Greedy Algorithms

Solutions to the Practical Exercises

DA 2023 Instructors Team

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2023

A - Fundamentals

Exercise 1

a) See source code.
b) The worst-case scenario corresponds to having to process all of the coins, namely if all the coins are needed to pay the required amount, T. In this case, the algorithm has a time complexity of O(n*S) since each coin can be added in O(1) time and no sorting needs to be performed.
c) The algorithm may not return the optimal solution for some coin systems. These systems are known as "non-canonical", as opposed to the "canonical" system, which always allows a greedy algorithm to compute the optimal solution. Most coin systems used in real-life are canonical (examples: Euro, British pound, American dollar).

A simple example of a non-canonical system is [1, 3, 4]. Consider the goal of making change for 6 cents (T = 6) while having an infinite/very high amount of stock for each coin (2 of each coin is enough for this example). The solution to the greedy algorithm uses 3 coins: one of 4 cents and two of 1 cent. However, the optimal solution uses only 2 coins, in this case two 3 cents coins.

Exercise 2

a) The input is a set of N activities $Acts = \{A_1, A_2, \ldots, A_N\}$ with start time s_i and finish time f_i with $i \in [1, N]$ and $s_i \leq f_i$.

For a solution X, the output is a list of N 0-1 integers (booleans) $X.Selected = [X.sel_1, X.sel_2, \ldots, X.sel_N]$, such that $X.sel_i$ is 1 if and only if the i^{th} activity is selected. The objective function (to maximize) is the number of activities selected:

$$X.NActs = \sum_{i=1}^{N} X.sel_i$$

The output list, *X.Selected*, is subject to the following constraint, that indicates that two selected activities must not overlap:

$$\forall i \forall j \ (i < j \wedge X.sel_i \wedge X.sel_j) \rightarrow (f_i \leq s_j \vee f_j \leq s_i)$$

b) See source code. The main idea of the algorithm is to sort the activities by increasing finish time and then select the ones that do not overlap with the most recently selected activity. There is no need to test if they overlap with the other previously selected activities, since they have earlier finish times and thus, if they overlap, so will the most recently selected activity.
c) The algorithm has two main steps:
   - Sorting the activity vector takes $\Theta(n \times \log(n))$ time.
   - Selecting the activities to include takes $\Theta(n)$ time ($\Theta(1)$ for each of the n activities, since activity overlap only needs to be tested once for each activity processed in the sorted vector).

   Thus, the time complexity is $\Theta(n \times \log(n) + n) = \Theta(n \times \log(n))$. This complexity is valid for the best, average and worst cases, since all activities always have to be processed.
d) To prove that the greedy choice is optimal, the following lemma will first be proved by induction.

Lemma: Considering that the activities are ordered in ascending order of finishing time, the finishing time of the i^{th} activity of the greedy solution S is lower or equal than the finishing time of the i^{th} activity of any other solution S': finish(S, i) ≤ finish(S', i)

Proof of the lemma by induction:

Base case: i = 1. Since S chooses the earliest finish activity, by definition, we have finish(S,1) ≤ finish(S',1).

Inductive step: i > 1. The induction hypothesis states that finish(S, i) ≤ finish(S', i).

Then, for the $(i+1)^{th}$ activity, S has at least the same activities as S' available for adding to the solution, since S can choose any activity j where $s_j \geq$ finish(S, i) whereas S' must choose activities for which $s_j \geq$ finish(S', i), and according to the inductive hypothesis, finish(S, i) ≤ finish(S', i).

Therefore, S can choose the earliest finishing time activity out of all available to S', or possibly an activity that finishes earlier and is unavailable to S', thus proving that finish(S, i + 1) ≤ finish(S', i + 1).

To prove that the greedy solution S is better than any solution S', let us assume by contradiction that S' is better than S, that is, S'.NActs > S.NActs.

According to the lemma, finish(S,i) < finish(S',i). Since S'.NActs > S.NActs, then there are additional activities selected by S', where $s_j \geq$ finish(S',i) for $i < j \leq n(S')$. However, for all those activities, we have $s_j \geq$ finish(S,i) so those activities are also available for S to choose, so S could obtain S'.NActs activities by selecting that same set of activities, which contradicts the assumption that S'.NActs > S.NActs.

