(1) Formulate MCF,
send 1 unit of goods from $s \rightarrow t$

We can frame this problem as a linear program:

$$\text{minimize } f(\vec{x}) \text{ s.t. } \begin{cases} A\vec{x} \leq b \\ A_{eq}\vec{x} = b_{eq} \\ lb \leq \vec{x}_n \leq ub. \end{cases}$$

Our objective function, $f(\vec{x})$:

$$f(\vec{x}) = 8x_1 + 6x_2 + x_3 + 3x_4 + 5x_5 + x_6 + 5x_7 + 3x_8$$

Our constraints:

$$\begin{aligned} x_1 + x_2 &= 1 \\ x_6 + x_7 + x_8 &= 1 \\ x_3 + x_4 - x_1 &= 0 \\ x_6 - x_3 &= 0 \\ x_5 + x_8 - x_4 &= 0 \\ x_7 - x_2 - x_5 &= 0 \end{aligned}$$

Thus, let:

$A_{eq} =$

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	b_{eq}
1	1	0	0	0	0	0	0	1
0	0	0	0	0	1	1	1	1
-1	0	1	1	0	0	0	0	0
0	0	-1	0	0	1	0	0	0
0	0	0	-1	1	0	0	1	0
0	-1	0	0	-1	0	1	0	0

And let: $A = [\,]$ (no inequalities here).

$b = [\,]$

$lb = \vec{0}$ → (can send 0 or more goods)

$ub = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$ → (cannot send more than 1 good)

On the following page is the MATLAB code written to satisfy these constraints, using *linprog* as our solver.

As a result, we receive the least cost path found:

$$X_1 \longrightarrow X_3 \longrightarrow X_6 = \$10.$$

The MATLAB code for Problem 1

```
clc, clearvars

% the objective function to minimize
f = [8, 6, 1, 3, 5, 1, 4, 3];

% exact equivalencies to constrain
Aeq = [
    1, 1, 0, 0, 0, 0, 0, 0;
    0, 0, 0, 0, 0, 1, 1, 1;
    -1, 0, 1, 1, 0, 0, 0, 0;
    0, 0, -1, 0, 0, 1, 0, 0;
    0, 0, 0, -1, 1, 0, 0, 1;
    0, -1, 0, 0, -1, 0, 1, 0
];

beq = [
    1;
    1;
    0;
    0;
    0;
    0
    ];

% less than or equal to amounts to constrain
A = [];
b = [];

% our bounds, and in this case, we are only sending through 1 unit, so the
% upperbound is just a one vector, the lower is zeros.
lb = zeros(8, 1);
ub = ones(8, 1);

% feed matrices & vectors to lin prog and print solution
sol = linprog(f, A, b, Aeq, beq, lb, ub)
```

(2) 2 Vendors: Apple & Dell

Constraint: numComputers ≥ 10 (at least 10)

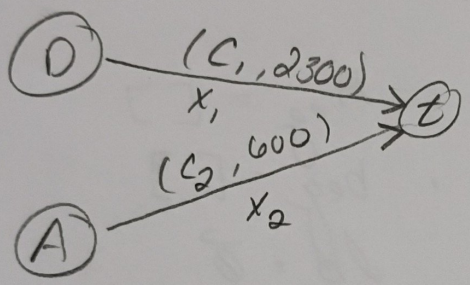
Dell: $2300, Apple: $600

Constraint: totalCost ≤ 10000

Constraint: number from 1 vendor cannot exceed 2* the other

* MINIMIZE COST

The network can be visualized as:



where
- D → Dell (source node)
- A → Apple (source node)
- t → UNM (sink node)
- C_1, C_2 → capacities ≥ 1 that can be sent along edges x_1, x_2 respectively.

We can frame this problem as a linear program:

$$\text{minimize } f(\vec{x}) \text{ s.t. } \begin{cases} A\vec{x} \le b \\ Aeq\vec{x} = beq \\ lb \le \vec{x} \le ub. \end{cases}$$

Our objective function, $f(x)$:

$$f(x_1, x_2) = 2300x_1 + 600x_2.$$

Our constraints:

Cost: $2300x_1 + 600x_2 \le 9999.99$ (< 10000)

Cap: ↓

Capacity: $x_1 + x_2 \geq 10 \Rightarrow -x_1 - x_2 \leq -10$ (need at least 10 comps)

$x_1 \leq 2x_2 \Rightarrow x_1 - 2x_2 \leq 0$

$x_2 \leq 2x_1 \Rightarrow x_2 - 2x_1 \leq 0$ (cannot exceed 2 * the other).

