Contents

1 Stable Matching

The stable matching solves the problem of assigning one element $x \in X$ to another element $y \in Y$(for example: men to wimen, intern to hospitals, ecc...).

- Unstable Pairs
  A pair p(x,y) is defined as unstable if $\exists y^* \in Y \; such \; that \; p(x, y^*) > p(x, y) \; and \; vice - versa.$
- Stable Matching
  A matching containing no unstable pairs.

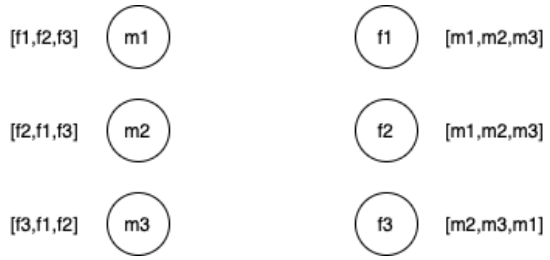


Figure 1: An instance of the problem containing the M and W sets and $\forall m \in M,\ \forall w \in W$ a list of preferences for the elements in the opposite set, in increasing order.

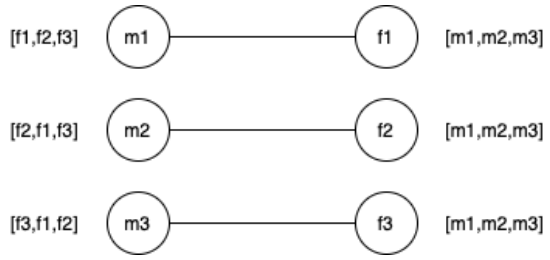


Figure 2: A solution for the instance of the problem, containing only stable pairs.

1.1 The Men-Wimen problem

The problem about finding the stable matching between a set W and M. Each $w \in W$ rank every $m \in M$ from best to worst, and every $m \in M$ do the same.

Algorithm 1: proposeAndReject(M, W) :

Input: M set of men, W set of women
Output: S the set containing all the stable pairs between M and W

$S \leftarrow$ set containing all the pairs between M and W.

while $\exists m \in M$ *that is free and haven't proposed yet* **do**
  $w_i = m[0] \leftarrow$ first woman in m's list of preference to whom he is not yet proposed ;
  if w_i *is free* **then**
    $S = S \cup p(m, w_i) \leftarrow$ create a pair between m and w_i;
  end
  if w_i *prefers* m *to her current partner* m_i **then**
    $S = S - p(m_i, w_i)$;
    $S = S \cup p(m, w_i)$;
  end
end
return S;

- Observation 1
  Every men propose to a women decreasing order of preference, for every women is the opposite.
- Observation 2
  Once a women is matched she never become un-matched, she just change her partner, in increasing order of preference.
- Observation 3
  Assuming $|M| = |W|$ then the algorithm has $\mathcal{O}(n^2)$ complexity.
- Observation 4
  The algorithm is very resistant to modifications: example new conditions on engagements.

2 Asymptotic Order of Growth

2.1 Brute Force

For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution, usually taking 2^n.

2.2 Polynomial Running Time

This is the most desirable running time of any algorithm because it scales with ease at the increment of the input size.
An algorithm can be defined as poly-time if the following property holds:

$$\exists c > 0, \exists d > 0, \textit{ such that } \forall n \in INPUT \textit{ the running time is bounded by } cn^d$$

We can now give the following definition of efficiency:

$$\textit{An algorithm is efficient if it is poly} - \textit{time.}$$

When discussing the running time of a deterministic algorithm is important to take into account the worst-case scenario because, in doing that, the running time is guaranteed $\forall n \in INPUT$. There are some exception to this rule, when the worst-case scenario is very rare.

When discussing of probabilistic algorithm we need, instead, to reason upon the expected running time.

2.3 Big-O Notation

Assuming we have an algorithm rappresented by the function:

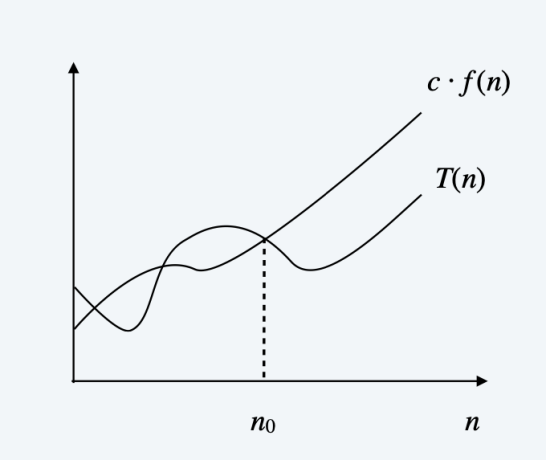
$$T(n) = 1.6n^2 + 3.5n + 8$$

We want a way to express the growing rate in a way that is insensitive to constant factors and low-ordered terms, in fact It can be said that T(n) grows like $\mathcal{O}(n^2)$.

- The lower bound $\mathcal{O}$
  We can say that T(n) is $\mathcal{O}(f(n))$ if $\exists c > 0, \exists n_0 \geq 0 / \ T(n) \leq cf(n), \ \forall n \geq n_0$
  We can give an alternative definition as follow:

$$\mathcal{O}(f(n)) = \lim_{n \to \infty} \frac{T(n)}{F(n)} < \infty$$

  Graphically:



Again, going back to our function: $T(n) = 1.6n^2 + 3.5n + 8$,

$$T(n) = n^3$$

$$T(n) = n^2$$

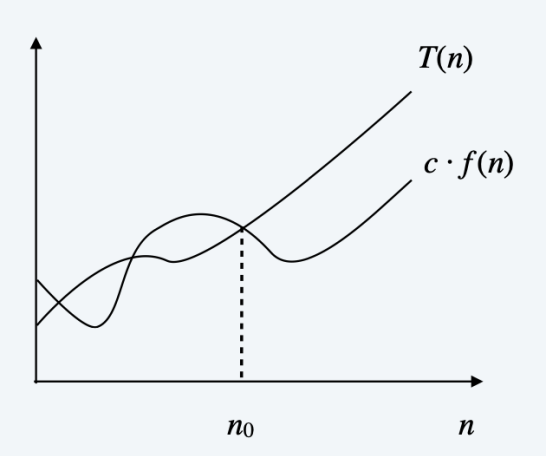
$$T(n) \neq n$$

- The upper bound Ω
  We can say that T(n) is $\Omega(f(n))$ $if \exists c > 0, \exists n_0 \geq 0 / \ T(n) \geq cf(n), \ \forall n \geq n_0$
  Again, going back to our function: $T(n) = 1.6n^2 + 3.5n + 8$,

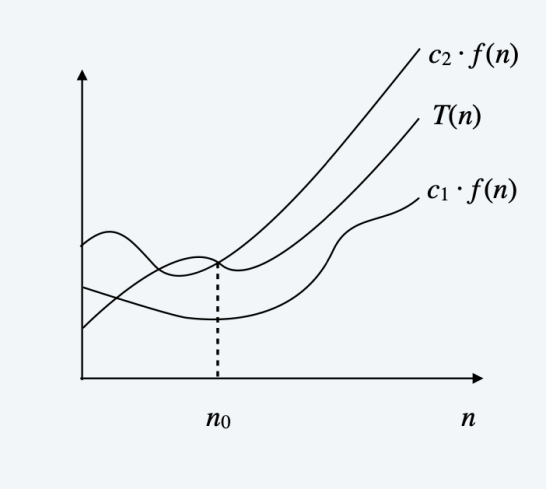
$$T(n) = \Omega(n)$$

since $T(n) \leq pn^2 \leq pn$.

Graphically:



- The tight bound Θ
  T(n) is $\Theta(f(n))$ if $\exists c_1 > 0, c_2 > 0$ *and* $\nexists$ n_0 *such that* $c_1 f(n) \leq T(n) \leq c_2 f(n)$ $\forall n \geq n_0$
  Graphically:



- Usefull Properties

  Transitivity:

  $$if\ f = \mathcal{O}(g)\ and\ g = \mathcal{O}(h),\ then\ f = \mathcal{O}(h)$$
  $$if\ f = \Omega(g)\ and\ g = \Omega(h),\ then\ f = \Omega(h)$$

  Sum of functions: Supposing that f and g are two functions such that, for some other function h, we have $f = \mathcal{O}(h)\ and\ g = \mathcal{O}(h),\ then\ f + g = \mathcal{O}(h)$.

2.4 Common Running Times

- Linear $\mathcal{O}(n)$

  The running time is proportional to the input size.