Exercise 3

a) The input is a set of N tasks, TaskSet = $\{t_1, t_2, \ldots, t_N\}$, where t_i is the duration of the i^{th} task, with $i \in [1, N]$.

The output is an ordered list of tasks in TaskSet $[t_1, t_2, \ldots, t_N]$ and the objective function (to minimize) is the average completion time, given by:

$$Cost(\{t_1, t_2, \ldots, t_N\}) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{i} t_i}{N}$$

Or simply, the sum of the completion times:

$$Cost(\{t_1, t_2, \ldots, t_N\}) = \sum_{i=1}^{N} \sum_{j=1}^{i} t_i$$

Notice that in these two formulae, brackets ({}) were used to represent sets of tasks, which do not have an order.

b) See source code. The main idea of the algorithm is to sort the tasks by increasing completion time.

c) The algorithm has two main steps:
   - Sorting the activity vector takes $\Theta(n \times \log(n))$ time.
   - Processing each task takes $\Theta(n)$ time ($\Theta(1)$ for each of the n tasks).

   Thus, the time complexity is $\Theta(n \times \log(n) + n) = \Theta(n \times \log(n))$. This complexity is valid for the best, average and worst cases, since all tasks always have to be processed.

d) Let Ts be a random ordering of N tasks. Let $j = i + 1$, such that $1 \leq i < j \leq N$ and $j = i + 1$. If we swap two consecutive tasks i and j:
   - the completion time of the activities before i does not change since they do not depend on the tasks that occur afterwards.
   - the completion time of the tasks that occur after j does not change either since the sum of the tasks before i is the same and the sum of the duration of tasks i and j is the same (they just swap order and the addition operation is commutative, *i.e.*, $a + b = b + a$).

Thus, only the completion time of tasks i and j changes.

The completion time of task i increases by t_j and the time of task j decreases by t_i, so the net change in the cost is:

$$\Delta = t_j - t_i$$

In order for an ordering of the tasks to be the optimal solution, there must not be any swap that allows for negative net changes (which decrease the cost of the solution). In other words, we must have:

$$\forall i \forall j \ (i < j) \rightarrow (\Delta = t_j - t_i \geq 0) <=> \forall i \forall j \ (i < j) \rightarrow (t_i \leq t_j)$$

In other words, tasks must be ordered by duration. In the formulae above, we assume that i < j (without loss of generality). As a result, the global optimal solution can be achieved greedily selecting the task with the smallest duration out of those that are still unselected.

Since the tasks must be ordered by duration, we can now define the recurrence formula for the optimal solution, CostOpt.

$$CostOpt(\{t_1\}) = t_1$$

Else, if TaskSet has more than 1 task:

$$CostOpt(TaskSet) = t_1 + CostOpt(TaskSet')$$

with $t_1 = min(TaskSet) \wedge TaskSet' = TaskSet \setminus \{t_1\}$

Exercise 4

a) The input is a set of N items with a weight $weight_i$, and a value, $value_i$, with i ∈ [1, N].

For a solution X, the output is a list of N floating-point numbers $X.Amounts = [X.a_1, X.a_2, \ldots, X.a_N]$, such that $0 \leq X.a_i \leq 1$, for all i ∈ [1, N], representing the amount used in the knapsack of the i^{th} item.

The objective function (to maximize) is the total value of the items used:

$$X.Value = \sum_{i=1}^{N} value_i \times X.a_i$$

The output list, *X.Amounts*, is subject to the following constraint, that indicates that the knapsack's maximum weight capacity cannot be exceeded:

$$\sum_{i=1}^{N} weight_i * X.a_i \leq maxWeight$$

b) See source code. The main idea of the algorithm is to sort the items by decreasing order of the of value/weight ratio, and then add as much as possible of each item (in the sorted order) until the knapsack is full.

c) The algorithm has two main steps:
- Sorting the activity vector takes Θ(n×log(n)) time.
- Processing each item takes O(n) time (Θ(1) for each of the n items), as in the worst-case, all items need to be processed (for instance, if the knapsack has a very large capacity).

