CENG 567 Design and Analysis of Algorithms
Homework 4

Ardan Yılmaz 2172195

December 2022

1 Q1. Burning Calories

Given 2 sets of exercises with how many calories they burn, ie, low: $l_1, l_2, ..., l_n$ and high-level exercises: $h_1, h_2, ..., h_n$, such that $0 < l_i < h_j$ for $\forall i, j \in \{0, n\}$, where l_i represents how many calories burnt when low-level exercise is done on day i. (Similarly for h_i). If a low-level exercise is chosen, any exercise might follow. However, a high-level exercise must proceed with a rest day.
Design a dynamic programming algorithm that takes values for $l_1, l_2, ..., l_n$ and $h_1, h_2, ..., h_n$ and returns the plan of maximum calories.

1. **Algorithm Design**
   An nx2 DP table is kept to store both the scores and the exercises up to that date to achieve that score for each day. There are four possible moves on the table proceeding to the next day:
   (1) a downward move from the first column indicating a low exercise after another low one or a rest,
   (2) a downward move from the second column indicating a rest day,
   (3) a move from the first column to the second in the next row (day), indicating a high-level exercise after either a rest day or a low-level exercise,
   (4) and a rest move from column 1 to 0 in the next row (day).
   Each entry in the table is filled considering these moves, ie, there are two possible choices for each box, except the initial row (filled with l_0 and h_0). These two choices are considered for each entry and the one maximizing the score is chosen. Alongside the score update, the choice of exercise is either appended to the previous ones (if not a rest). Finally, in the final row, we return to the exercises with the max score.

Algorithm 2 Burning_Cals($l_1, ..., l_n,\ h_1, ..., h_n$)

```
$l \leftarrow l_1, ..., l_n$
$h \leftarrow h_1, ..., h_n$
$Cals[n][2]$ //nx2 DP array items of type:$< score >:< exercises : [..] >$
$Cals[0][0] = \{l[0] : [”l_0”]\}$
$Cals[0][1] = \{h[0] : [”h_0”]\}$
for i in range(1,n) do
    if Cals[i-1][0].score + l[i] $\geq$ Cals[i-1][1].score then
        Cals[i][0]={Cals[i-1][0].score + l[i]:Cals[i-1][0].exercises.append(”l_” + str(i))}
    else
        Cals[i][0] = {Cals[i-1][1].score : Cals[i-1][1].exercises}
    end if
    if Cals[i-1][0].score + h[i] $\geq$ Cals[i-1][1].score then
        Cals[i][1]={Cals[i-1][0].score+h[i]:Cals[i-1][0].exercises.append(”h_”+str(i))}
    else
        Cals[i][1] = {Cals[i-1][1].score : Cals[i-1][0].exercises}
    end if
end for
if Cals[n][0].score $\geq$ Cals[n][1].score then
    return Cals[n][0].exercises
else
    return Cals[n][1].exercises
end if
```

2. **A worked example**
   Say, the following lists of low and high-level exercises are given: $l = [3, 2, 1]$, $h = [4, 5, 4]$.

$3, [l_0]$	$4, [h_0]$
$5, [l_0, l_1]$	$7, [l_0, h_1]$
$7, [l_0, h_1]$	$9, [l_0, l_1, h_2]$

The following are the steps taken to fill in the Cals table:
1. Initialization step:
$Cals[0][0] = \{l[0], [l_0]\} = \{3, l_0\}$
$Cals[0][1] = \{h[1], [h_0]\} = \{4, h_0\}$

2. Decide Cals[1][0]
This box is to represent a program either for $[l_0, l_1]$ or $[h_0, rest]$ $(Cals[0][0].score + l[1]) = 5 > 4 = (Cals[0][1].score)$, hence,

$Cals[1][0] = \{5, [l_0, l_1]\}$

3. Decide Cals[1][1]
This box is to represent a program either for $[l_0, h_1]$ or $[h_0, rest]$
$(Cals[0][0].score + h[1]) = 7 > 4 = (Cals[0][1].score)$, hence,
$Cals[1][1] = \{7, [l_0, h_1]\}$