  Example: finding the maximum into an array.

- Linearithmic time $\mathcal{O}(nlogn)$

  Very common in divide-and-conquer algorithms, is also the running time of most of the sorting algorithms (ex: MergeSort or HeapSort).

  Note: if an algorithm A execute n times a sorting operation, let's say n=3, and the sorting operation is still the most computational costly operation in the whole algorithm: then A is still $\mathcal{O}(nlogn)$ because $3\mathcal{O}(nlogn) \simeq \mathcal{O}(nlogn)$.

- Quadratic time $\mathcal{O}(n^2)$

  Common when enumerating all pairs of elements, ex: given a list of n points in the plane (x_1, y_1), ..., (x_n, y_n), find the pair that is closest to each other.

- Cubic time $\mathcal{O}(n^3)$

  Example:Set disjointness. Given n sets: $S_1, ..., S_n$ each of which is a subset of 1, 2, ..., n, is there some pair of these which are disjoint?

- Ploynomial time $\mathcal{O}(n^k)$

  Example:Independent set of size k: Given a graph, are there k nodes such that no two are joined by an edge? The solution is to enumerate all subsets of k nodes.

- Exponential time

  Given a graph, what is maximum cardinality of an independent set? Enumerate all the possible subsets $\Rightarrow \mathcal{O}(n^2 2n)$

- Sublinear time

  Search in a sorted array. Given a sorted array A of n numbers, is a given number x in the array? $\Rightarrow \mathcal{O}(logn)$ Binary search.

2.5 Why it Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Figure 3: Comparison of the running times of different complexity algorithms.

Notice that, as we distance from the linear complexity, for bigger value of $n \in INPUT$, the execution time goes very high, this is bad: let's just say we want to deploy our non-linear algorithm on a distributed system dealing with a lots of records, as they grows, the system could become unable to process them in a feasible time.

2.6 The Heap Data Structure

Is a balanced binary tree T(V,E) that satisfy the following property, defining with C the set containing all the children of a node w_i

$$\forall w_i \in V, \forall v_i \in C,\ key(w_i) \leq key(v_i)$$

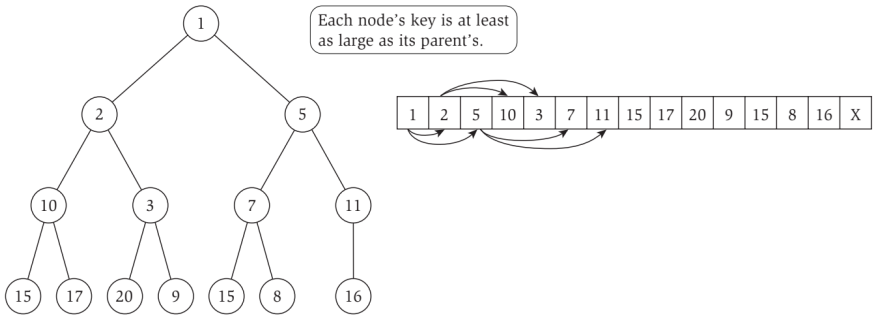


Figure 4: An example of heap.

This data structure is very helpfull, instead of using an array or a list, here are the complexity of some operations involving the heap structure:

$$create\ heap = \mathcal{O}(n)$$
$$insert\ heap = \mathcal{O}(logn)$$
$$find\ min = \mathcal{O}(1)$$
$$delete = \mathcal{O}(logn)$$

3 Greedy Algorithms

Formal definition: the greedy algorithm "stays ahead" locally: meaning that is better than any other solutions, alternatively we can prove that a greedy algorithm's solution is optimal by transforming the know optimal solution for the problem, into our solution, this method is called "argument exchange".

Practical definition: A lot of the time a greedy algorithms involves scanning all the $n \in INPUT$ of the problem, if the current element respect some conditions, then it can be added to the optimal solution returned by the algorithm.

3.1 The Cashier Problem / Coin changing

Given a currency with the following values: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Example: 34 dollars, how many coins?

Solution: at each iteration, add coin of the largest value that does not take us past the amount to be paid.

Algorithm 2: cashierAlgorithm(amount,C):

Input: *amount* containing the amount to give back to the client, C set containing all the coins of the current currency.
Output: S the set containing all the coins to give back to the client

$S \leftarrow$ set containing all the coins to give back to the client;
$total = 0 \leftarrow$ total expressed in the current currency system;
$Cs \leftarrow$ set containing all the coins in the currency sorted by increasing values;

```
while total < amount do
    for i=Cs.length to 0 do
        if total + Cs[i] < amount then
            total = total + Cs[i];
            S = S ∪ Cs[i];
            break;
        else if total + Cs[i] = amount then
            S = S ∪ Cs[i];
            return S;
    end
end
return S;
```

Claim The cashier algorithm always returns an optimal solution.

Proof The aim of the algorithm is to give as little change as possible (in terms of number of coins), if this is not true, then it must exist a solution $|S^*| < |S|$, but the algorithm always give highest compatible coin in the for loop, therefore a contradiction. □

3.2 Interval Scheduling

We have a set J of jobs to execute on a machine and $\forall j \in J\ j = (s_j, f_j)$ where s and f are start and finish.Two jobs are compatible if the don't overlap: $\forall j_i, j_j \in C\ f_i \geq s_j\ or\ vice - versa$.
Once we defined the problem, is just the matter of identifying the best strategy to sort all the jobs in J. Turns out, the best strategy is to sort all the jobs by finishing time, from first to last (for a former proof please check the professor's slide, but the main idea is to prioritize the jobs that release the machine as soon

as possible).

Algorithm 3: earliestFinishingTime(J):

Input: J set containing all the jobs
Output: S the set containing all the compatible jobs

$S \leftarrow$ set containing all the compatible jobs;
$Js \leftarrow$ set containing all the sorted jobs in increasing order of finishing time $f1 < ....f_j$

for *i=0 to Js.lenght* **do**
  if $J[i].start \geq S.last.finish$ **then**
    $S = S \cup Js[i]$
  end
end

return S;

Claim The earliestFinishingTime always return an optimal solution.
Proof Assume this is not true, then it must exist a solution $|S^*| > |S|$,containing at least one more compatible job, but the algorithm check every job in J, therefore a contradiction. □

Claim The complexity of the algorithm is $\mathcal{O}(nlogn)$
Proof The for loop costs $\mathcal{O}(n)$, while, as stated before, the sorting costs $\mathcal{O}(nlogn)$, since $\mathcal{O}(nlogn) > \mathcal{O}(n)$, the whole algorithm is $\mathcal{O}(nlogn)$. □

3.3 Interval Partitioning

Assuming we have a set of lectures L and, $\forall l \in L,\ l = (s, f)$ where s and f are start time and finishing time, find the minimum amount of classroom tho schedule all the lectures, so that no two lectures overlap in the same classroom (the definition of overlap is the same as 3.2).

Algorithm 4: intervalPartitioning(L):

Input: L set containing all the lectures
Output: S set containing the minimum amount of classroom needed

$S \leftarrow$ set containing the minimum amount of classroom needed;
$Ls \leftarrow$ set containing all the sorted lectures in increasing order of starting time $s1 < ....s_j$

for *i=0 to Ls.lenght* **do**
  $C = \varnothing$
  if $J[i].start \geq C.last.finish$ **then**
    $C = C \cup Ls[i]$;
  else
    $S = S \cup C$;
    $C = \varnothing$;
    $C = C \cup Ls[i]$;
  end
end

return S;

The number of classrooms needed is more or equal to the depth of $|S|$, that is also the maximum amount of items that contains at any given time.

Claim The intervalPartitioning always return an optimal solution.
Proof Same as 3.2, if it exist a better solution S^* we would have $|S^*| > |S|$ but that is not possible, since the algorithm checks every lecture in the input set. □

Claim The complexity of the algorithm is $\mathcal{O}(nlogn)$

Proof Identical as 3.2. □

3.4 Minimizing The Lateness

Assuming now we have just one resource that can process just one job at a time, we want to process all possible jobs. For the notation: $\forall j \in J$, j requires t_j amount of time to be executed, with a finishing time $f_j = s_j + t_j$, we have also to define a due date d_j and finally the lateness of a job as: $max(0, d_j - f_j)$

Algorithm 5: earliestDeadLineFirst(J):

Input: L set containing all the lectures
Output: S set containing the minimum amount of classroom needed

$S \leftarrow$ set containing all the compatible jobs;
$Js \leftarrow$ set of all jobs sorted by increasing deadline $s1 < ....d_j$

$t = 0$;
for *i=0 to Js.lenght* **do**
    $s_i = t$;
    $f_i = s_i + Js[i].finish$;
    $S = S \cup (s_i, f_i)$;
    $t = t + Js[i].time$;
end
return S;

Claim EarliestDeadLineFirst algorithm return an optimal solution.
Proof Identical as 3.2 / 3.3. □

Claim The complexity of the algorithm is $\mathcal{O}(nlogn)$
Proof Identical as 3.2. □

3.5 Dikstra Algorithm

Def: A path is a sequence of edges which connects a sequence of nodes.