For bounds on x:

let $lb = \vec{0}$ (needs to be a positive amount, at least 0).

let $ub = [\,]$ (no upper bound pertinent to either)

Thus, let:

$$A = \begin{bmatrix} x_1 & x_2 & b \\ 2300 & 600 & 9999.99 \\ -1 & -1 & -10 \\ 1 & -2 & 0 \\ -2 & 1 & 0 \end{bmatrix}, \quad A_{eq} = [\,], \quad beq = [\,], \quad lb = \vec{0}, \quad ub = [\,]$$

On the following page is the Matlab code written to satisfy these constraints, using linprog as our solver.

There are some issues when using linear programming for this problem.

(1) Given the constraints above, there is no solution under $10,000.

(2) When the $10,000 constraint is removed, the optimal solution is:

3.33 from Dell

6.67 from Apple

And in this case, we cannot split the units we are sending. So, our only option here is to send 347 (violates are no 2-times-the-other-company constraint) or 446 (valid) to get an approximate optimal solution.

So, in case ①, the cause of the issue is that we have to enact the $x_a \leq 2x_b$ constraint on capacity. Without this, the obvious optimal solution is to just buy Apple computers. In case ② we can find an optimal solution (not in our price range), but we have to round figures up or down and make compromises since we can't ship 1/3 of a computer (at least I hope not.)

The MATLAB Code for Problem 2

```
clc, clearvars;

% the objective function to minimize
f = [2300, 600];

% exact equivalencies to constrain
Aeq = [];
beq = [];

% less than or equal to amounts to constrain
A = [
    2300, 600; % this constraint makes it so there are no solutions at all
    -1 -1;
    1, -2;
    -2, 1;
    ];

b = [
    9999.99; % this constraint makes it so there are no solutions at all
    -10;
    0;
    0;
    ];

% upperbound is technically nothing, the lower is zeros.
lb = zeros(2, 1);
ub = [];

% feed matrices & vectors to lin prog and print solution
sol = linprog(f, A, b, Aeq, beq, lb, ub)
```

(3) Card flipping to find a local min $o(n)$.

(a.) Small o means "strictly less than", or $o(n) \Rightarrow \lim_{n \to \infty} \frac{f(n)}{n} = 0$.
Therefore $o(n)$ means we need an algorithm run time that grows asymptotically less than linear time.

Two increasing functions of n that are $o(n)$:

$\log n$, $\sqrt{n}$.

(b.) Strategy to find a minimum $o(n)$:

A = cards (to be referenced as an array for clarity)

(1) Let: n = number of cards
l = 0
r = n − 1
min = A[l]

(2) while (l <= r)
// calculate a mid point
$mid = \left\lfloor \frac{l+r}{2} \right\rfloor$

if ((mid == 0 || A[mid] ≤ A[mid − 1]) &&
(mid == n − 1 || A[mid] ≥ A[mid + 1])
min = A[mid] // we have found a local min.
break // stop immediately, we only want 1.
else if (mid > 0 && A[mid − 1] < A[mid]
r = mid − 1 // move left to smaller neighbor
else
l = mid + 1 // smaller neighbor to right.

The strategy: by using a binary search tactic, we can find a local minimum among st the cards $O(n)$ → or, more specifically, $O(\log n)$.

The strategy works as follows:

1. find the middle card in our current range.
2. Flip that card, and flip that card and the left and right cards as needed (at most 3 flips).

2. Case 1: our middle card is a minimum: $A[mid-1] \geq A$

$A[mid-1] \leq A[mid] \leq A[mid+1]$

generally for non-end cards.

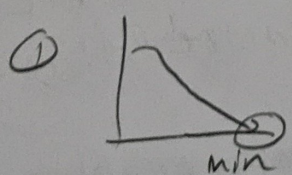
Case 2: our middle card is NOT a minimum.