Thus, the time complexity is O(n×log(n) + n) = O(n×log(n)).

d) To prove that the greedy solution S is better than any solution S', let us assume by contradiction that a solution S' not obtainable using the greedy strategy is optimal. Therefore: S'.Value > S.Value.

On the one hand, since S' is optimal, it uses as much of the knapsack's capacity as possible (or else a better solution could be found by any more of any item). On the other hand, by definition, S also uses as much of the knapsack's capacity as possible since this is a greedy algorithm. Therefore, as solutions S and S' differ, there is at least one item i used in S whose amount ($S.a_i$) is lower than in S' and another item j whose used amount is larger in S' than S. Item j has a lower value/weight ratio than item i, or else, by definition of the greedy algorithm, S would use some of (or all of) the space occupied by item i to use (more of) item j. We thus obtain:

$$\frac{value_i}{weight_i} > \frac{value_j}{weight_j}$$

It should be noted that items i and j cannot have the same value to weight ratio because, if there is no such item j such that the strict inequation above holds, then S' is also obtainable using the greedy algorithm, which is a contradiction.

Consider modifying solution S' to make a new solution S": by adding a small amount of item i and reducing the amount of item j as follows:

$$S''.a_i \times S'.a_i + \varepsilon \text{ and } S''.a_j \times S'.a_j - \frac{weight_i}{weight_j}\varepsilon$$

The total weight of the solution is the same since:

- the increase in weight due to using more of item i is: $weight_i \times \varepsilon$.
- the decrease in weight due to using less of item j is: $weight_j \times \frac{weight_i}{weight_j}\varepsilon$.
- both these quantities are the same: $weight_i * \varepsilon = weight_j \times \frac{weight_i}{weight_j}\varepsilon$.

However, S".Value > S'.Value since:

- the increase in value due to using more of item i is: $value_i \times \varepsilon$.
- the decrease in value due to using less of item j is: $value_j \times \frac{weight_i}{weight_j}\varepsilon$.
- the overall difference in value is given by: $value_i \times \varepsilon - value_j \times \frac{weight_i}{weight_j}\varepsilon = \varepsilon \times (value_i - value_j \times \frac{weight_i}{weight_j})$. This difference is positive since (using the ratio inequation from before):

$$\frac{value_i}{weight_i} > \frac{value_j}{weight_j} \Leftrightarrow value_i > value_j \times \frac{weight_i}{weight_j} \Leftrightarrow value_i - value_j \times \frac{weight_i}{weight_j} > 0$$

Therefore, we reach a contradiction: S' cannot be optimal since a better solution S" can be derived. Ergo, it is not possible to obtain a better solution than the one computed using the greedy algorithm, which is thus optimal.

e) The adapted version of the greedy algorithm only adds an item to the knapsack if it can be fully added, or else, it skips to the next item. Items are still processed in decreasing order of the value/weight ratio.

As a concrete example, consider the following problem instance:

values = [4, 4, 7], **weights** = [2, 2, 3], **n** = 3, **maxWeight** = 4

Here, the value/weight ratios are, respectively, 2, 2 and 7/3 > 2. Therefore, the greedy algorithm would start by adding the third item and would then be unable to add more items, resulting in a total value of 7 (and a total weight of 3). The optimal solution is using the first two items, which yields a total value of 8 and uses all 4 units of weight of the knapsack. Thus, the greedy strategy of the fractional knapsack problem cannot be translated to the integer version of the problem.

B - Notable applications

Exercise 5
See source code.

Exercise 6
See source code.

Exercise 7
See source code.

Exercise 8

a) The challenge faced by Bernard corresponds to the **minimum spanning tree (MST) problem**. This problem can be solved with Prim's or Kruskal's algorithm. Both alternative solutions will be presented.

Option A (Prim) The table below details the computations for Prim's algorithm. In each iteration, the table details: the current vertex being processed, the updates to vertices' shortest distance to the MST and ancestor in the MST, and the current state of the heap/priority queue. This algorithm needs to start on any vertex. In this solution, the algorithm is run from node A.

