COMP 251 - Assignment 4

Jeremie Poisson - McGill ID 260627104

McGill School of Computer Science – December 7th, 2015

1 Corrupted text sequence

(a) We are asked to design an efficient algorithm to determine whether a string s of n characters is a valid sequence of words. In order to solve the problem efficiently, we first need to define a proper notation :

- $s[i]$ will represent the substring of s from index 1 (first index) up to i inclusively. Similarly, $s[i..j]$ will represent the substring of s from index i up to j inclusively;

- $dict(s)$ is the function defined as follows applied to a string s;

$$dict(s) = \begin{cases} true & \text{if } s \text{ is a valid dictionary word} \\ false & \text{otherwise} \end{cases}$$

- $valid[i]$ will represent whether the string $s[i]$ is formed using a sequence of valid words (recursively defined) or not.

We need to recursively define $valid[i]$ for any string $s[i]$. To find a proper dynamic programming solution, it is crucial to take a look at the subproblems this problem generates...

In this case, we could define valid subproblems as being any string that that satisfy both of the following cases :

For any indices $1 \leq i < j$, we say that $valid[i]$ is a valid subsequence if and only if :

- $valid[i]$ (from 1 up to $i < j$) is a valid subsequence formed of valid words (recurrence relation);
- $dict(s[j + 1..i]) = true$ (i.e. the rest of the subsequence is a valid word.

In general, we will want to maximize these subproblems over all the fixed interval of indices where $1 \leq j < i$. This means that, for any given string s of length n, we effectively need to compute $valid[1]$, $valid[2]$, ..., $valid[n]$.

This yields the following recurrence relation :

$$valid[j]_{\forall 1 \leq i < j} = \begin{cases} false & \text{if } j \leq 0 \\ \max\{valid[i] \;\&\&\; dict(s[i+1..j])\} & \text{otherwise} \end{cases}$$

As mentioned before, given any string s of length n, we need to compute subproblems up to $valid[n]$. Hence, in order to start validating string s for checking if it is a valid sequence of words, we simply call the following:

$$\Rightarrow valid[n](\text{over string } s \text{ of length } n)$$

Algorithm 1 Corrupted text sequence

```
1:  procedure VALID(s, n)
2:      valid[1..n] = (empty array initialized to false values from 1..n)
3:      for j = 1; j < n; j++ do
4:          for i = 1; i < j − 1; i++ do
5:              if valid[i] && dict(s[i+1..j]) = true then
6:                  valid[j] = true
7:              end if
8:          end for
9:      end for
10:     return valid[n]
11: end procedure
```

This algorithm is clearly $O(n^2)$, as in its worst case, the algorithm is performing up to n iterations (for the inside loop) by looping over the n elements contained within string s (outer loop).

(b) It is possible to use memoization in order to keep track of valid words using the previously described algorithm. As we already have a case where we compute whether there exist a valid word from a certain index in our algorithm, it is fairly easy to alter the original algorithm to keep track of them.

Algorithm 2 Corrupted text sequence (altered using memoization)

```
1: procedure VALID(s)
2:     valid[1..n] = (empty array initialized to false values from 1..n)
3:     words = (empty list initialized to empty strings)
4:     for j = 1; j < n; j++ do
5:         for i = 1; i < j − 1; i++ do
6:             if valid[i] && dict(s[i+1..j]) = true then
7:                 valid[j] = true
8:                 words.append(s[i+1..j])
9:             end if
10:        end for
11:    end for
12:    return words
13: end procedure
```

2 Minimizing lateness

(a) Let's consider such a set J, where all jobs are schedulables and the set is of maximum size. By definition, we defined that a job is *schedulable* if and only if it is able to meet its deadline.

Now, considering the interval scheduling problem applied to set of jobs J (where the objective was to minimize the maximum lateness), we know that the minimum lateness for J is effectively 0, by definition. We have shown in the previous assignment that a suitable greedy algorithm to solve the maximum lateness problem is to sort jobs by their deadlines in increasing order accordingly.

It follows that sorting jobs in increasing order of their deadlines generates a feasible schedule (i.e. a set of jobs that are schedulable), as expected.

(b) We start by considering the subproblems being a set of jobs from 1 up to n. As with most of the dynamic programming problems we have seen, we start considering the last job n to define a proper recurrence relation.

We will also define the function *opt* as follows:

$opt(i, d)$ will denote the maximum number of jobs within the set of jobs $\{1..i\}$ that are meeting their deadlines by a time d.

Once again, we consider the following cases for the last job:

- **Case 1 :** *job i is included in the optimal solution ($opt(i, d)$).*
  At this point, the subproblem (read job n —we are considering the last job) is meeting the deadline d imposed by the *opt* function ($d_n \geq d$).