4. Decide Cals[2][0]
This box is to represent a program either for $[l_0, l_1, l_2]$ or $[l_0, h_1, rest]$
$(Cals[1][0].score + l[2]) = 6 < 7 = (Cals[1][1].score)$, hence,
$Cals[2][0] = \{7, [l_0, h_1]\}$

5. Decide Cals[2][1]
This box is to represent a program either for $[l_0, l_1, h_2]$ or $[l_0, h_1, rest]$
$(Cals[1][0].score + h[2]) = 9 > 7 = (Cals[1][1].score)$, hence,
$Cals[2][1] = \{9, [l_0, l_1, h_2]\}$

And finally $Cals[2][1] > Cals[2][0]$, so sequence $[l_0, l_1, h_2]$ is returned.

3. **Proof (By induction on the number of days)**

   (a) **Base Case: n=2** (n=1 is trivial to show)
   We are given l_0, l_1 and h_0, h_1. And possible sequences are: (1) $[l_0, l_1]$, (2) $[l_0, h_1]$, and (3) $[h_0, rest]$. (Those starting with a rest day are excluded, as starting with a lower-level exercise is clearly a better option).
   The given algorithm considers all these three cases and returns the one with the max score.

   (b) **Inductive Hypothesis:** Assume this algorithm works for up to n days.

   (c) **Inductive Step:** Show it also works for n+1 days.
   As per the inductive hypothesis, the algorithm fills the DP table correctly up to n rows. Then it considers possible cases for the $(n+1)^{st}$ row. For this, three cases need to be considered:
   1. (optimal solution for n-1 days) + l_n + h_{n+1}
   2. (optimal solution for n-1 days) + h_n + $rest$
   3. (optimal solution for n-1 days) + l_n + l_{n+1}

   And, as the algorithm considers all these cases, it finds the maximum.
   This completes the proof.

4. **Analysis:** As the inner for loop runs for n times and all other operations, interior or exterior to loop, are constant. Hence, this is an O(n) algorithm. **[1]**

2 Q2. 2D Sheet Cutting

Consider an m × n sheet of metal where m, n are positive integers. You are given an array P with a real-valued price P(i, j) for any integer-dimension rectangle of size i × j. Assume P(i, j) = P(j, i). You want to cut the sheet maximizing the total price of the resulting pieces. You can cut any piece, horizontally or vertically, with the cut going completely across. Design an algorithm that indicates how to cut for maximum total price, producing instructions if the form "Cut a i × j piece at x = k (or at y = k)".

Consider a sheet of size 2 by 2 and the function Cut(n, m) that calculates the max price for a n by m sheet. Cut(n, m) considers all possible cuts of the sheet and makes the corresponding calls to itself, recursively. An example of repeating sub-problems can be seen as a tree in the worked example (2-j).

1. **Algorithm Design**
   As per the repeating sub-problems, we will take a dynamic programming approach, with an (n+1) by (m+1) DP table whose $(i, j)^{th}$ entry is to hold the max price of the cut of size i x j. The max price for a cut of a particular size is calculated considering possible vertical and horizontal cuts, then the optimal cut is recorded in the DP table. Thus, on the request of a sub-solution, the optimal answer recorded in the DP table is to be used. Additionally, the optimal cut for each sub-problem is recorded in the DP table, as well.

Algorithm 4 SheetCutting(prices, n, m)

```
DP[n+1][m+1] //dp table
DP[*][*] = 0 //init all entries to 0
for i in range(1,n+1) do
    //Processing only the upper-half of the matrix is enough
    for j in range(i,m+1) do
        _max = Prices[i][j]
        opt_cut = None
        //find the optimal cut for ixj sized sheet horizontally
        for k in range(1, i+1) do
            if DP[k][j] + DP[i-k][j] > _max then
                _max = DP[k][j] + DP[i-k][j]
                opt_cut = Cut a i × j piece at x = k
            end if
        end for
        //find the optimal cut for ixj sized sheet vertically
        for k in range(1, j+1) do
            if DP[i][k] + DP[i][j-k] > _max then
                _max = DP[i][k] + DP[i][j-k]
                opt_cut = Cut a i × j piece at y = k
            end if
        end for
        DP[i][j] = {$\_max, opt\_cut$}
        DP[j][i] = {$\_max, opt\_cut$} //using the symmetry
    end for
end for
Now trace back the DP table to print where to cut
Until m or n reaches down to 1 or opt_cut = None, pop the stored opt_cut
at DP[n][m]. Go to the ideal cuts, ie, update m = m-k if opt_cut reads "Cut
a i x j piece at y = k"; else update n = n-k. There will be 2 different prices
for each cut.
```