Iteration	Current vertex (dist, path)	Vertices (dist, path)	Heap [top .. bottom]
1	A	B(2, A) C(2, A), D(0, A)	[D, B, C]
2	D(0, A)	C(1, D), E(1, C), F(3, D)	[C, E, B, F]
3	C(1, D)	_	[E, B, F]
4	E(1, C)	G(2, E)	[B, G, F]
5	B(2, A)	_	[G, F]
6	G(2, E)	_	[F]
7	F(3, D)	_	[]

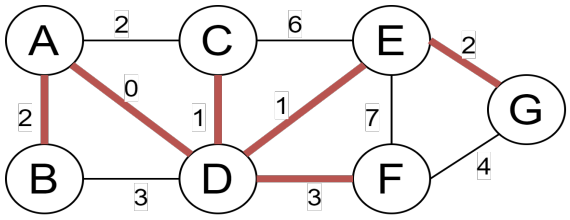
In this example, the distance to node C was updated, which required executing the *decreaseKey* operation in the heap, so that node C is moved to the top of the head, in O(log(n)) time.

The algorithm ends once the heap is empty.

Option B (Kruskal): Using Kruskal's algorithm, the edges are ordered by increasing weight:

Edge (weight)	Used?
AD (0)	yes
CD (1)	yes
DE (1)	yes
AB (2)	yes
AC (2)	X
EG (2)	yes
BD (3)	X
DF (3)	yes
FG (4)	X
EC (6)	X
EF (7)	X

Solution (the edges of the tree are marked in red):



Conclusion: Using either of the algorithms, the total score is: 0 + 1 + 1 + 2 + 2 + 3 = 9

b) The challenge faced by Bianca corresponds to the **second minimum spanning tree problem**.

The main idea of the algorithm is to temporarily exclude each edge of the graph, one at a time, and recompute the MST.

Pseudo-code for finding the second minimum spanning tree:

Input and auxiliary functions:
- V: set of vertices
- E: set of edges
- kruskal(V, E): runs Kruskal's algorithm assuming E has its edges ordered by increasing weight. Returns the tree and its total cost.

Output: second minimum spanning tree and its weight

```
compute_second_MST(V, E)
   sort(E)
   m, cost ← kruskal(V,E)
   minSecondCost ← ∞
   mSecond ← NULL
   foreach (edge e: m.edges)
      m',cost' ← kruskal(V,E={e})
      if(cost' < minSecondCost)
         msecond = m'
         minSecondCost ← cost'
   return mSecond, minSecondCost
```

The time complexity of this algorithm is $O(E \times \log(E)+E+(V-1)\times E) = O(E \times \log(V)+V \times E) = O(V \times E)$

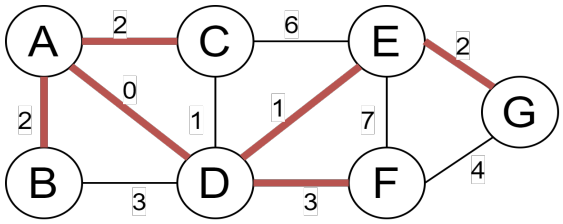
The $(V-1)\times E$ term corresponds to the for loop, which executes (V-1) times (one for each edge included in the minimum spanning tree m) and runs in O(E) time per iteration, given this version of Kruskal's algorithm already receives a presorted list of edges.

c) The table below details how Kruskal's algorithm is run after trying to exclude each edge. Cells marked in yellow correspond to cases where an edge that was not part of the MST was included in the new spanning tree.

Edge (weight)	Used after excluding the edge?					
	AD	CD	DE	AB	EG	DF
AD (0)	X	yes	yes	yes	yes	yes
CD (1)	yes	X	yes	yes	yes	yes
DE (1)	yes	yes	X	yes	yes	yes
AB (2)	yes	yes	yes	X	yes	yes
AC (2)	yes	yes	X	X	X	X
EG (2)	yes	yes	yes	yes	X	yes
BD (3)	X	X	X	yes	X	X
DF (3)	yes	yes	yes	yes	yes	X
FG (4)	X	X	yes	X	yes	yes
EC (6)	X	X	X	X	X	X
EF (7)	X	X	X	X	X	X
Total weight	11	10	12	10	11	10

There are three solutions that give 10 points to Bianca, thus giving a total of 19 points for the team.