We should include this subproblem in our optimal solution –i.e. adding one to the optimal solution.

We also need to recursively check for other subproblems that may be part of the updated optimal solution. That is, subproblems (jobs) from 1 up to $i-1$ that are meeting the deadline $(d-t_i)$ —*as no jobs can be executed concurrently, we must subtract the time required to process job i from the maximum deadline.*

$$\Rightarrow 1 + opt(i-1, d-t_i)$$

- **Case 2 :** *job i is **not** included within the optimal solution ($opt(i,d)$).* We shouldn't include this solution to our optimal solution. Hence, we simply recursively check other subproblems from 1 to $i-1$, without updating our optimal solution whatsoever.

$$\Rightarrow opt(i-1, d)$$

Both cases yields the following recurrence relation :

$$opt(i, D) = \begin{cases} 0 & \text{if } i \leq 0 \\ \max\{opt(i-1, d), 1 + opt(i-1, d-t_i)\} & \text{otherwise} \end{cases}$$

Again, using memoization, we keep track of jobs to be included in the optimal solution. We are now able to describe a dynamic programming algorithm that solves this problem.

For its input, the algorithm takes in the following values :

- An integer n corresponding to the total number of jobs;
- An integer D corresponding to the maximum deadline $D = \max_i d_i$

As the algorithm is implemented using a dynamic programming approach, we will maintain a multidimensional array $sub[][]$ for each jobs (up to n) and its corresponding deadlines times (up to D). We will also maintain a $sol[n]$ list containing pairs (*job index*, *time*) for memoizing the solutions throughout the algorithm execution.

Basically, the algorithm is calculating the best deadline for every jobs i to be executed and keeps track of those pre-calculated values its sub-problem array *sub*. This value is calculated using the recurrence relation shown earlier. Hence, for each job i, the optimal time at which job i is to be executed will be found at the D^{th} entry of its sub-problem deadline array $sub[i]$ (as we are iterating over all deadline times).

This algorithm is clearly a polynomial-time algorithm, thus enters our definition of efficient, as we are iterating over the number of jobs and over all deadline values.

Algorithm 3 Scheduling jobs - minimizing lateness

```
1: procedure SCHEDULE(n, D)
2:     sub[0...n][0...D] = (empty array containing sub-problems)
3:     sol[0...n] = (empty list containing optimal sol. pairs)
4:     (initialize sub[0][d] for all deadline up to D to 0)
5:
6:     for each job i up to n do
7:         for d = 0; d < D; d++ do
8:             if sub[i − 1][d] > (sub[i − 1][d − t_i] + 1) then
9:                 sub[i][d] = sub[i − 1][d]
10:            else
11:                sub[i][d] = sub[i − 1][d − t_i] + 1
12:            end if
13:        end for
14:        sol.append((i, sub[i][D]))
15:    end for
16:    return sol
17: end procedure
```

3 Escape Problem

(a) It is fairly intuitive to imagine a native solution that emulates the purpose of vertices capacities in a normal flow-network.

The whole prupose of vertices capacities is certainly to limit the flow entering any given vertex by a number N. Hence, one could consider such a network G to be a normal flow-network G', where every capacity-capped vertex y is duplicated to form a pair of vertices (x', y) and where an edge E of capacity N is connecting vertices x' to y. Every edges entering y in G must be routed to x' in G' whereas edges leaving y in G' should stay untouched.

By doing so, we are forcing any flow entering the capacity-capable vertex y in G to flow into an alternate edge E of capacity N (which is in fact the vertex maximum capacity in G), therefore restricting the flow before it even enters the vertex y.

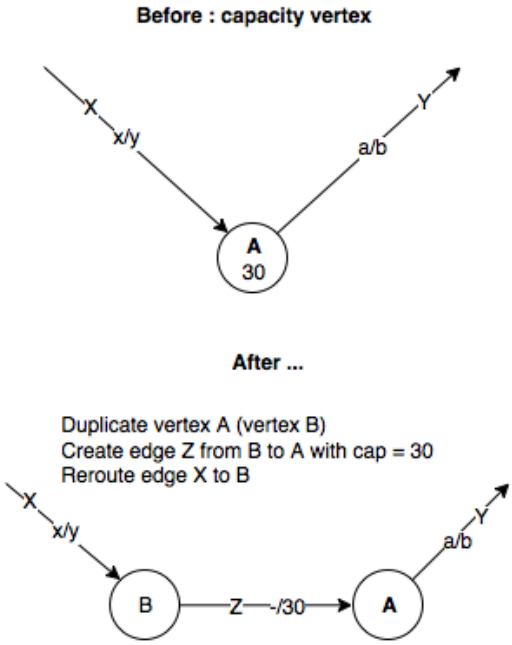
Considering

$$G = (V, E)$$

Using such a technique requires us to generate a flow-network graph G' containing $2|V|$ vertices (as we are duplicating every vertices in G — assuming that it is every vertex that is capacity-cappped) and $E+|V|$ edges (as we are effectively adding an edge for every capacity-capped vertices).