Def: A cycle is a path with no repeated nodes or edges other than the starting and ending nodes.

Shortest Path problem: Given a digraph G = (V, E), edge lengths $le \geq 0$, source $s \in V$, and destination $t \in V$, find the shortest directed path from s to t.

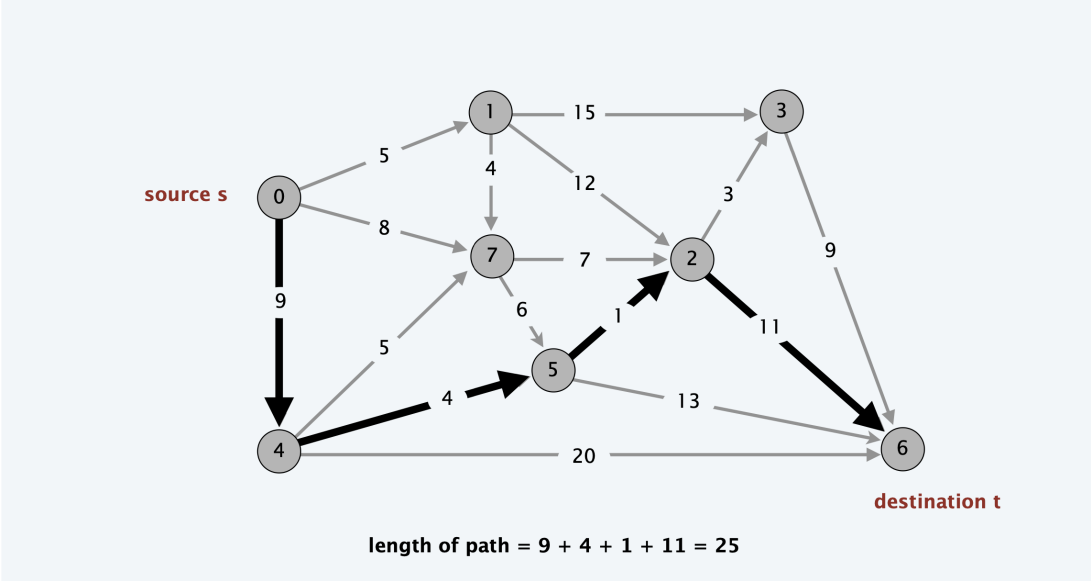


Figure 5: A solution for the Shortest path problem.

Algorithm 6: Dikstra(G,s):

Input: G set containing all the edges of G, s source of the path
Output: S set containing all the edges of a shortest path

$S \leftarrow$ set containing all the compatible jobs;
$Q \leftarrow$ priority queue for nodes contained in the unexplored part of G ;
$D \leftarrow$ set of distances from s $\forall\ e \in G$

$d(s,s) = 0$;
$D = D \cup d(s,s))$;

for *i=0 to G.lenght* **do**
  $e_i = G[i] \leftarrow$ current edge to check;
  if $e_i \neq s$ **then**
    $d(s, e_i) = \infty$;
    $D = D \cup d(s, e_i))$;
    $Q = Q \cup (e_i, d(s, e_i)) \leftarrow$ Insert in queue the current node with key$(e_i) = d(s, e_i)$;
end

while $Q \neq \emptyset$ **do**
  $u = Q.getMin$;
  $Q = Q - \{u\}$;
  foreach $(u, v) \in G$, *leaving* u **do**
    if $d(s, v) > d(s, u) + l(u, v)$ **then**
      $S = S \cup d(s, u) + l(u, v)$;
      decrease-key of v to d(u) + l(u, v) in Q.;
  end
end
return S;

Claim The Dikstra Algorithm always return an optimal solution ($\forall u \in S,\ d(u)$ is the length of the shortest $s \to u$ path).

Proof Base case: $|S| = 1$ is easy since S = s and d(s) = 0, assume that's true for $|S| > 1$:
let t be next node added to S, and let (u, t) be the final edge: the shortest $s \to u$ path + (u, t) = $s \to t$ path of length $\pi(t)$.

Now consider any $s \to t$ path P: it's is no shorter than $\pi(t)$. Let (x, y) be the first edge in P that leaves S, and let P' be the subpath to x, P is already too long as soon as it reaches y, so putting all together:

$$l(P) \geq l(P') + l(x, y) \geq d(x) + l(x, y) \geq \pi(y) \geq \pi(t)$$

□

Claim The Dikstra Algorithm complexity, for a graph G=(V,E), using a priority queue is: $\mathcal{O}((V + E)\ log\ V)$.

Proof It highly depends on the type of queue used, but we have to check all the nodes in the priority queue plus all the edges coming out of every node, for further detail, please check the professor's slides. □

3.6 Kruskal Algorithm

Given an undirected, weighted, connected graph G=(V,E) we want to find the all the edges that connects all the nodes with a minimum cost, alternatively we can say that the algorithm finds the minimum spanning tree (MST) of G.

Formal definition: Given an undirected, weighted, connected graph G = (V, E) with edge costs $c(e)$, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge costs is minimized.

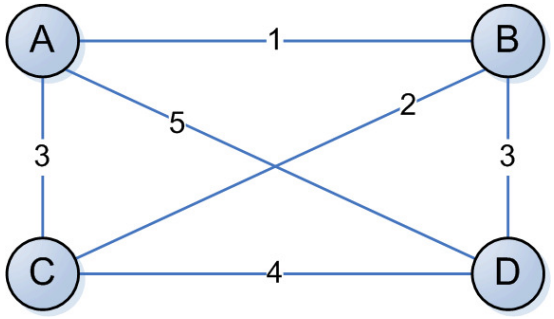


Figure 6: An instance of the MST problem.

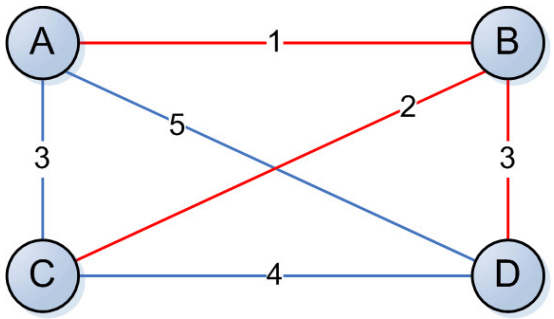


Figure 7: A solution for the MST problem.

Algorithm 7: kruskal(G):

Input: G set containing all the edges of G
Output: S set containing all the edges of MST

$S \leftarrow$ set containing all the compatible jobs;
$E \leftarrow$ set of all the edges sorted by increasing weight $c(e)_1 < ....c(e)_j$

for *i=0 to G.lenght* **do**
  $e_i = G[i] \leftarrow$ current edge to check;
  if $S \cup e_i$ *is not a loop* **then**
    $S = S \cup e_i$;
end

return S;

Claim Kruskal algorithm returns an optimal solution.
Proof Identical as 3.2 / 3.3. □

Claim The complexity of the algorithm is $\mathcal{O}(nlogn)$
Proof Identical as 3.2. □

4 Dynamic Programming

Formal definition: Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

Practical definition: A lot of the times dynamic programming involves taking the best possible result of the subproblem (that maximizes the overall results), and store it in a data structure so it should not be recomputed later: this process is called memoization.For an algorithm dealing with dynamic programming usually there are 2 possible approaches:

- Top-down

  The most common approach: as we said we divide the problem, into sub-problems and recursively memorize the result in a table (again using the memoization tecnique), whenever attempting to solve the sub-problems, we first check if we already computed the result in the table, saving precious execution time.

- Bottom-up

  We formulate the solution to a problem recursively by solving the subproblems first, and then using their solutions to formulate a larger overall solution.