Solution obtained by excluding edge CD:



Exercise 9

There are 6 different characters: 'a', 'b', 'd', 'n', 's' and the space character. Thus, the minimum number of bits needed is: ⌈log2 (6)⌉ = ⌈2.5849625 …⌉ = 3. The following encoding can be used:

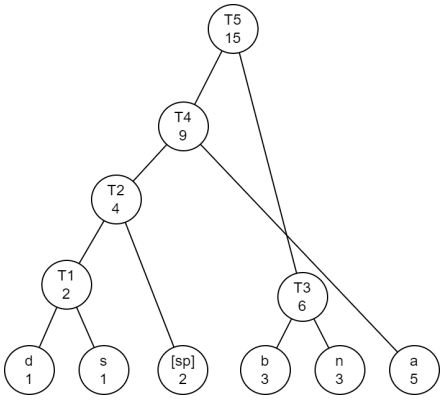
Character	a	b	d	n	s	space = [sp]
Code	000	001	010	011	100	101

The encoded string is: 001 000 011 101 001 000 010 101 001 000 011 000 011 000 100
b a n [sp] b a d [sp] b a n a n a s

The total cost of the text encoding is 3 * number of characters = 3 × 15 = 45 bits.

a) Using Huffman's algorithm, the Huffman encoding is performed by building a binary tree. It starts with a set of nodes, one per distinct character, whose weight is the number of times the character is present in the text. This algorithm is greedy since it prioritizes combining the two nodes with least weight (in the case of a tie, choosing any of the tied nodes is fine). When combining two nodes, a new node is created whose weight is the sum of the character counts of its child nodes.

The resulting tree is:



The node annotation "Tn" indicates that it was the n-th partial tree to be built. Each character's encoding is determined by traversing the Huffman tree, from its root to each leaf (where the characters are located). Each time one left child of the binary tree is selected during the traversal, a 0 is added to the prefix of the code, and, for each right child, a 1 is added. This results in the table below.

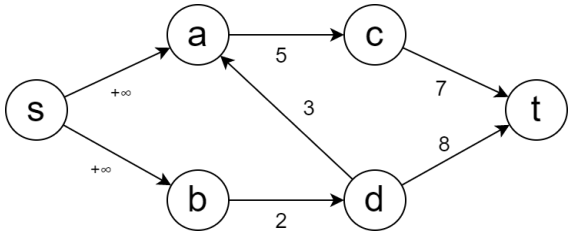
Character	a	b	d	n	s	space = [sp]
Code	01	10	0000	11	0001	001

The encoded string is: 10 01 11 001 10 01 0000 001 10 01 11 01 11 01 0001
b a n [sp] b a d [sp] b a n a n a s

The total cost of the text encoding is 36 bits, which is less than with the fixed-length encoding.

Exercise 10

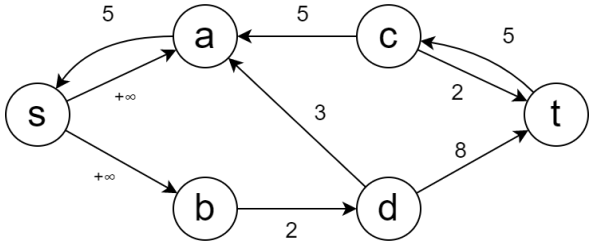
a) This corresponds to the **maximum flow problem**, which can be modeled by the flow graph below.



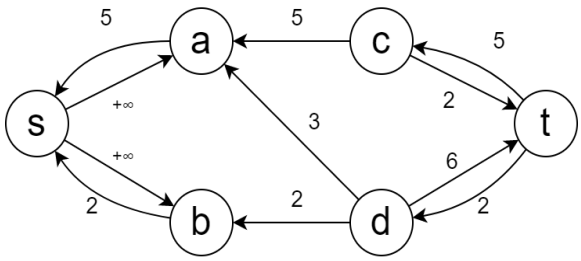
The maximum flow can be computed using the Edmonds-Karp algorithm. This algorithm requires there to only be one source and one sink node. The sink is node t, but there are two sources: nodes a and b. Thus, a new node s is created to serve as the graph's single source. Node s is connected to the original source nodes by edges of infinite capacity, to prevent them from being the flow bottleneck.