A simple modification in the for loops to look for possible cuts may be done. Variable k does not need to try all possible values of i, it can go up to half of i and j. (Due to symmetry in the Prices table).

2. **A worked example**

Suppose we have a 2x2 sheet, and the value of each piece is given in the following table:

0	0	0	0
0	10	15	20
0	15	15	25
0	20	25	10

The following are the steps taken to fill in the DP table.

(a) Allocate a DP[n+1][m+1] table

(b) Initialize all elements in row 0 and col 0 to 0

(c) DP[1][1] = $\{Prices[1][1], None\}$ //No option to cut a 1x1 sheet

(d) DP[1][2] = $max\{DP[1][1] + DP[1][1], Prices[1][2]\}$ = {20, cut a 1x2 piece at y=1 } //can either cut the sheet into 2 1x1 pieces or leave it.

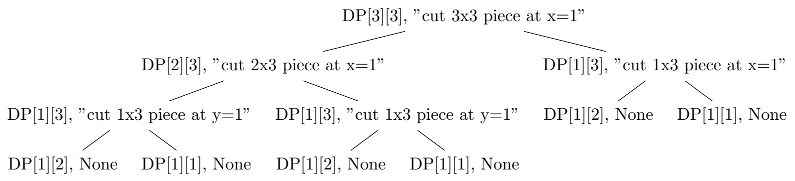
(e) DP[1][3] = $max\{DP[1][2] + DP[1][1], Prices[1][3]\}$ = {30, cut a 1x3 piece at y = 1 }

(f) DP[2][2] = $\{DP[2][1] + DP[2][1], DP[1][2] + DP[1][2], Prices[2][2]\}$ = {30, cut a 2x2 piece at y=1 }

(g) DP[2][3] = $max\{DP[2][2]+DP[2][1], DP[2][1]+DP[2][2], DP[1][3]+DP[1][3], Prices[2][3]\}$ = {60, cut a 2x3 piece at x=1}

(h) DP[3][3] = $max\{DP[3][2] + DP[3][1], Prices[3][3]\}$ = {90, cut a 3x3 piece at x=1}

(i) **Tracing back, we arrive at the following tree:**


DP[3][3], "cut 3x3 piece at x=1"
DP[2][3], "cut 2x3 piece at x=1"
DP[1][3], "cut 1x3 piece at x=1"
DP[1][3], "cut 1x3 piece at y=1"
DP[1][3], "cut 1x3 piece at y=1"
DP[1][2], None
DP[1][1], None
DP[1][2], None
DP[1][1], None
DP[1][2], None
DP[1][1], None

Reading the above tree using topological order gives the desired output.

3. **Proof (by contradiction):** Say tracing DP[n][m] back does not give an optimal solution, ie, there exists another sequence of cuts with a higher price. For this to happen, there needs to be at least one $(i, j)^{th}$ element in the DP table whose cut at a different index yields greater price. This can never be the case, as we look at all possible cuts and prices table. Hence, we arrive at a contradiction.

4. **Run time analysis:**
The most costly operation for this algorithm is the for loops. The following equation gives the time complexity (where C is a constant):

$\sum_{i=1}^{n+1} \sum_{j=1}^{m+1} (\sum_{k=1}^{j+1} C + \sum_{k=1}^{i+1} C) =$

$\sum_{i=1}^{n+1}\sum_{j=1}^{m+1}\sum_{k=1}^{j+1} C + \sum_{i=1}^{n+1}\sum_{j=1}^{m+1}\sum_{k=1}^{i+1} C =$