4.1 Weighted Interval Scheduling

Same as 3.2 but this time $\forall j \in J$, there is an associated weight $w(j) > 0$, in this case, unfortunately, the greedy algorithm fails.
To find a possible solution, we start to label jobs by finishing time $f1 \leq f2 \leq ... \leq f_n$ and we donote with p_j = the largest index $i < j$ such that job i is compatible with j:

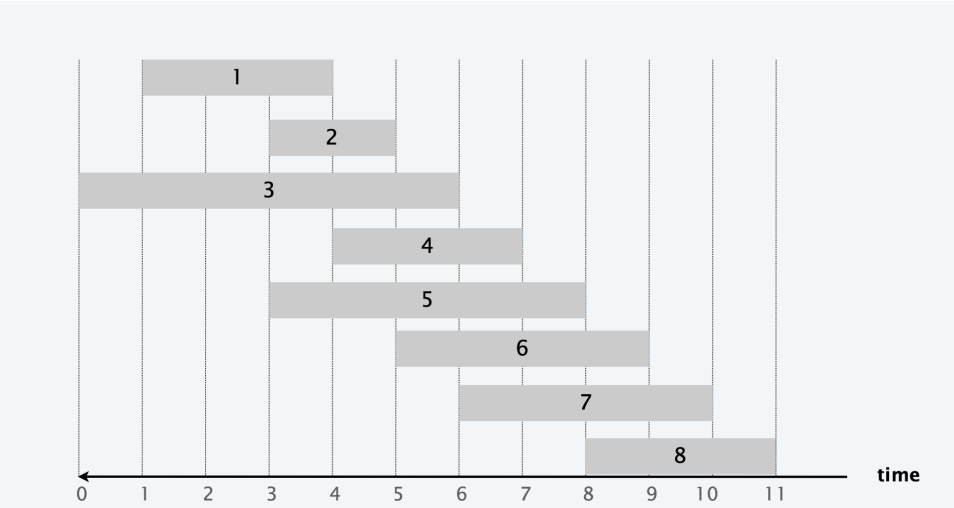


Figure 8: An instance for the weighted scheduling problem.

Example p(8) = 5 because the earliest compatible $j \in J$ with j_8 is j_5.More examples: p(7) = 3, p(2) = 0, ecc...

Now let's denote the OPT(j) as the value of optimal solution to the problem consisting, of job requests $1, 2, ..., j$. When dealing with dynamic programming, we have to build a formal definition of the recursion

function, that defines the optimum at every step, but first we have to understand what happen at each iteration of the algorithm, the optimum can either:

- OPT selects job j

  The algorithm then collect the profit v_j, then is locked out by all the incompatible jobs $\{p(j) + 1, p(j) + 2, ..., j\text{–}1\}$, but must contain all the remain compatible jobs $\{1, 2, ..., p(j)\}$.

- OPT does not select job j

  Must include optimal solution to problem consisting of remaining compatible jobs $\{1, 2, ..., j\text{–}1\}$.

  Merging all these informations we can build the recursion function:

$$OPT(j) = \begin{cases} 0 \; if \; j = 0 \\ max\{vj \; + \; OPT(p(j)), OPT(j-1)\} \end{cases}$$

Now for the implementation:

Algorithm 8: DynamicWeightedInterval(J):

Input: J set of jobs, each one with a s_j start time, a finishing time f_j and a reward v_j
Output: S containing the largest amount of compatible weighted jobs

$Js \leftarrow$ all the jobs sorted by increasing finishing time $f_1 \leq f_2 ... f_j$

for *i=0 to J.lenght* **do**
  if $i = 0$ **then**
    $M[i] = 0$;
  else
    $M[i] = \emptyset$;
end

$costMatrix = MComputeOpt(M, Js, J.last)$;
$return findSolution(M, currentJob)$

Algorithm 9: MComputeOpt(M,J,currentJob):

Input: M Initialized Matrix containing all the reward,;
J set of jobs, $currentJob$ to calculate the reward.
Output: M matrix containing all the computed values for each job

if $M[currentJob] = \emptyset$ **then**
  $M[currentJob] =$
  $max(currentJob.vj + MComputeOpt(M, J, currentJob.pj), MComputeOpt(M, J, J[currentJob\text{–}1]))$;
else
  return $M[currentJob]$;

Algorithm 10: findSolution(M,currentJob):

Input: M Initialized Matrix containing all the reward,;
$currentJob$ to calculate the reward.
Output: S containing the largest amount of compatible weighted jobs

if $currentJob = 0$ **then**
  return $\emptyset$;
else if $currentJob.v_j + M[currentJob.p_j] > M[currentJob\text{–}1]$ **then**
  $return\{j\} \cup FindSolution(M, currentJob.p_j)$.
else
  $FindSolution(M, currentJob - 1)$.

Claim The algorithm runs in $\mathcal{O}(nlogn)$

Proof The recursive processes takes $\mathcal{O}(n)$ each, the most computational intensive operation is the sorting that takes $\mathcal{O}(nlogn)$, so overall we have $\mathcal{O}(2n + nlogn) \simeq \mathcal{O}(nlogn)$. □

4.2 Knapsack Problem

Given N objects and a Knapsack(it's a backpack) with a capacity W, we have $\forall n \in N,\ w_n \geq 0\ v_n \geq 0$, the goal is to fillup the Knapsack without exceeding it's capacity with the maximum value.

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance (weight limit W = 11)

Figure 9: An instance for the knapsack problem.

Example: $\{1, 2, 5\}$ has value 35, $\{3, 4\}$ has value 40, $\{3, 5\}$ has value 46 (but exceeds weight limit).

- Defining the Optimum

  This time we have to take into account both the variation of the value of the content of the knapsack and the weight limit left:

$$OPT(j) = \begin{cases} 0\ if\ i = 0 \\ OPT(i-1, w)\ if\ w_i > w \\ max\{OPT(i-1, w),\ vi + OPT(i-1, w - w_i)\}\ otherwise\} \end{cases}$$

Now for the implementation:

Algorithm 11: DynamicKnapsack(N,W):

Input: N set of all the objects, each one with a v_j value, and w_i weight, W the maximum capacity of the knapsack
Output: S containing the most valuable objects without exceeding w

for *i=0 to w* **do**
  $M[0, wi] = 0$;
end
for *j=1 to N* **do**
  for *k=1 to W* **do**
    if $N[j].weight > k$ **then**
      $M[j, k] = M[j–1, k]$
    else
      $M[j, k] = max\{M[j–1, k], vi + M[j–1, w–w_j]\}$.
  end
end
return $M[n, W]$

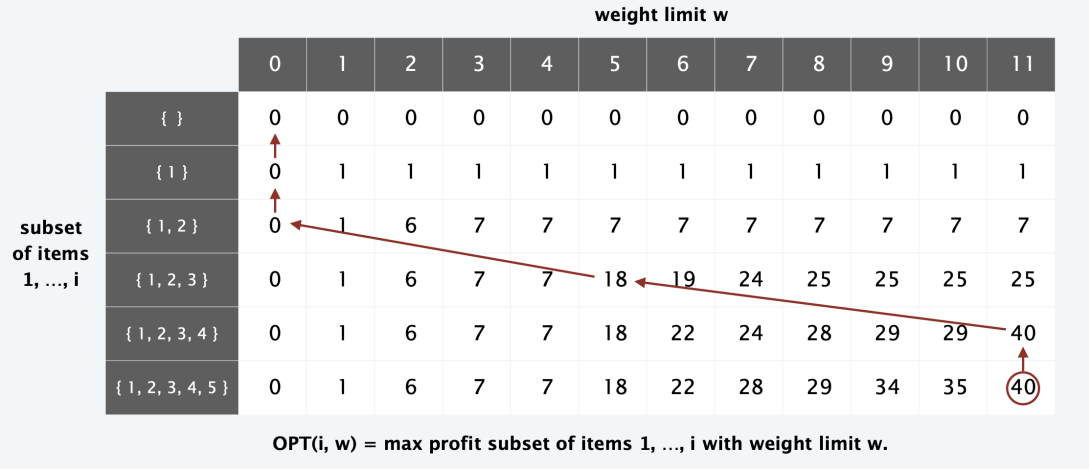


Figure 10: A table rapresenting the solution for the knapsack problem.

This algorithm is defined as pseudo-polinomial, because the execution time is proportional to the knapsack's W.

Claim The algorithm runs in $\Theta(nW)$

Proof We need to build a table of all the combinations of values and weight of dimension $\Theta(nW)$, $\forall n \in N$ so that's the execution time of the algorithm. □

4.3 Sequence Alignment

In this problem we want to measure the degree of similarity between two strings, first of all we have to make sure that the two strings have the same length, then we define a gap as $\exists i \in s_1/s_2[i] = \emptyset$ or vice-versa, instead a mismatch as: $\exists i \in s_1/s_2[i] \neq s_1[i]$.