The Edmonds-Karp algorithm works with residual graphs, which indicate how much flow can still traverse each edge. Initially, the residual graph is equal to the capacity graph. In each iteration, a augmenting path from node s to t is found using BFS (Breadth-First Search) and then the residual graph is updated, namely by creating edges in the reverse direction of the augmenting path, which allows for the actual flow of an edge to be reduced in subsequent iterations if necessary. The algorithm ends when there are no augmenting paths in the residual graph.

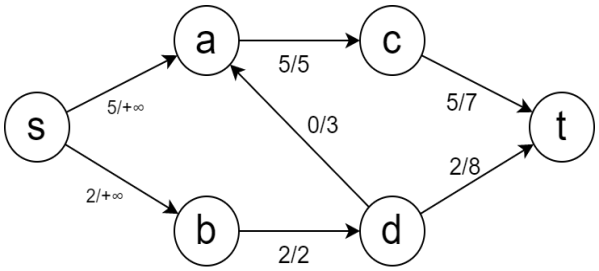
In the graph above, one possible augmenting path is: s→a→c→t with a flow of min(+∞,5,7) = 5. After using this augmenting path to push flow from s to t, the residual graph is as shown below:



In this new residual graph, one possible augmenting path is now s→b→d→t with a flow of min(+∞,2,8) = 2, leading to a residual graph as shown below.



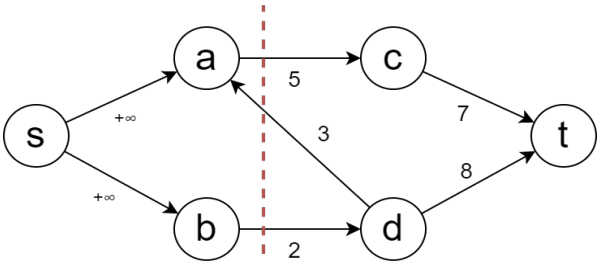
The algorithm ends as there are no augmenting paths. The maximum flow is the sum of the flows of the previously computed augmenting paths: 2 + 5 = 7. The flow graph for the maximum flow is:



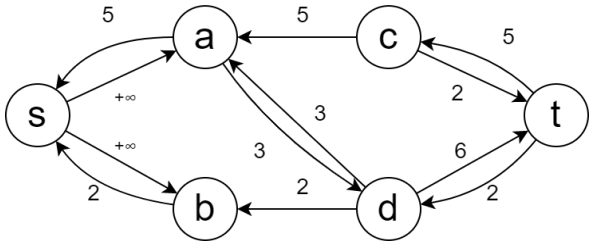
Note: in this case, the two augmenting paths traverse different nodes of the residual graph, thus the second augmenting path could have been found during the BFS before the first one, since both paths traverse the same number of vertices.

b) According to the max-flow min-cut theorem, the minimum s-t cuts of this problem's capacity graph are equal to its maximum flow: 7. A s-t cut partitions the graph's nodes in two disjoint sets, one containing s (set S), and the other, t (set T). The weight of a cut is the sum of the capacities of the edges flowing from a node from set S to a node from set T.

The figure below depicts this graph' only s-t cut. Notice how the edge of weight 3 does not count for the cut' weight, which is equal to 5 + 2 = 7.

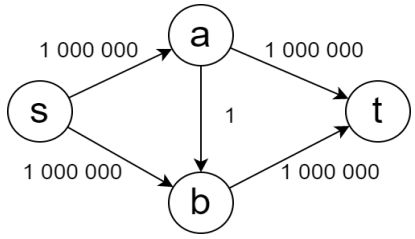


c) The Edmonds-Karp algorithm does not need to be run from a new initial graph to determine the new maximum flow. Instead, the algorithm can run in the final residual graph of the previous execution of this algorithm by making the following modification to the graph: a new edge of capacity 3 is added from a to d (the edges in the reverse direction is maintained). Note that this principle can be used to solve maximum flow problems in undirected graphs. The new residual graph is:



There is a new augmenting path: s -> a -> d -> t, with a flow of min(+∞, 3, 6) = 3. Therefore, adding the pump helps to increase the maximum flow from 7 to 10 (= 7 + 3).