$\sum_{i=1}^{n+1}\sum_{j=1}^{m+1} O(Cj) + \sum_{j=1}^{n+1}\sum_{i=1}^{m+1} O(iC) =$

$\sum_{i=1}^{n+1} O(m^2) + \sum_{i=1}^{m+1} O(n^2) = O(mn^2 + nm^2)$

3 Q3. Escape Problem

Consider a directed graph G = (V,E) and two disjoint sets of nodes X, S ⊂ V. A set of evacuation routes is a set of paths so that (i) each node in X is the start of one path, (ii) each path ends at a node in S, and (iii) the paths do not share any edges.

a **Show how to decide in polynomial time whether a set of evacuation routes exists.**
This problem is similar to the disjoint-edge paths problem. To reduce this problem so that the known Ford-Fulkerson algorithm can be used, the following steps should be taken:
1. Add a source node, s, and for each node $v \in X$, add an edge (s,v) with capacity 1
2. Add a sink node, t, and for each node $v \in S$, add an edge (v,t) with capacity 1.
3. Given graph G=(V, E) is converted into $G' = (V', s, t, E', c)$ such that $V' = V \cup \{s, t\}$, $E' = E \cup \{(s, v)$ for $\forall v \in X\} \cup \{$(v, t) $for\ \forall v \in S\}$
4. Running the FF algorithm, if the max flow $\geq$ number of paths ($|X|$ or $|S|$), then there exists a set of evacuation routes. This is an $O(Cnm) = O(nm)$ operation. (where C the the edge with the max capacity and 1 in this case).

b **Do the same when (iii) reads "the paths do not share any nodes"**
To ensure that no paths share a node, ie, a node is passed through only once, each node is duplicated and for each duplication, an edge with capacity 1 is added in between. Thus, if there is a flow passing through one node, there will be more augmenting paths passing from that node in the residual graph. The original edges between the nodes need to be altered accordingly. For each node $v \in V$, we have generated a node couple v, v', and need to adjust the original edges. Say we have, $v_1, v_2 \in V$ and $(v_1, v_2) \in E$, that is, in the adjusted graph v_1, v_1' and $v_2, v_2' \in V'$, and we need to add the following edge (v_1', v_2). To fully reduce this problem into one where we can use the FF algorithm, we add two more nodes: source s and sink t. Then, we need to add an edge between the source and each node in S, and an edge between each node in X and sink t.

More formally, from the given graph G = (V, E), G' = (V', E', s, t, c) is constructed. Running the FF algorithm on G', we can decide the existence of a set of evacuation routes in polynomial time. If the max flow generated is greater than or equal to the number of paths, there exists a set of routes. The following are the formal definitions of steps taken to construct G' from G.

1. $V' = \{$ add v' for $\forall v \in V\} \cup \{s, t\}$
Also, define an auxiliary set to store the duplicated node pairs, DN. $DN = \{(v, v')$ for $\forall v \in V\}$

2. $E_1 = \{u$ for $\forall u \in DC\}$ //Adding an edge between all couple pairs (only between the couples)

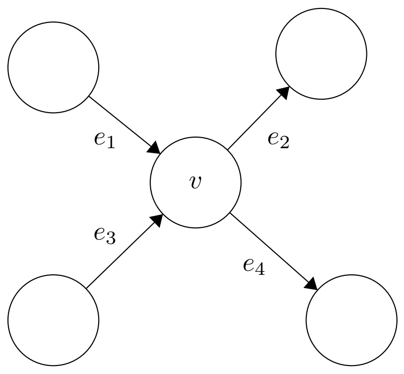
3. $E_2\{(u', v)$ for $(u, u'), (v, v') \in CN$ x CN if $(u, v \in E)\}$ // Adding the original edges. If there exists an edge between nodes v_1 and v_2, then for the coupled nodes, $(v_1 v_1')$ and $(v_2 v_2')$, we add an edge from v_1' to v_2 to ensure each node is used once.