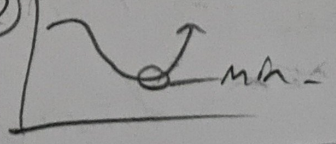
↳ From here, we will shrink the problem size in half in the direction of the smaller 2 neighbors.

↳ by making this choice we have 2 possibilities:

1. the numbers will continue decreasing, making the endpoint the guaranteed local min.
2. the numbers will start increasing, which will guarantee a minimum found in the "valley" so to speak.

(visually):





(4.) Directed graph $G(V, E)$,

$s, t \in V$, all other nodes $v_0, v_1, \ldots v_n \in V$

Find the shortest $s \to t$ path with linear programming.

$$\text{minimize } f(\vec{x}) \text{ s.t. } \begin{cases} A\vec{x} \leq b \\ A_{eq}\vec{x} = b_{eq} \\ lb \leq \vec{x} \leq ub \end{cases}$$

Let our $\vec{e}$ signify all edges, $e_1, e_2 \ldots e_n \in E$

Let $\vec{c}$ signify the cost at each corresponding edge in $\vec{x}$.

Therefore, our objective function is:

$$f(\vec{x}) = \vec{c} \cdot \vec{e} = c_1 e_1 + c_2 e_2 + \ldots + c_n e_n.$$

We can formulate the shortest path problem as a min cost flow problem (solveable with linprog) by using the above objective function to minimize and only send through

1 UNIT OF GOODS.

The constraints we must set on capacity/flow are:

→ the capacity out flow from s MUST EQUAL 1

→ the capacity in flow to t MUST EQUAL 1.

→ the in flow and out flow at each intermediate node v MUST BE EQUIVALENT.

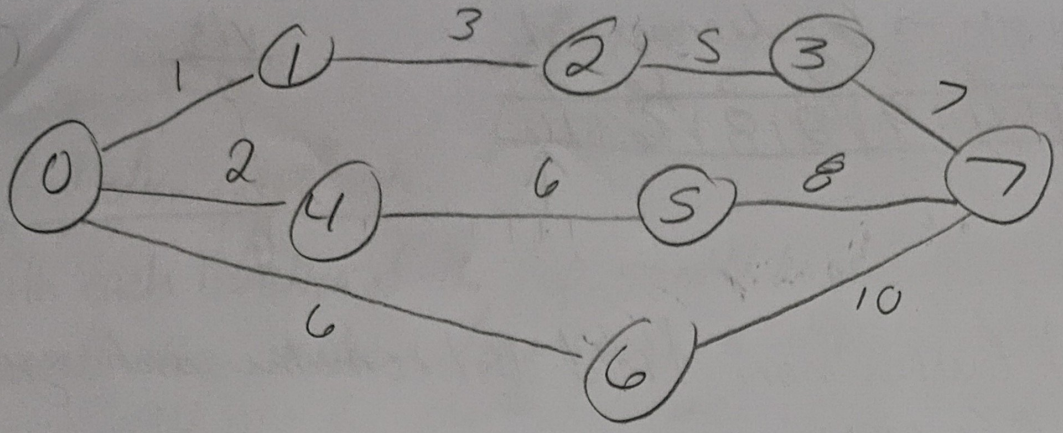
And lastly, our lb & ub will be constrained so that:

$$lb = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} = \vec{0} \qquad ub = \begin{bmatrix} 1 \\ 1 \\ \dots \\ 1 \end{bmatrix}$$

can send 0 or 1 goods through any edge, never more or less.

By limiting the unit of goods sent through the graph G from $s \to t$, the only point becomes on finding the "cheapest" possible route from $s \to t$ for 1 traveller, so to speak, which can be translated to the shortest possible path!

5.



Dijkstra.

★ Main induction hypothesis:

Let v be a visited node. Let u be an unvisited node. dist[v] is the shortest distance from source node to v. For each dist[u], the shortest distance from u is through the visited nodes only (or ∞ if unreachable).