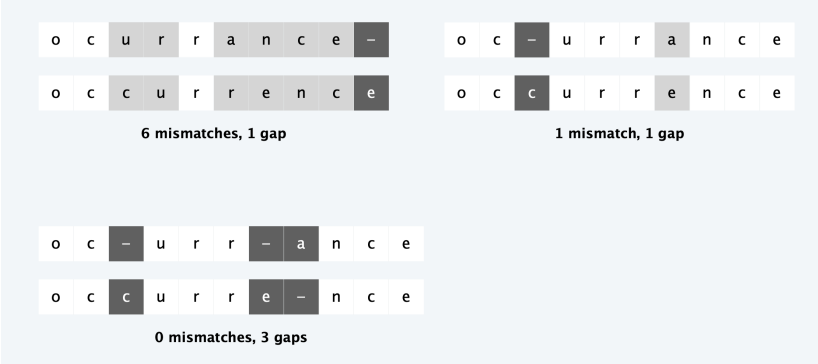


Figure 11: different approaches for the gap and occurrences in the sequence alignment problem.

To define a degree of similarities between two strings, we need a way to define a cost that rappresents the degree of difference between the strings, so for each gap we define a penalty δ and a mismatch penalty $\alpha_{p,q}$, so the total cost would be the sum of mismatch and gaps. Now we can re-formulate our problem as: given two strings $x_1, x_2 ... x_m$ and $y_1, y_2 ... y_m$, we want to find a minimum cost alignment defined as: a set of ordered pairs $x_i - y_i$ such that each item occurs in at most one pair and no crossings (x_i–y_j and $x_{i'}$–$y_{j'}$ cross if $i < i'$, but $j > j'$).

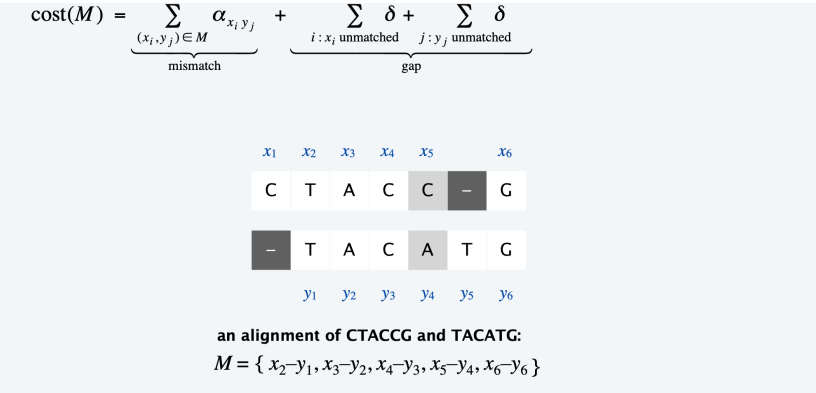


Figure 12: Cost Definition.

Now, as usual we have to define the optimum as OPT(i, j) = min cost of aligning prefix in the two strings.

- Case 1: matches x_i–y_j.
  Pay mismatch for x_i–y_j + min cost of aligning $x_1, x_2...x_{i-1}$ and $y_1, y_2...y_{j-1}$.
- Case 2:OPT leaves x_i unmatched.
  Pay gap for x_i + min cost of aligning $x_1, x_2...x_{i-1}$ and $y_1, y_2...y_j$.
- Case 3:OPT leaves y_i unmatched
  Pay gap for y_j + min cost of aligning $x_1, x_2...xi$ and $y_1, y_2...y_j$.

$$OPT(j) = \begin{cases} i\delta \ if \ i = 0 \\ min \ otherwise \\ j\delta \ j = 0\} \end{cases}$$

$$min = \begin{cases} \alpha_{x_i y_j} OPT(i-1, j-1) \\ \delta OPT(i-1, j) \\ \delta OPT(i, j-1) \end{cases}$$

Algorithm 12: SequenceAllignment(x,y):

Input: x String y String
Output: S containing the minimum cost alignment.

for *i=0 to x.lenght* **do**
  $M[i, 0] = \delta i$;
end

for *j=0 to y.lenght* **do**
  $M[j, 0] = \delta j$;
end

for *i=0 to x.lenght* **do**
  for *j=0 to y.lenght* **do**
    $M[i, j] = min(\alpha[x_i, y_j] + M[i–1, j–1], \delta + M[i–1, j], \delta + M[i, j–1])$;
  end
end

return S = M[x.lenght][y.lenght]

Claim The algorithm runs in Θ(mn) for two strings of length m and n.

Proof We need to build a table of all the combinations of values and weight of dimension Θ(mn), same as 4.2 □

4.4 Bellman-Ford

Algorithm for shortest path for graphs with negatives paths, unfortunately the Dikstra fails:

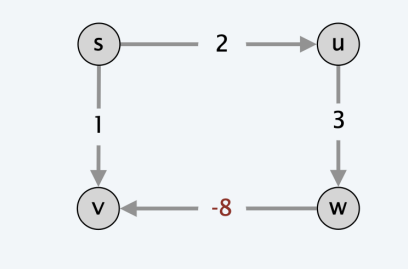


Figure 13: G=(V,E) with negative paths: the Dikstra algorithm will select c(s,v) = 1, ignoring the correct solution: c((s,u)+(u,w)+(w,v))=-3

Formal definition:A negative path is a directed cycle such that the sum of its edge weights is negative.

$$c(W) = \sum_{e \in W} c_e < 0$$

Claim If some path from v to t contains a negative cycle, then there does not exist a cheapest path from v to t.

Proof If there exists such a cycle W, then can build a $v \to t$ path of arbitrarily negative weight by detouring around cycle as many times as desired. □

Claim If G has no negative cycles, then there exists a cheapest path from v to t that is simple (and has $\leq$ n – 1 edges).

Proof Consider a cheapest $v \to t$ path P that uses the fewest number of edges, If P contains a cycle W, we can remove portion of P corresponding to W without increasing the cost. □

the OPT(i,v) = cost of shortest $v \to t$ path that uses i edges, with i $\leq |E|$.

- Case 1: Cheapest $v \to t$ path uses $\leq$ i – 1 edges.
  OPT(i, v) = OPT(i – 1, v)
- Case 2:Cheapest $v \to t$ path uses $\leq$ i edges.
  if (v, w) is first edge, then OPT uses (v, w), and then selects best $w \to t$ path using $\leq$ i – 1 edges.

$$OPT(i,v) = \begin{cases} \infty \; if = 0 \\ min \; otherwise \end{cases}$$

$$min = \{OPT(i-1,v), min_{(v,w) \in E}\{OPT(i-1,w) + c_{vw}\}\}$$

Implementation:

Algorithm 13: BellmanFord(G,s,t):

Input: G directed graph with negative paths;
s source node of the path, t final node of the path.
Output: S containing the shortest path s ← t

Claim The Bellman-Ford algorithm's cost is $\mathcal{O}(nm)$.

Proof The successor graph cannot have a negative cycle,thus, following the successor pointers from s yields a directed path to t.

Let $s = v_1 \leftarrow v_2 \leftarrow ... \leftarrow v_k = t$ be the nodes along this path P, upon termination, if successor(v) = w, we must have d(v) = d(w) + c_{vw}, putting it all together, we have:

$$d(s) = d(t) + c(v_1, v_2) + c(v_2, v_3) + ... + c(v_{k-1}, v_k)$$

□

5 Network Flow

A flow network is a grap G=(V,E) with source $s \in V$ and $t \in V$, with $\forall e \in E,\ c(e) \geq 0$.An $s \to t$ cut is a partition (A, B) of the vertices with $s \in A$ and $t \in B$, while the capacity is the sum of all the capacities of all the edges from A to B: cap(A,B) = $\sum_{e\ out\ of\ A} c_e$

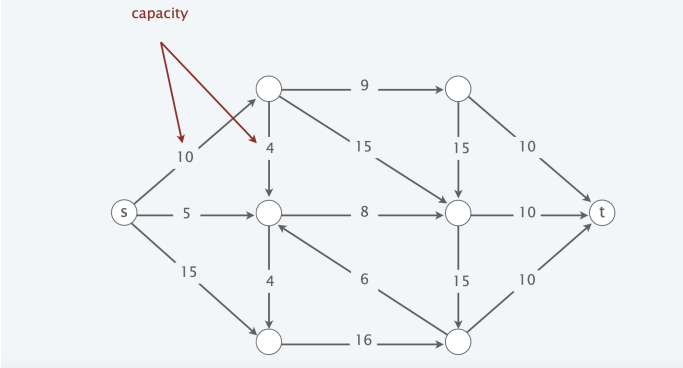


Figure 14: An instance of network flow.