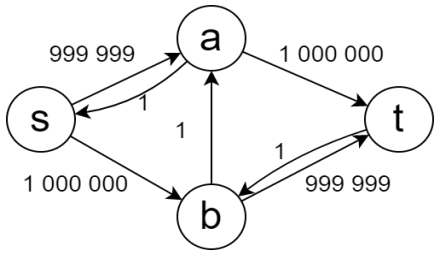
Exercise 11
Consider the graph below, where the maximum flow is 2 000 000.



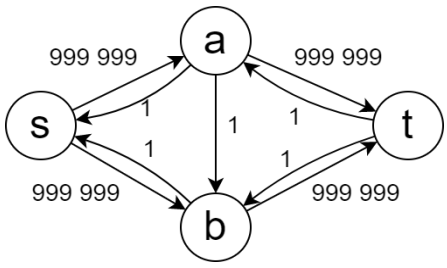
The choice of the augmenting path finding algorithm has a very large impact on the number of iterations needed to run the Ford-Fulkerson method in the graph above.

If the augmenting path finding algorithm is BFS (Breadth-First Search), then it the method corresponds to the Edmond-Karp algorithm and the maximum flow can be found in two iterations: in the first one, the augmenting path is s -> a -> t, with a flow of 1 000 000; in the second one, the augmenting path is s -> b -> t, also with a flow of 1 000 000.

However, consider an augmenting path finding algorithm that modifies BFS by ignoring edges that reach the sink when the current path only has one edge. In this case, the only valid initial augmenting path is: s -> a -> b -> t, with a flow of 1. This produces the following residual graph.



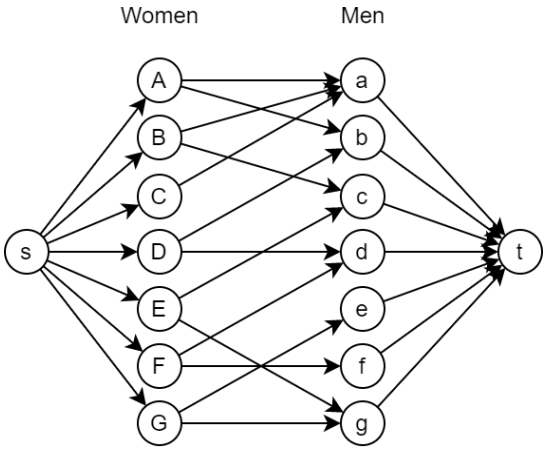
The next unique valid augmenting path is: s -> b -> a -> t, with a flow of 1, which produces the following residual graph:



Following the same procedure, the current flow is incremented by 1 in each iteration. Thus, a total of 2 000 000 iterations are needed, compared to just 2 for Edmonds-Karp. Thus, for this graph, an incorrect choice of the augmenting path algorithm can lead to an execution time that is 1 000 000 times greater, on average.

Exercise 12

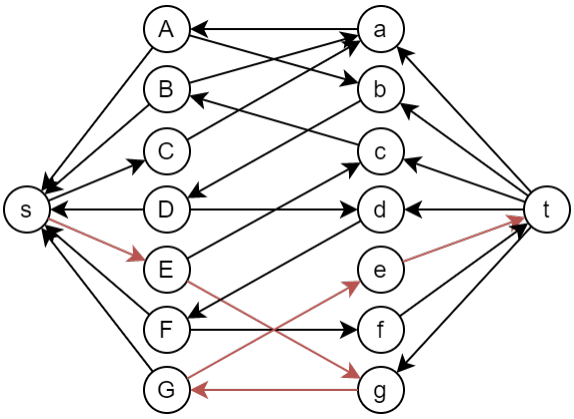
a) This problem corresponds to the **Maximal Matching problem**, which can be solved by modeling it as a maximum flow problem. The equivalent capacity graph for this problem (without taking into account the initial matches) is:



This graph follows a set of rules:

- All edges have a capacity of 1.
- The first layer of nodes represents women, and the second, men.
- There is an edge connecting a woman to a man, if and only if, the match is possible (i.e. both people are friends).
- A flow of 1 going from a node associated with a woman to a node associated with a man represents a match between these two people.
- The outgoing flow of s (and the incoming flow to t) is equal to the current number of matches.