4. $E' = E_1 \cup E_2 \cup E$

5. $c = \{1$ for $\forall e \in E\}$

Running the FF algorithm on G', one can decide the existence of such a path set in O(Cnm) = O(nm). (where C is the capacity of the edge with the max capacity).

c **Give an example with the same G, X and S when (a) is possible but (b) not.**



Say we have the above graph, two evacuation paths: p_1 and p_2, where $e_1, e_4 \in p_1$ and $e_3, e_2 \in p_2$. Evacuation routes p_1 and p_2 do not violate conditions in (a), ie, they do not share any edges. However, they violate conditions in (b), ie, they share node v. **[1]**

4 Q4. Enrollment Problem

There are n students in the CS program, and each of them must take the Algorithms course. There are m sections of Algorithms offered: A1, ..., Am. They begin at times T1, ..., Tm. Each student can sign up for at most one section. Because students also have other courses to take, each student can only attend some subset of the Algorithms sections. This varies by student: some may only be free during one or two sections, while others might be able to attend all of them. In addition, each section Ai has a capacity Ci on the total enrollment, so you cannot put more than Ci students in that section. Your goal is to assign students to sections so that as many students as possible are enrolled in a section of Algorithms (note that it might be impossible to enrol everybody into a section, this is fine). Design an algorithm to solve this problem by using network flows. Note that it is not enough to only describe the graph to solve the problem. You should write an algorithm to describe how to use that graph to solve the given enrollment problem.

This is a problem of max cardinality bipartite matching of student-section sets, ie, maximizing the number of students to take the Algorithms course. This is the well-known bipartite matching problem, where students form one set of nodes and the sections form the other. To solve this problem, we need to form a graph to represent the semantics behind the problem and then reduce it into a network, where we can use Ford-Fulkerson's (FF) algorithm. For this, the following steps should be taken:

1. Create n nodes for each student
2. Create m nodes for each section
3. For each student node v, add an edge from v to all the sections that fit their schedule.

   Up to now, the graph is formed. Now we need to reduce it to a suitable network where the FF algorithm can be used.

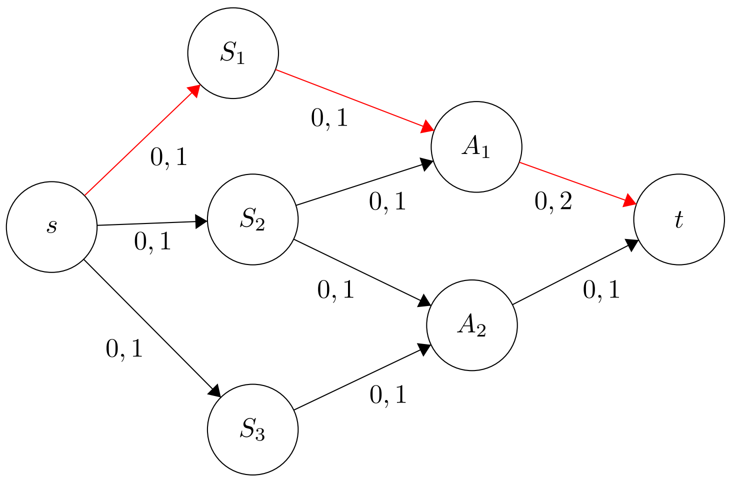
4. Add a source node s
5. From s, add an edge to each student with capacity 1 (a student can only be enrolled in one section)
6. Add a sink node t
7. From each section node, add an edge with a capacity equal to that section's capacity. (To enforce the capacity constraint of the section).
8. Run the FF algorithm on this network to compute the max flow.

1. **Algorithm Design:** The following is the pseudo-code for the algorithm to find a max cardinality matching, given a set of students, sections and their begin times.

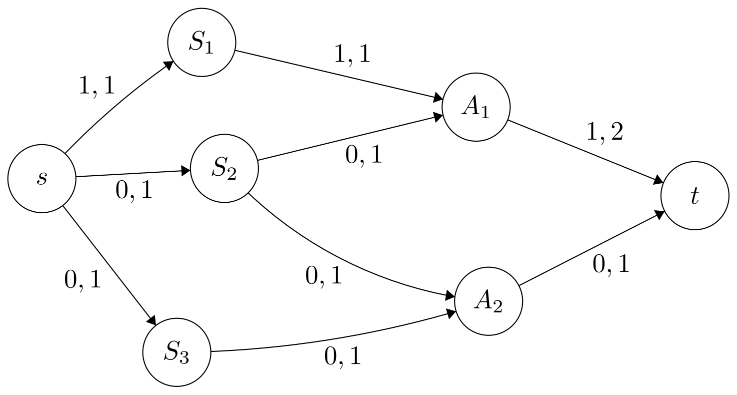
   - Create a network as described above
   - Run FF algorithm on that network
   - Based on the residual network computed in the last iteration, get the matchings. Each edge e from sections to students with a capacity of 1 is a match.