5.1 The Maximum Flow Problem

An $s \to t$ is a function f that satisfies:

$$\forall e \in E:\ 0 \leq f(e) \leq c(e)\ (capacity)$$

$$\forall v \in V - \{s,t\}: \sum_{e\ out\ of\ v} f(e) = \sum_{e\ into\ v} f(e)\ (flow\ conservation)$$

The maximum flow problem, aim at maximizing the flow function, while respecting the constraint above.

5.2 The maximum flow/ minimum cut relationship

- Residual Graph

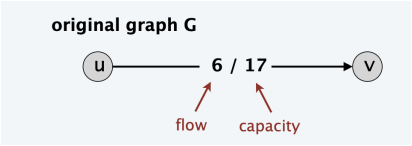


Figure 15: The orignal flow

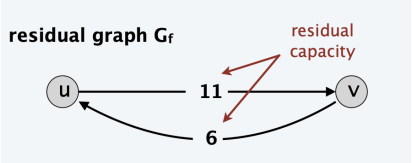


Figure 16: The residual edge.

The residual graph G is simply the graph resulting upon applying the residual transformation $\forall e \in E$

- Augmenting Path
  An augmenting path is a simple $s \leftarrow t$ path P in the residual graph G_f, The bottleneck capacity of an augmenting P is the minimum residual capacity of any edge in P.

- Flow value Lemma
  Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f:

$$\sum_{e\ out\ of\ A} f(e) = \sum_{e\ into\ A} f(e) = v(f)$$

  proof:

$$v(f) = \sum_{e\ outof\ s} f(e) = \sum_{v \in A} (\sum_{e\ outof\ v} f(e) - \sum_{e\ into\ v} f(e)) = (\sum_{e\ outof\ A} f(e) - \sum_{e\ into\ A} f(e))$$

- Weak Duality
  Let f be any flow and (A, B) be any cut. Then, v(f) $\leq$ cap(A, B).

  proof:

$$v(f) = \sum_{e\ outof\ A} f(e) - \sum_{e\ into\ A} f(e) \leq \sum_{e\ outof\ A} f(e) \leq \sum_{e\ outof\ A} c(e) = cap(A, B)$$

- Integrality Theorem
  If c(e) $\forall e \in E$ are integers, then there exists a max flow f for which every flow value f(e) is an integer.

Claim The value of the maximum flow equals to the minimum cut, furthermore, when the flow is maximized there are no augmenting path left.

Proof If there exists a cut (A, B) such that cap(A, B) = val(f),there is no augmenting path with respect to f, so f is a max-flow. □

5.3 Ford-Fulkerson Algorithm

It's a simple greedy approach algorithm: basically we start with a f(e) = 0 $\forall e \in E$, then we will find all the augmenting path in G(V,E) until there are no more left, the result is both a maximum flow and an min-cut.

Algorithm 14: FordFulkerson(G,s,t):

Input: G directed graph ;
s source node of the flow, t sink of the flow.
Output: S containing all the paths that maxes the maximum flow. $\leftarrow$ t

Claim The Ford-Fulkerson complexity is $\mathcal{O}(|E|f^*)$

Proof the algorithm terminates only when there are no more augmenting paths on G and by definition there is no more flow in the residual net f^*. Since the algorithm needs to check all the augmenting paths until there are no more left, the execution is closely releated on how many augmenting paths are in G. □

5.4 Matching and Bipartite Matching

Formal definition: Given an undirected graph G = (V, E) a subset of edges $M \subseteq E$ is a matching if each node appears in at most one edge in M, the maximum matching is a max cardinality matching.

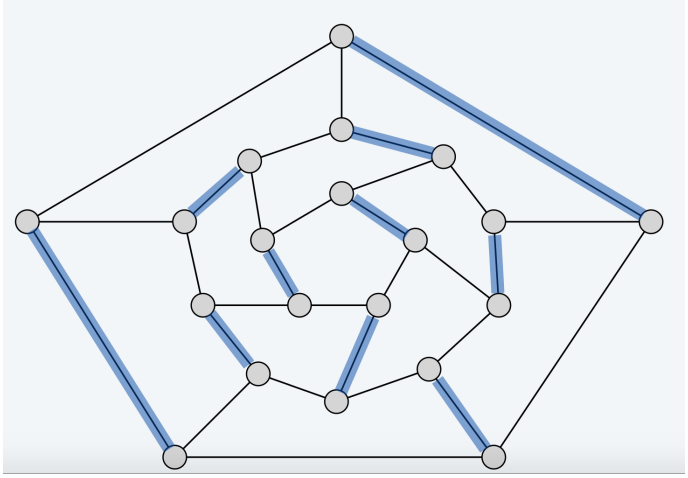


Figure 17: The maximum matching in a Graph G

Formal definition: A graph G is bipartite if the nodes can be partitioned into two subsets L (Left) and R (Right) such that every edge connects a node in L to one in R, a bipartite matching G = (L ∪ R, E), find a max cardinality matching.

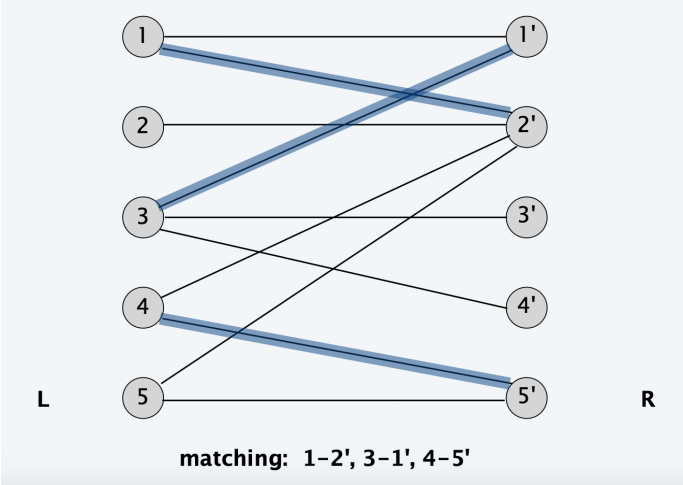


Figure 18: The maximum bipartite matching in a Graph G

Hall Theorem: Let S be a subset of nodes, and let N(S) be the set of nodes adjacent to nodes in S, Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. G has a perfect matching iff $|N(S)| \geq |S|$ for all subsets $S \subseteq L$.

Proof: Suppose G does not have a perfect matching: formulate a max flow problem and let (A, B) be min cut in G', by max-flow min-cut theorem: $cap(A, B) < |L|$.
Define LA=$L \cap A, LB = L \cap B, RA = R \cap A$, so the cap(A, B) = $|LB| + |RA|$. Since min cut can't use ∞ edges: $N(LA) \subseteq RA$ *with* $|N(LA)| \leq |RA| = cap(A, B)$–$|LB| < |L|$–$|LB| = |LA|$ so S=LA.

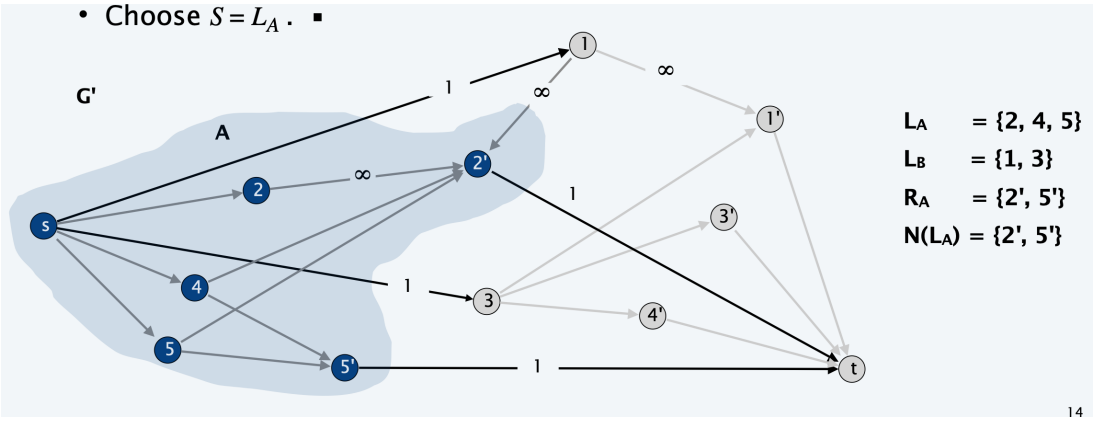


Figure 19

Claim The max cardinality of a matching in G = value of max flow in G'.