① dist =

0	1	2	3	4	5	6	7
0	∞	∞	∞	∞	∞	∞	∞

vis: (empty)

② Remove Min → 0; decrease all dist as necessary.

dist =

0	1	2	3	4	5	6	7
0	1	∞	∞	2	∞	6	∞

vis: 0

③ Remove Min → 1; decrease all dist as necessary

dist =

0	1	2	3	4	5	6	7
0	1	4	∞	2	∞	6	∞

vis: 0, 1

④ Remove min → 4; decrease dist

dist =

0	1	2	3	4	5	6	7
0	1	4	∞	2	8	6	∞

vis: 0, 1, 4

⑤ Remove Min → 2; decrease dist

0	1	2	3	4	5	6	7
0	1	4	9	2	8	6	∞

vis: 0, 1, 4, 2

⑥ Remove min → 6; decrease dist

0	1	2	3	4	5	6	7
0	1	4	9	2	8	6	16

!!!

vts
0
1
4
2
6

⑦ Repeat twice more, final dist remains unchanged from #6 above:

dist =

0	1	2	3	4	5	6	7
0	1	4	9	2	8	6	16

*First Shortest Path found by Dijkstra:

0 → 6 → 7 (cost = 16)

This path is found first by Dijkstra due to its mechanism for picking the minimum distance unvisited node and updating all unvisited's distance with a new minimum as needed. In step #6 above we find 0→6→7 first because, of the unvisited nodes, 6 has a minimum distance (3=9, 5=8). So, 7 is updated with a distance of 16 first from node 6. The other paths (obviously all 16 distance from 0→7) will not affect this min distance. Thus, the first shortest path found here is 0→6→7.

Bellman Ford

*Main Induction Hypothesis:

With each iteration of the loop, we will be calculating the shortest path to all other vertices from source s using at most k edges. Either we will have the shortest distance using $\leq k$ edges at dist[v] or ∞ if unreachable in $\leq k$ edges

① dist = | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | (0 1 2 3 4 5 6 7) Initialize

② dist = | 0 | 1 | ∞ | ∞ | 2 | ∞ | 6 | ∞ | U = ∅, [U] ≠ ∞

③ dist = | 0 | 1 | 4 | ∞ | 2 | ∞ | 6 | ∞ | U = 1, [U] ≠ ∞

④ dist = | 0 | 1 | 4 | 9 | 2 | ∞ | 6 | ∞ | U = 2, [U] ≠ ∞

⑤ dist = | 0 | 1 | 4 | 9 | 2 | ∞ | 6 | 16 | U = 3, [U] ≠ ∞ ← !

...

⑦ dist = | 0 | 1 | 4 | 9 | 2 | 8 | 6 | 16 | → Repeat |V| - 1 times.

→ We repeat the relaxation of edges loop once more to detect a negative cycle ($k = |V|$ th iteration), but in this case, there are no negative edge weights, so no negative cycle.

*First Shortest Path found by Bellman Ford:

0 → 1 → 2 → 3 → 7 (cost = 16)

This path is found first by Bellman Ford because, despite all paths having the same cost, this path reaches 7 first because of the order in which dist is updated here. The edges are relaxed through an iteration through each vertex, 0 -> 7. 3 -> 7 is relaxed before 5 -> 7 or 6, 7, hence found first.

Floyd-Warshall.

*Main Induction Hypothesis:

In each iteration of the algorithm, we will compute all pair shortest k-paths. A k path is defined as all intermediate vertices in the path (neither source nor sink) are labelled with numbers $\leq k$. Each iteration computes the shortest path between any 2 vertices u, v using only intermediate nodes numbered distinctly 0 -> k. Let k = -1:

dist =

	0	1	2	3	4	5	6	7
0	0	1	∞	∞	2	∞	6	∞
1	1	0	3	∞	∞	∞	∞	∞
2	∞	3	0	5	∞	∞	∞	∞
3	∞	∞	5	0	∞	∞	∞	7
4	2	∞	∞	∞	0	6	∞	∞
5	∞	∞	∞	∞	6	0	∞	8
6	6	∞	∞	∞	∞	∞	0	10
7	∞	∞	∞	7	∞	8	10	0

(In initialization, every path with intermediary nodes $\leq k$, where k starts at -1; essentially, an adjacency matrix).