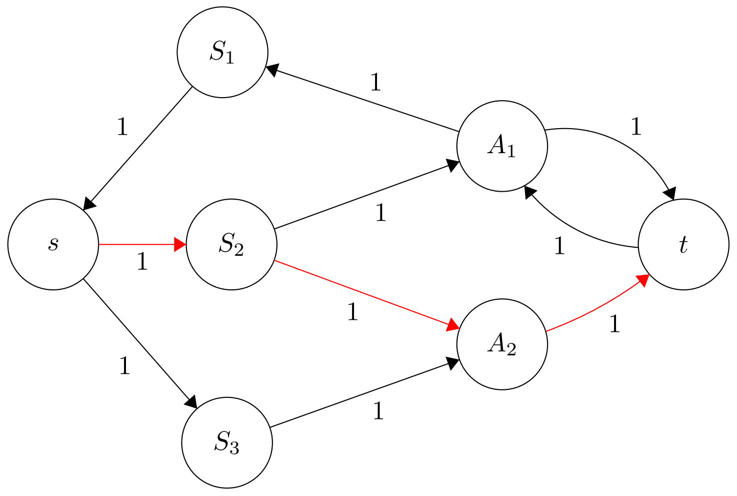
2. **A worked example:** Say we are given three students: S_1, S_2, S_3, two sections of algorithms: A_1, A_2, with capacities 2 and 1, respectively. And, we know the following: (1) S_1 can take A_1, S_2 can take A_1, and S_3 can only take A_2.

   - **The following is the initialization of the network.** Considering only the capacities on edges, the residual network is the same as the initialization step. And the path in red is chosen to be augmented.

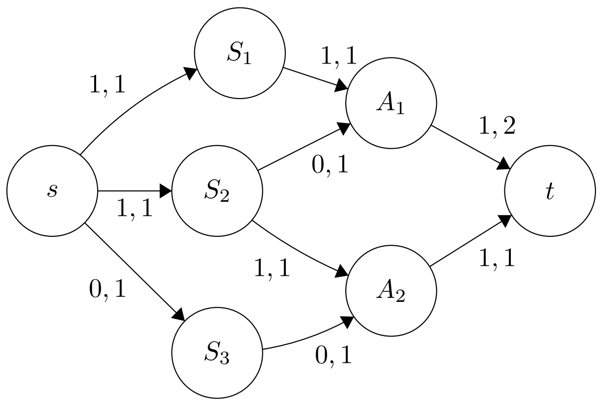


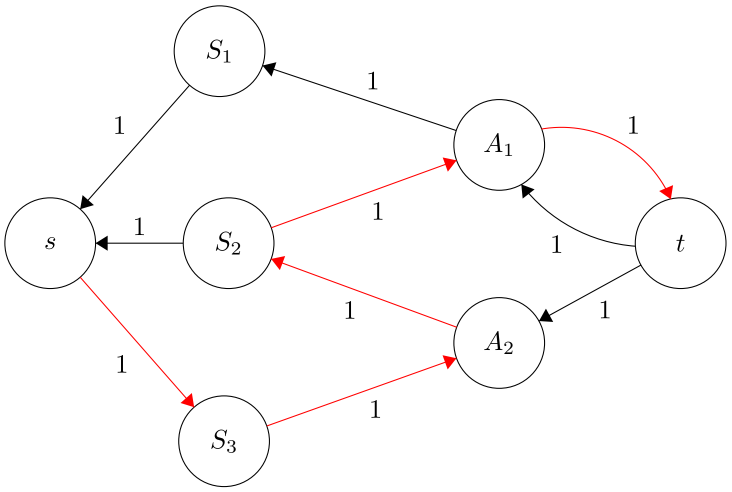
   - **The resulting network and the corresponding residual graph (next path to augment is in red) from this augmentation:**