Proof First we need to build $G_{\prime} = (L \cup R \cup \{s, t\}, E^{'})$, direct all edges from L to R and give them unit capacity, then connect s to all $e \in L$ and t to all $e \in R$ with edges with infinite capacity, using the integrality theorem we know that the flow can assume only value 0-1. Finally consider the set of edges from L to R with f (e) = 1, each node in L and R participates in at most one edge in M, with with f (e) = 1 so $|M| = k$: considering the cut $(L \cup s, R \cup t)$. □

5.5 Edge Disjointed Paths

Definition:Two paths are edge-disjoint if they have no edge in common.

Edge Disjointed Problem: Given a digraph G = (V, E) and two nodes s and t, find the max number of edge-disjoint $s \Rightarrow t$ paths.

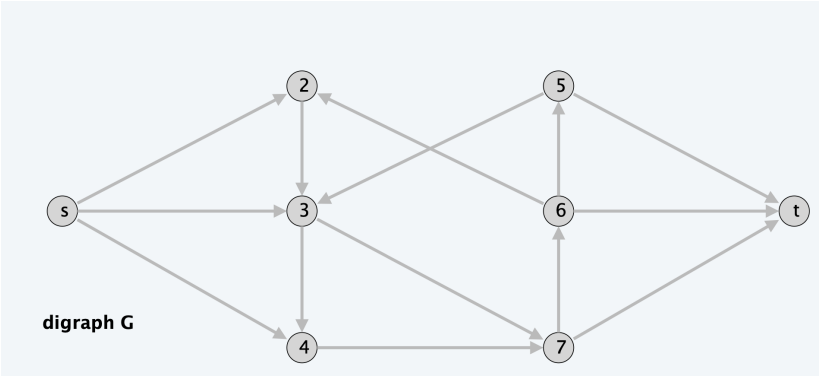


Figure 20: Instance of the Disjointed Path problem

Claim Given a digraph G=(V,E) with $c(e) = 1$, $\forall e \in E$ the max number edge-disjoint $s \Rightarrow t$ paths equals value of max flow.

Proof Suppose max flow value is k: Integrality theorem ⇒ there exists 0-1 flow f of value k. Now Consider edge (s, u) with f(s, u) = 1.
By conservation, there exists an edge (u, v) with f(u, v) = 1 continue until reach t, always choosing a new edge, the solution produces k (not necessarily simple) edge-disjoint paths. □

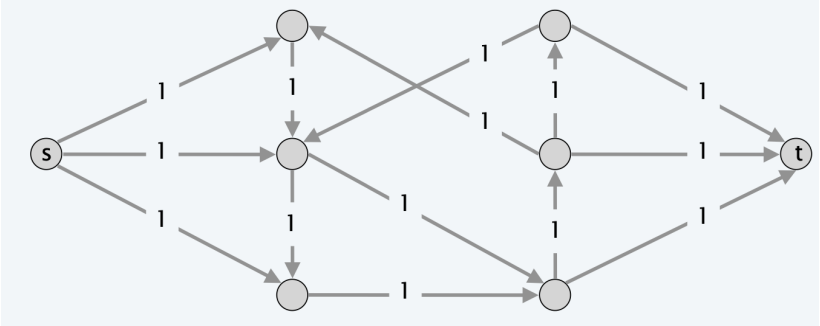


Figure 21: Instance of the Disjointed Path problem

Claim *Manger's Theorem*: The max number of edge-disjoint $s \Rightarrow t$ paths is equal to the min number of edges whose removal disconnects t from s.

Proof Suppose max number of edge-disjoint paths is k: then value of max flow = k, using the max-flow min-cut theorem ⇒ there exists a cut (A, B) of capacity k. Let F be set of edges going from A to B. $|F| = k$ and disconnects t from s. □

The Manger's theorem can even be applied to undirected graphs: (it's always possible to transform an undirected graph to directed by simply replacing every edge (u,v) with 2 direct edges (u,v) and (v,u)), then the formulation and proof is the same as above.

5.6 Circulation With demands

Given a digraph G = (V, E) with nonnegative edge capacities c(e) and node supply and demands d(v), a circulation is a function that satisfies:

$$\forall e \in E: \ 0 \leq f(e) \leq c(e) \ (capacity)$$

$$\forall v \in V: \sum_{e \ out \ of \ v} f(e) = \sum_{e \ into \ v} f(e) = d(v) \ (flow \ conservation)$$

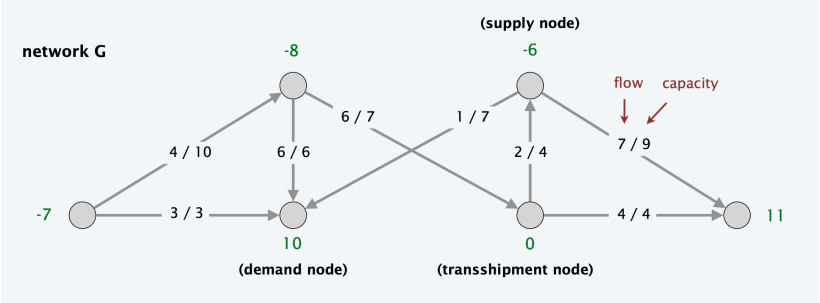


Figure 22: Instance of the Circulation problem

In order to transform the circulation into a flow problem we need to connect the source s to each node with $d(v) < 0$ and t with each node with $d(v) > 0$.
The original Graph G has a circulation if G' has a max flow of value:

$$\sum_{v \text{ in } d(v)>0} d(v) = \sum_{v \text{ in } d(v)<0} -d(v)$$

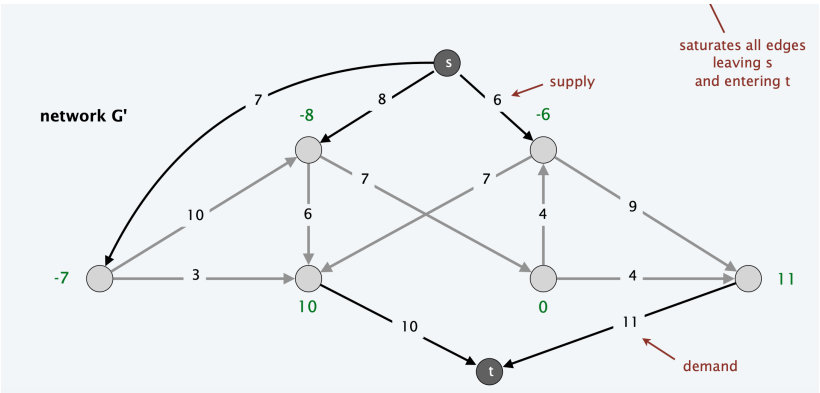


Figure 23: Instance of the Circulation problem with source and sink

Claim Given (V, E, c, d), there does not exist a circulation iff there exists a node partition (A, B) such that $\sum_{v \in B} d(v) > cap(A, B)$.

Proof By construction, simply looking at the minimum cut of G'. □

Circulation with Lowerbound:

Same as before, the problem is the following: Given (V, E, l, c, d), (with l(e) that express lowerbound on $\forall e \in E$) does exist a feasible circulation?
The solution is the same as before, modeling the lowerbound problem as a circulation with demands:

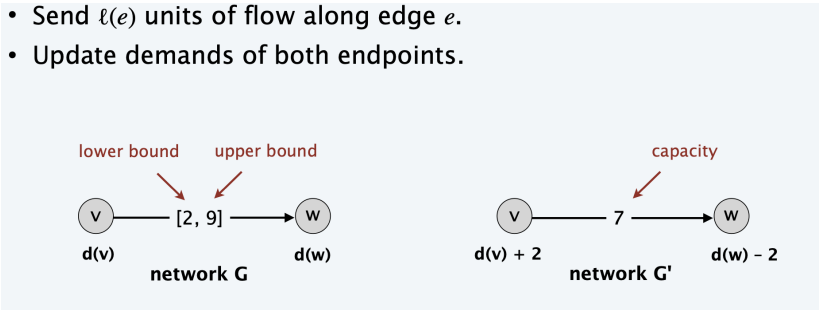


Figure 24: Instance of the Circulation problem with lowerbound

Claim There exists a circulation in G iff there exists a circulation in G'. Moreover, if all demands, capacities, and lower bounds in G are integers, then there is a circulation in G that is integer-valued.

Proof By construction: f(e) is a circulation in G iff f '(e) = f (e) – l(e) is a circulation in G'. □

5.7 Survey Design

Design survey asking n consumers about n products but you can only survey consumer i about product j if they own it, you can ask consumer i between ci and ci' questions and only between pj and pj' consumers about product j, the goal is to design a survey that meets these specs, if possible.