Given that there is an initial matching ({(A, a), (B, c), (D, b), (F, d), (G, g)}), some of the graph's edges have a flow of 1, which produces the residual graph below, which serves as the starting point of the Edmonds-Karp algorithm. The current flow is 5, since there are 5 matches.

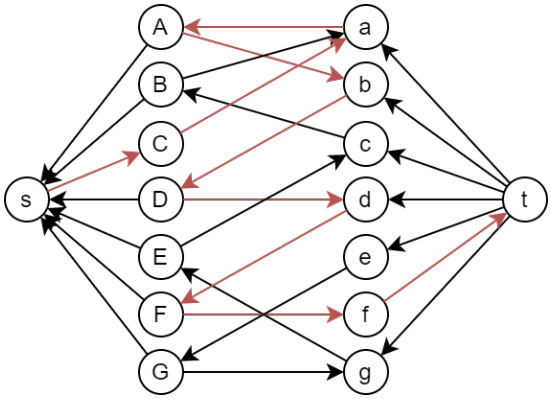


The graph has one augmenting path (highlighted in red): s->E->g->G->e->t, with a flow of 1. This

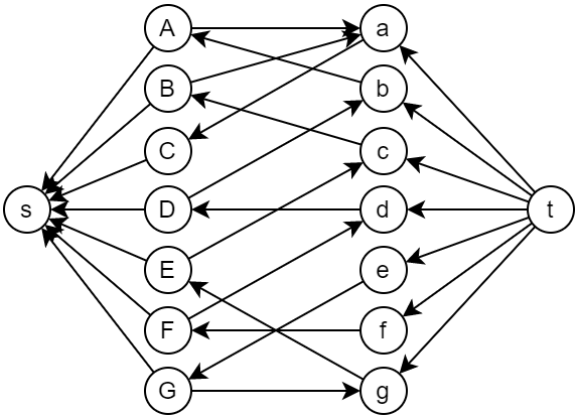
path uses one of the "backwards edges" created between iterations of the Edmonds-Karp algorithm, which corresponds to resetting one of the capacity graph's edge's flow back to 0 and thus represents removing one match from the current matching. This path represents:

- pairing Elizabeth with George;
- unpairing George with Grace;
- pairing Grace with Eugene.

The new residual graph is:



The augmenting path is: s -> C -> a -> A -> b -> D -> d -> F -> f -> t, with a flow of 1, thus producing a new residual graph:



Since there are no more augmenting paths, the optimal solution has been reached, with a flow of 7. The optimal matching is: {(A, b), (B, c), (C, a) (D, d), (E, g), (F, f), (G, e)}, with the following pairs: Anna-Bruno, Berta-Charles, Claire-Anthony, Diane-Dennis, Elizabeth-George, Faith-Fred, Grace-Eugene.

The final residual graph actually does not need to be computed since the maximum number of matches is 7 (since there are only 7 women and they cannot dance with two different people at the same time) and the current flow is already 7.

Another way of identifying that the maximum flow has already been reached is computing the s-t cut that separates the source node (node s) from all the other nodes, which has a weight of 7. One consequence of the max-flow min-cut theorem is that the max flow is less or equal to any s-t cut weight. Thus, since the aforementioned s-t cut has a weight of 7 and the current flow is 7, the maximum flow has to be 7.

b) When maximal matching problems are modeled as maximum flow problems as in this exercise, the flow increases by 1 in each iteration of the Edmonds-Karp algorithm (or with the Ford-Fulkerson method, in general), given that all edges have a weight of 1. Therefore, if M is the cardinality of the maximal matching and m is the cardinality of the initial matching, only (M - m) iterations are needed to find the optimal solution, rather than M, thus saving the time of performing the first m iterations.