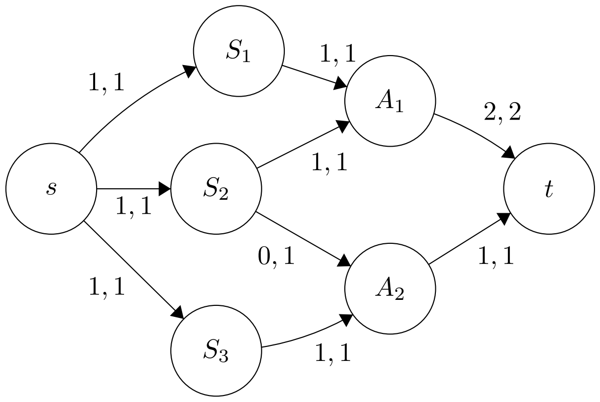


- **The resulting network and the corresponding residual graph (next path to augment is in red) from this augmentation:**

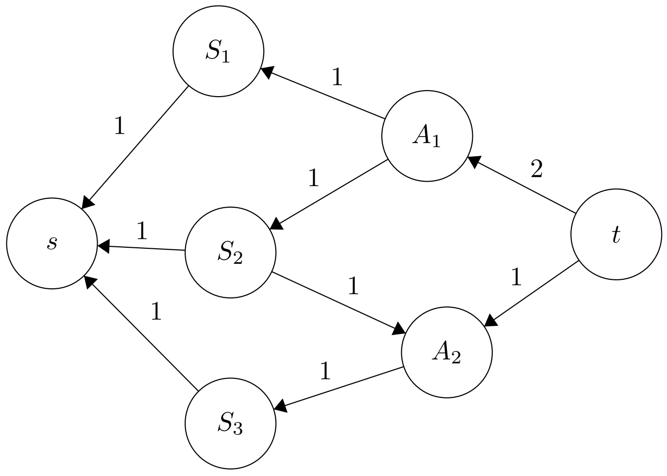




- S_2 **was initially assigned to** A_1**, however, pumping a flow in the reverse direction,** S_2 **is assigned to** A_2**. The following is the resulting network:**



- **And the corresponding residual graph is:**



- **There is no s-t augmenting path, so the FF stops.** Now checking the edges from algorithms to students, we have (A_1, S_1), (A_1, S_2), and (A_2, S_3). Hence, S_1 and S_2 are enrolled in A_1 and S_3 is enrolled in A_2.

3. **Proof of Correctness:** As there might be multiple max cardinality matching of student-section sets, showing FF finds the max cardinality matching when input a network formed as described above suffices. This is a two-way proof: (1) max cardinality matching = max flow found by FF, and (2) the other way around. The following proof is taken from slides. [2]

   (a) **Max cardinality matching = max flow found by FF**
      i. Given max matching M of cardinality k.
      ii. Consider flow f that sends 1 unit along each of k paths.
      iii. f is a flow and has cardinality k.

   (b) **max flow found by FF = Max cardinality matching**
      i. Let f be a max flow in G' of value k.
      ii. Consider M = set of edges from Students (S) to Sections (A) with f(e) = 1.
      iii. Each node in S and A participates in at most one edge in M
      iv. $|M| = k$

4. **Run time analysis:** The pre-processing step to produce and convert the graph into a suitable one is an $O(|S||A|)$ operation. (Checking suitability of each section for each student). The FF algorithm is an $O(Cmn)$, where C = max capacity of any section, n = $|S| + |A| + 2$, and $m \leq |S||A|$. And finally, checking edges from sections to students is an $O(m)$ operation in the worst case (which it never is). These three are sequential operations, and the most expensive one is the FF algorithm, with $O(Cnm)$ complexity.

5 REFERENCES

[**1**] We have exchanged ideas with Yalgınay Yaltırık and Deniz Germen.
[**2**] Lecture slides on NETWORK FLOW