Solution: Model as circulation problem with lower bounds:add edge (i, j) if consumer j owns product i, connect every product to t and every client to s and lastly connect t to s: the integrality circulation is a feasible survey design.

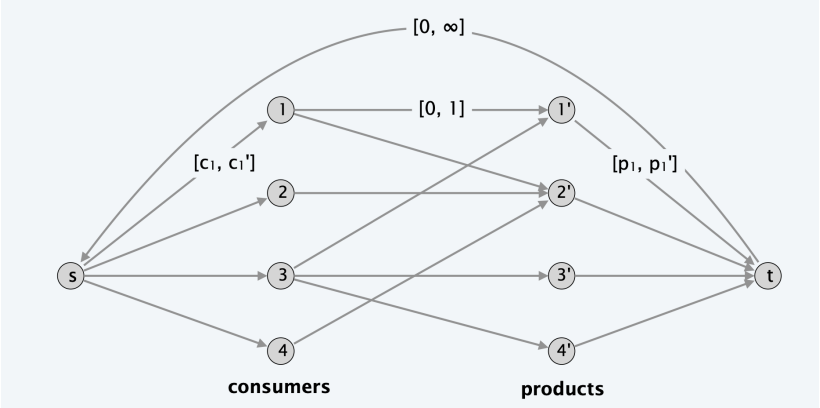


Figure 25: Instance of the Survey Problem

5.8 Airline Scheduling

Airline Scheduling is Complex computational problem faced by nation's airline carriers. Produces schedules that are efficient in terms of: equipment usage, crew allocation, customer satisfaction and in presence of unpredictable issues like weather, breakdowns.
The solution proposed below is a "toy problem" because we can: reusing flight multiple times, each flight i leaves origin o_i at time si and arrives at destination di destination at time f_i.The goal is to minimize number of flight crews.

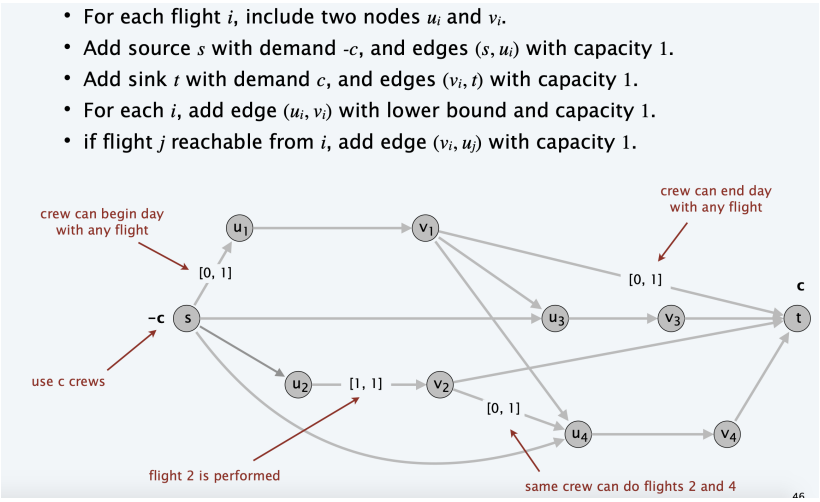


Figure 26: Instance of the Airline Problem

Claim The airline scheduling problem can be solved in $O(k^3 logk)$ time, with k = number of flights.

Proof The number c (of crews is uknown) we have $O(k)$ nodes and $O(k^2)$ edges so, at most, k crews are needed and by solving the k-circulation problem with at most, k augmenting paths, we obtain $O(k^3 logk)$. □

5.9 Project Selection

Having a set of possible projects P : project v has associated revenue p_v, a set of prerequisites E : if $(v, w) \in E$, cannot do project v unless also do project w. A subset of projects $A \subseteq P$ is feasible if the prerequisite of every project in A also belongs to A. Given a set of projects P and prerequisites E, choose a feasible subset of projects to maximize revenue.

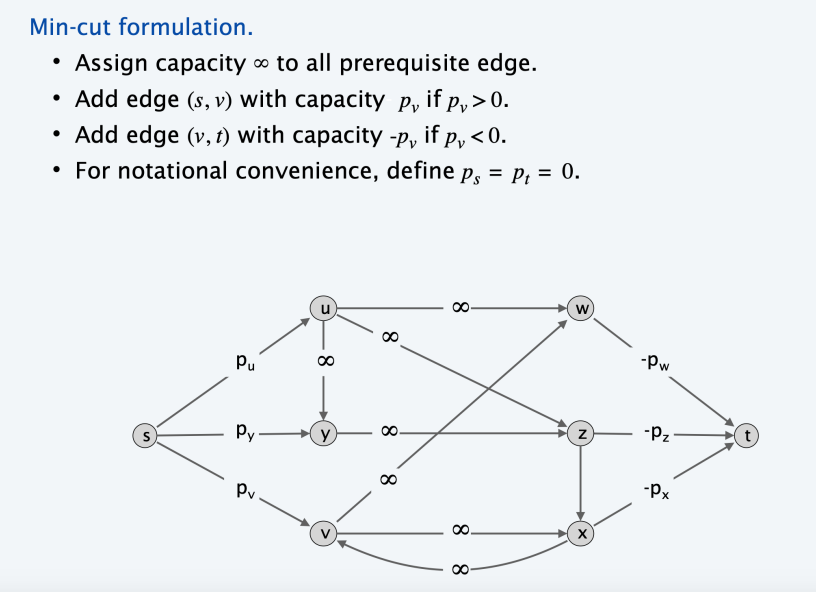


Figure 27: Instance of the Projects selection Problem

Claim (A, B) is min cut iff A $- \{s\}$ is optimal set of projects.

Proof Infinite capacity edges ensure $A - \{s\}$ is feasible. □

5.10 Baseball Game

Which teams have a chance of finishing the season with the most wins? Ex: Montreal is mathematically eliminated (Montreal finishes with ≤ 80 wins but Atlanta has already 83). The answer depends not only on how many games already won and left to play, but on whom they're against.
So we have a set of teams S, distinguished team $z \in S$, team x has won w_x games already and x and y play each other r_{xy} additional times.

Given the current standings, is there any outcome of the remaining games in which team z finishes with the most (or tied for the most) wins?

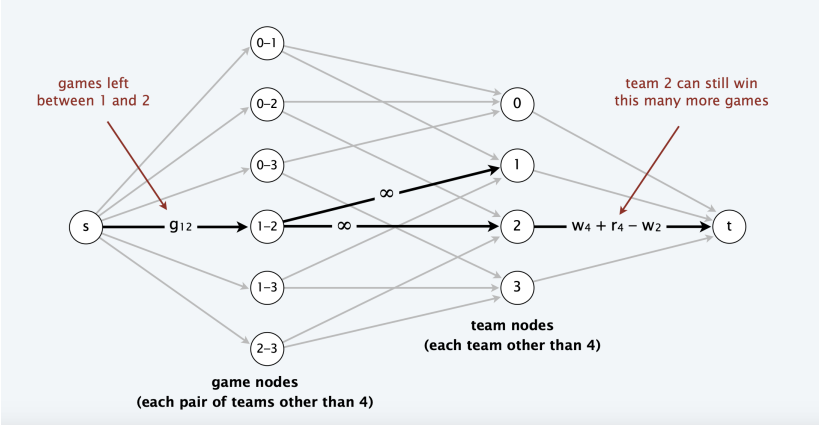


Figure 28: Instance of the Baseball Game Problem

Claim Team 4 not eliminated iff max flow saturates all edges leaving s.

Proof Integrality theorem ⇒ each remaining game between x and y added to number of wins for team x or team y, the capacity on (x, t) edges ensure no team wins too many games. □

Certificate of elimination.

$$T \subseteq S, \quad w(T) := \overbrace{\sum_{i \in T} w_i}^{\#\text{ wins}}, \quad g(T) := \overbrace{\sum_{\{x,y\} \subseteq T} g_{xy}}^{\#\text{ remaining games}},$$

Theorem. [Hoffman-Rivlin 1967] Team z is eliminated iff there exists a subset T^* such that

$$w_z + g_z < \frac{w(T^*) + g(T^*)}{|T^*|}$$

Pf. ⇐

- Suppose there exists $T^* \subseteq S$ such that $w_z + g_z < \frac{w(T^*) + g(T^*)}{|T^*|}$.
- Then, the teams in T^* win at least $(w(T^*) + g(T^*)) / |T^*|$ games on average.
- This exceeds the maximum number that team z can win. ▪