3)$ in worst case if there is an edge between each pair of vertices.

b) $c(G) \leq 2|E| / |V|$

To demonstrate $c(G) \leq 2|E|/|V|$, the first thing to do is to comment on the relationship between connectivity and the minimum degree of a graph ($\delta(G)$). This is done, since from knowing this minimum degree, it's possible to reach $c(G) \leq 2|E|/|V|$.

Definitions

$c(G) \equiv$ connectivity of a graph. It's the minimum s-t connectivity (which is the maximum number of edge-disjoint paths from s to t) over all pairs s and t. Another definition of connectivity is that $c(G)$ is the minimum number of elements that needs to be removed to separate the remaining nodes into two or more isolated graphs.

$\delta(G) \equiv$ minimum degree of a graph. It's the degree of the vertex with the least number of edges incident to it.

RELATION BETWEEN CONNECTIVITY & MINIMUM DEGREE OF G

Claim

For every graph G:

(a) $c(G) \leq \delta(G)$

(b) if $\delta(G) \geq n-2$, then $\kappa(G) = \delta(G)$.

Proof (a)

Let G be a graph of order n and let $v \in V(G)$ be a vertex in G such that the degree of v is the minimum degree of G ($\delta(G)$). Then, we can disconnect v from G by removing $\delta(G)$ edges. Hence, the connectivity of G must be smaller or equal to $\delta(G)$. $\Rightarrow$ $\boxed{c(G) \leq \delta(G)}$

Proof (b)

Let G be a graph of order n such that $\delta(G) \geq n-2$.

If $\delta(G) = n-1$ then G is complete and by definition $c(G) = n-1 = \delta(G)$

Note that $\delta(G)$ can not be bigger than $n-1$ for simple graphs, so we are left with the case for $\delta(G) = n-2$. Let $S = \{v_1, v_2 \dots, v_{n-2}\}$ be a subset of $V(G)$ such that $c(G-S) = 0$. Suppose we can remove one vertex from S, and still have $G-S$ be disconnected. But then $|V(G-S)| = n-(n-3) = 3$, and the lowest degree for a vertex in $G-S$ is $\delta(G-S) = \delta(G) - |S| = n-2-(n-3) = 1$, so it must be conected. Therefore, if $\delta(G) = n-2$, then it must be the case that $\boxed{c(G) = \delta(G)}$

After this, all that remains is to demonstrate that $\delta(G) \leq 2 \dfrac{|E|}{|V|}$.

Claim

The minimum degree in a graph, $\delta(G)$, is less than or equal to twice the number of edges of G divided by the number of vertices in G, that is $\delta(G) \leq \dfrac{2|E|}{|V|}$.

Proof

We know that $2 \cdot \dfrac{|E|}{|V|}$ represents the average degree in a graph G.

In addition, we know that $\sum_{v \in V(G)} \deg(v) = 2|E|$.

Hence, it follows that $\delta(G)$ is the minimum degree in the graph:

$$\sum \delta(G) \leq \sum_{v \in V(G)}$$

$$n\,\delta(G) \leq 2|E|$$

$$\delta(G) \leq 2\frac{|E|}{n}$$

But we know that n represents the number of vertices in the graph. So,

$$\boxed{\delta(G) \leq 2\frac{|E|}{|V|}}$$

Putting together all that has been demonstrated so far, we get the following:

$$\boxed{c(G) \leq \delta(G) \leq 2\frac{|E|}{|V|}}$$

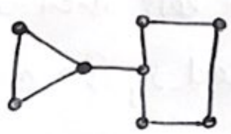
Let's see some examples:

1º 

$|V|=3$ $|E|=3$

$c(G) = 2$
$\delta(G) = 2$

$c(G) = \delta(G) = 2 \cdot \frac{3}{3} = 2$

2º 

$|V| = 8$
$|E| = 9$

$c(G) = 1$
$\delta(G) = 2$

$c(G) < \delta(G) < 2 \cdot \frac{9}{8} = \frac{9}{4}$

3º 

$|V| = 6$
$|E| = 12$

$c(G) = 4$
$\delta(G) = 4$

$c(G) = \delta(G) = 2 \cdot \frac{12}{6} = 4$

Everything I have shown actually refers to vertex connectivity ($c(G)$).
There is another type of connectivity called edge-connectivity,
whose definition is:

$\lambda(G) \equiv$ edge connectivity. It's the minimum number of edges whose deletion from a graph G disconnects G.