Since the resulting graph G' is a normal flow-network graph of comparable size, it follows that finding a maximum-flow is equivalent to solving an ordinary maximum-flow problem.

Here is an image illustrating the process :



(b) We can solve the escape problem by transforming the $n \times n$ undiretected graph that is the grid into a network-flow graph with edges and vertices capacities. Once the graph as been converted, finding a solution to the escape problem becomes an ordinary max-flow problem.

We convert the $n \times n$ undirected graph grid to the network-flow graph by the following steps :

1. We create a dummy *source* vertex that is connected to every starting point (i.e. the black vertices in the example) into source vertices;
2. We create a dummy *target* vertex that is connected by an undirected edge to every vertices for which $i = 1$, $i = n$, $j = 1$, or $j = n$;
3. For every vertices left in the $n \times n$ grid graph, we assign a vertex capacity of 1;
4. We convert every undirected edges $e = (u, v)$ in graph to a pair of two directed edges of maximum capacity 1 (i.e. $f' = (u, v)$ and $f'' = (v, u)$).

Now, to solve the escape problem and verify if there is, in fact, an escape for every starting point, we simply calculate the max-flow of G using any max-flow algorithm (such as Ford-Fulkerson).

$$x = max\_flow(G)$$

As every edge have capacity of 1, if the number x returned is strictly equal to the number of starting point (i.e. number of vertices in G), then there is a solution to the escape problem —every starting point was able to find an escape to the target t.

If, however, the returned value x is less than the number of vertices, then there is no solution to the escape problem —some of the starting points weren't able to flow properly to an escape point due to a capacity constraint (remember that capacities were at 1).

4 Blood transfusions

(a) To solve efficiently this problem, it is useful to think of it as a max-flow problem.

Specifically, we could design a graph G composed of 2 sets of vertices for all blood-type:

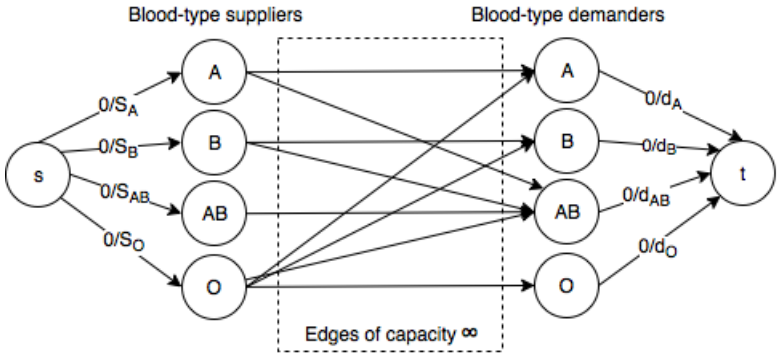
- a first set of vertices of all blood-types that are *supplying* a specific type of blood, which are connected to a dummy *source* vertex;
- a second set of vertices of all blood-types that are *demanding* a specific type of blood, which are connected to a dummy *target* vertex.

For each edges originating from the *source* s vertex towards a specific blood-type *supply* vertex x, we assign a capacity corresponding to the units of blood we currently have in supply (eg. capacity for edge from s to A-type blood-type supplier vertex would be s_A).

Similarly, we proceed similarly for the edges originating from the *demand* vertices to the *target* vertex, where the capacity is the units of blood that is projected for a specific blood-type (eg. capacity for edge from A-type blood-type demand vertex to the *target* vertex would be d_A).

We then create edges between every *supply vertices* of blood to every *demand vertices* for pairs whose signatures are compatible for blood transfusions.

Problem 8



The rest of the solution to the blood-transfusion problem is to trivially find the max-flow of such a graph G using any max-flow algorithm. To know whether there would be enough blood supplies for a projected demand, we simply look up the *demand to target* edges (for all blood-types) are saturated to capacity after running the max-flow algorithm on G.

Being a max-flow problem, we could potentially choose any max-flow algorithm to solve the problem efficiently. Hence, the worst case running time could vary. As an example, using the Ford-Folkerson algorithm would yied a worst-case running time of $O(U \cdot E^2)$, where U is the highest edge capacity.

(b) Firstly, we need to find an allocation (i.e. define a cut) of maximum-size in the above graph. One could consider a maximum cut consisting of the *source* as well as the A and B demand and supply vertices. This way, we are making sure to define a cut that is of maximum capacity, therefore maximizing the number of patients.

Such a cut yields a total of capacity (i.e. the total number of patients) of:

$$50 + 36 + 8 + 3 = 97$$

Hence, even when selecting a cut that maximizes the number of patients, the expected demand of 100 units of blood could not possibly be handled in its entirety.

A way to popularize the situation would be to look at the total number of demands for blood-type A and O and look at how those demands could be satisfied. In the current situation, there is a total of $45 + 42 = 87$ demands for both blood-type. However, when looking at the total supplies that could satisfy these demands (blood type A and O) there is only $50 + 36 = 86$ units to be used. Hence, the expected demand could not possibly be satisfied.

5 7.21

Given a graph G, one could simply execute the Ford-Fulkerson algorithm, computing the max-flow value of G and look at any edges whose capacity has been reached. Those kind of edges are limiting the max-flow of G, acting as bottleneck when trying to augment the flow along a certain path. However, it is not true that all edges at capacity are *critical edges*. Let e be an edge $\in G$ whose capacity has been reached. If a cycle containing a backward edge of e exists in the residual graph (of the Ford-Fulkerson algorithm execution), it would still be possible to push some flow along the cycle in order to reduce the flow in e; the conservation law of the network-flow graph G would still be satisfied. To check whether an edge is essentially critical or not, we simply have to check whether that any edges at capacity is not part of such a cycle. If it's not, that edge is *critical*.

To find edges that are *critical* within a graph G, one could design an algorithm following these steps :

(a) Find the max-flow of G using Ford-Fulkerson, and keep the residual graph G';

(b) For each edge $e(x, y) \in G$ at capacity, check whether there is a path in G' from x to y;

(c) If there is a path from x to y in G', then e is not critical;

(d) If no path exists from x to y in G', then e is effectively a *critical* edge.

The main parts of the algorithm can be implemented using standard algorithms such as Ford-Fulkerson (for determining a max-flow in G) and breath first search (to find a s-t path in G). This algorithm can be such that its worst case running time enters our definition of efficient algorithms (poly-time algorithms), if designed properly. Overall, if breath-first search is used, the running time of the algorithm would be dominated by the Ford-Fulkerson algorithm which is $O(U \cdot E^2)$ where U is the largest capacity.

6 7.22

While it is possible that the calculation of the maximum flow with the erroneous edge had an impact on the maximum flow, it could also be the case that it didn't. As mentionned earlier in 7.21, if this edge —let's call it e— is part of a cycle (also note that e is *critical* along that path), this error wont have any impact on the maximum flow as it would be possible to redirect the flow through another edge that is part of a directed cycle. If no directed cycle can be found, we can create one by adding an edge from s to t of capacity 1 and executing a depth first search algorithm to find a path from u to v in the residual graph. Once we have found such a path, we can reduce the flow in each edge along

that path by 1. At this point, it is true that the flow could be increased by one along some other path. However, it is possible to run one iteration of the Ford-Fulkerson algorithm to find it.

All of those operations take linear time, where we basically need to iterate over all vertices (depth-first search) and every edges (reducing flow by one) in the worst case, yielding the following worst case running time :

$$O(|V| + |E|)$$

7 7.23

Finding a maximum vertex cover would be fairly easy in this situation. However, in this problem, we show that it is possible to use the minimum cut of a network-flow graph G to find a *minimum* vertex cover.

Finding any vertex cover essentially boils down to considering if an edge is or not, part of a certain vertex cover. Similarly, when considering a specific cut in a network-flow, we are essentially considering whether each vertices are more on the source's side of the graph or on the target's side. Hence, it makes tons of sense to make use of a network-flow graph for this problem: we have a 1-to-1 representation of every vertices and deciding on a specific cut is fairly similar to determining a vertex cover. Also, one could use these cuts to represent a specific vertex cover in such a way that the capacity of the cut is identical to the size of the vertex cover. Therefore, finding a *minimum* vertex cover would become fairly trivial i.e. finding a minimum cut in network-flow graph G.

We define 2 different sets of vertices X and Y, where the vertices in X are considered to be on the source's side and the ones in Y on the target's side. For each edges from source s to vertices in X, we assign a capacity of 1. We proceed similarly for each edges from Y going to target t. Hence, any cut capacity would represent the number of selected edges i.e. the size of a vertex cover. For any cut (U, V) in G, a possible vertex cover Z would be defined as follows :

$$Z = \{e \in U\} \cup \{e \in V\}$$

All of this is still assumptions at this point, though. We currently do not make sure that every edges in Z is, in fact, touching every vertices —i.e. valid vertex cover by definition. To do so, we create edges of capacity ∞ between the two set of edges X and Y. This makes sure that every minimum cut will effectively represent a valid vertex cover by definition —and also the *minimum* vertex cover as expected.

We have just shown that is it possible to reduce the minimum vertex cover problem to a minimum cut problem by using a bipartite graph.