If the same demonstration is made up to now, the same conclusion will be reached. All together would be:

$$\boxed{c(G) \leq \lambda(G) \leq \delta(G) \leq 2\frac{|E|}{|V|}}$$

vertex connectivity — edge connectivity — minimum degree of the graph

By means of the relation found with the minimum degree of a graph ($\delta(G)$) and the connectivity ($c(G) \equiv$ vertex connectivity or $\lambda(G) \equiv$ edge connectivity), an algorithm that solves the connectivity problems will be shown:

ALGORITHM (For this algorithm I am using edge-connectivity).

INPUT

A connected non-trivial graph $G = (V, E)$.

OUTPUT

Value of $\lambda(G)$ – edge connectivity

1º. Select a spanning tree T of G, and let Y be the set of all non-leaves of T.

2º. Select an arbitrary vertex $v \in Y$, and let $X = Y - \{v\}$

3º. Compute $\lambda(v,w)$ for every $w \in X$

represents the least number of arcs whose deletion would destroy every directed path from v to w.

Steps to compute $\lambda(v,w)$

- If G is a graph, replace each edge xy with arcs (x,y) and (y,x).
- Assign v as the source vertex and w as the sink vertex.
- Assign the capacity of each arc to 1, and call the resulting network H.
- Find a max-flow function f in H.
- Set $\lambda(v,w)$ equal to the total flow of f. STOP

4º. Assign $c = \min\{\lambda(v,w) \mid w \in X\}$.

5º. Assign $\lambda(G) = \min\{c, \delta(G)\}$. STOP.

Problem 4

a) Considering a set of n elements, we know from the topics covered in class that the number of subsets is 2^n. In addition, half of the total number of subsets would be odd and the other half would be even. Then, there are a total of $\frac{2^n}{2} = 2^{n-1}$ odd sets in n elements.

The number of sets of size 1 in n elements is n.

Therefore, the number of sets of size 3 or higher in a set of n elements would be equal to $\underbrace{2^{n-1}}_{\text{number of odd sets.}} - \underbrace{n}_{\text{number of sets of size 1}}$

I have proved that for matching in general graphs, the number of odd sets ≥ 3 is $N = 2^{n-1} - n$

b) Construct the dual of the minimum cost matching problem in general graphs specified as:

$$\min \; c^T x$$
$$\text{s.t} \; \sum_j x_{ij} = 1$$
$$\sum_{i,j \in S_k} x_{ij} + \gamma_k = s_k \;, \quad \forall S_k \in \mathcal{O}$$
$$x_{ij} \geq 0 \;\; \forall i,j \;; \quad \gamma_k \geq 0 \;;$$

where $\mathcal{O}$ is the collection of all odd sets and the size of $S_k = 2s_k+$

Let's follow the steps learned in class.

I am going to use the Lagrangian approach to design the dual of the minimum cost matching problem in general graphs.

$$L(x, \gamma, \alpha, \beta, \mu, \lambda) = \sum_{ij} c_{ij} x_{ij} + \sum_i \alpha_i (1 - \sum_j x_{ij}) +$$

$$+ \sum_k \beta_k (S_k - \sum_{ij} x_{ij} - \gamma_k) + \sum_{ij} \mu_{ij} (-x_{ij}) + \sum_k \lambda_k (-\gamma_k)$$

$$\Phi(\alpha, \beta, \mu, \lambda) = \inf_{x,\gamma} \{ L(x, \gamma, \alpha, \beta, \mu, \lambda) \}$$

- Infimum related to "x":
  - $\inf_x \{ L(x, \gamma, \alpha, \beta, \mu, \lambda) \}$

  The infimum related to x is finite only if

  $$c_{ij} - \alpha_i - \alpha_j - \sum_k \beta_k - \mu_{ij} = 0 \; (*)$$

  If $(*) \neq 0 \longrightarrow -\infty \Rightarrow$ we don't want this.

- Infimum related to "γ":
  - $\inf_\gamma \{ L(x, \gamma, \alpha, \beta, \mu, \lambda) \}$

  The infimum related to γ is finite only if

  $$-\beta_k - \lambda_k = 0 \; (**)$$

  If $(**) \neq 0 \longrightarrow -\infty \Rightarrow$ we don't want this.

Then, assuming $c_{ij} - \alpha_i - \alpha_j - \sum_k \beta_k - \mu_{ij} = 0$

and $-\beta_k - \lambda_k = 0 \Rightarrow \Phi(\alpha, \beta, \mu, \lambda) = \sum_i \alpha_i + \sum_k \beta_k S_k$

$\boxed{\text{DLP}}$

$$\max \quad \sum_i \alpha_i + \sum_k \beta_k S_k$$

s.t

$$\alpha_i + \alpha_j + \sum_k \beta_k + \mu_{ij} = c_{ij}$$

$$\beta_k = -\lambda_k$$

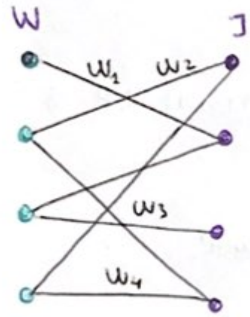
$$\Big\Downarrow \begin{matrix} \mu_{ij} \geq 0 \\ \lambda_k \geq 0 \end{matrix}$$

$$\boxed{\begin{array}{lll} \max & \sum_i \alpha_i + \sum_k \beta_k S_k & \\ \text{s.t} & \alpha_i + \alpha_j + \sum_k \beta_k \leq c_{ij} & \forall i,j \\ & \beta_k \leq 0 & \forall k \end{array}}$$

Problem 5

Let's talk first about what a weighted bipartite matching problem is and its relation to a polytope.

Given a bipartite graph $G = (V, E)$, with $V = W \cup J$, and weights w on edges e:



A matching is a set of edges covering each node at most once.

$n = |V|$; $m = |E|$

In this case, we are focused on using polyhedral interpretation.

<u>Linear programme</u>

$$\begin{aligned} \min \quad & w^T x \\ \text{s.t} \quad & \sum_j x_{ij} = 1 \quad \forall i \\ & \sum_i x_{ij} = 1 \quad \forall j \\ & x_{ij} \geq 0 \end{aligned} \quad \equiv \quad \begin{aligned} \min \quad & w^T x \\ \text{s.t} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

The feasible region of the matching LP is the convex hull of indicator vectors of matchings.

$F = \text{convex hull} \{ X_M : M \text{ is a matching} \}$

- At this point, let's prove that the polytope corresponding to the weighted bipartite matching problem has integral vertices, using the total unimodularity concept learned in class.

Definition of total unimodularity

A matrix $A \in \mathbb{Z}^{n \times n}$ is called totally unimodular if every square submatrix has determinant equal to 0, +1 or -1.

Claim:

The constraint matrix of the bipartite matching LP is totally unimodular.

Proof:

- $A_{ve} = 1$ if e incident on v, and 0 otherwise.
- Using induction on size of submatrix A':
  - Trivial case $k = 1$

    If A' has all zero column, then $\det A' = 0$.

    If A' has column with single 1, then holds by induction.

    If all columns of A' have two 1's.

    - Partition rows (vertices) into W and J.
    - Sum of rows W is $(1, 1, \dots, 1)$, similarly for J.
    - A' is singular, so $\det A' = 0$.

Therefore, for min cost bipartite matching $\left\{ \begin{array}{c} \min\ w^T x \\ Ax = b \\ x \geq 0 \end{array} \right\}$

⟶ The matrix A is totally unimodular.

Using the fact that all solutions of this linear program are basic feasible solutions (vertices). These solutions are of the form:

B is a submatrix of A.

$$\boxed{x = B^{-1} b}$$

Hence, as A is totally unimodular, $B^{-1}b = \left(\frac{1}{\det(B)} B^{adj}\right) b$

and all entries of B^{-1} are $0, \pm 1$.

$\boxed{x \text{ is integral}}$