Now, iterate through until k = 7. and test for all combinations of $i, j \in [0, 7]$ and update with:

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

k = 0 :

	0	1	2	3	4	5	6	7
0	0	1			2		6	
1	1	0	3		3		7	
2		3	0	5				7
3			5	0				
4	2	3			0	6	8	
5					6	0		8
6	6	7			8		0	10
7				7		8	10	0

min of
nodeA → nodeB
OR
nodeA → ∅ +
∅ → nodeB

K = 1

	0	1	2	3	4	5	6	7
0	0	1	4		2		6	
1	1	0	3		3		7	
2	4	3	0	5	6		10	
3			5	0				7
4	2	3	6		0	6	8	
5					6	0		8
6	6	7	10		8		0	10
7				7		8	10	0

min of
nodeA → nodeB
OR
nodeA → 1 +
1 → nodeB

K = 2

	0	1	2	3	4	5	6	7
0	0	1	4	9	2		6	
1	1	0	3	8	3		7	
2	4	3	0	5	6		10	
3	9	8	5	0	11		15	7
4	2	3	6	11	0	6	8	
5					6	0		8
6	6	7	10	15	8		0	10
7				7		8	10	0

min of
nodeA → nodeB
OR
nodeA → 2 +
2 → nodeB

K=3

	0	1	2	3	4	5	6	7
0	0	1	4	9	2		6	16
1	1	0	3	8	3		7	15
2	4	3	0	5	6		10	12
3	9	8	5	0	11		15	7
4	2	3	6	11	0	6	8	18
5					6	0		8
6	6	7	10	15	8		0	10
7	16	15	12	7	18	8	10	0

... (0 -> 7 remains 16).

K=7.

	0	1	2	3	4	5	6	7
0	0	1	4	9	2	8	6	16
1	1	0	3	8	3	9	7	15
2	4	3	0	5	6	12	10	12
3	9	8	5	0	11	15	15	7
4	2	3	6	11	0	6	8	14
5	8	9	12	15	6	0	14	8
6	6	7	10	15	8	14	0	10
7	16	15	12	7	14	8	10	0

Final all shortest paths output.

min of
node A -> node B
OR
node A -> 3 + 3 -> node B

<= Shortest Path found By Floyd Warshall:

0 -> 1 -> 2 -> 3 -> 7.

The reason this is found first as the shortest path from 0 -> 7 is due to the numbering of the vertices.

On K = 0; we ask, what is the shortest path from every vertex to every other vertex using 0 as an included "stepping stone".

We repeat the question with, K = 1, K = 2, K = 3 -> and this is then the first time the 0 -> 7 distance is updated to 16 (the obvious shortest path cost from 0 -> 7)

We can see in action, that the algorithm first considers

K=0 → 0→7 (∞) or 0→0 + 0→7 (0 + ∞): 0→7 is infinity still, no update.

K=1 → 0→7 (∞) or 0→1 + 1→7 (1 + ∞): 1→7 is infinity, no update

K=2 → 0→7 (∞) or 0→2 + 2→7 (4 + ∞): 2→7 is infinity, no update.

K=3 → 0→7 (∞) or 0→3 + 3→7 (9 + 7): 9+7 < ∞, update. dist[0][7] = 16.

The intermediate calculations of 0→1, 0→2, 0→3 were shortest path calculations at K=0, 1, 2. The only path 0→7 that fits the constraints of all intermediary nodes at ≤ K=3 is the path

0→1→2→3→7.

(6.) Given: undirected simple graph $G(V, E)$

— Is this graph a tree?

General approach: Check if the graph G is ① connected and ② the number of edges == number of vertices − 1.

Let G be represented by an adjacency list.

```
* isTree(G(V,E))
    return numEdges(G) == numVertices(G) - 1
           && isConnected(G)
```

NOTE: I am going to not make the assumption right now that the adjacency list representation is keeping track of the numbers of edges & vertices internally. Hence, the following methods.

```
* numEdges(G(V,E))
    e = 0
    for(each v ∈ V)
        e += v.edges.length // length is the size of this adjacency list.
    return e
```

```
* numVertices(G(V,E))
    return V.length // return size of V set / list / etc.
```

☆ isConnected (G=(V,E))

```
visited = new array of size numVertices(G),
          with all values initialized to false.

// If anything not visited by single dfs traversal, graph is not
   connected.
dfs(V[0], visited).

for(each v ∈ V)
    if(!visited[v.index])
        return false

return true.
```

☆ dfs(v, visited)

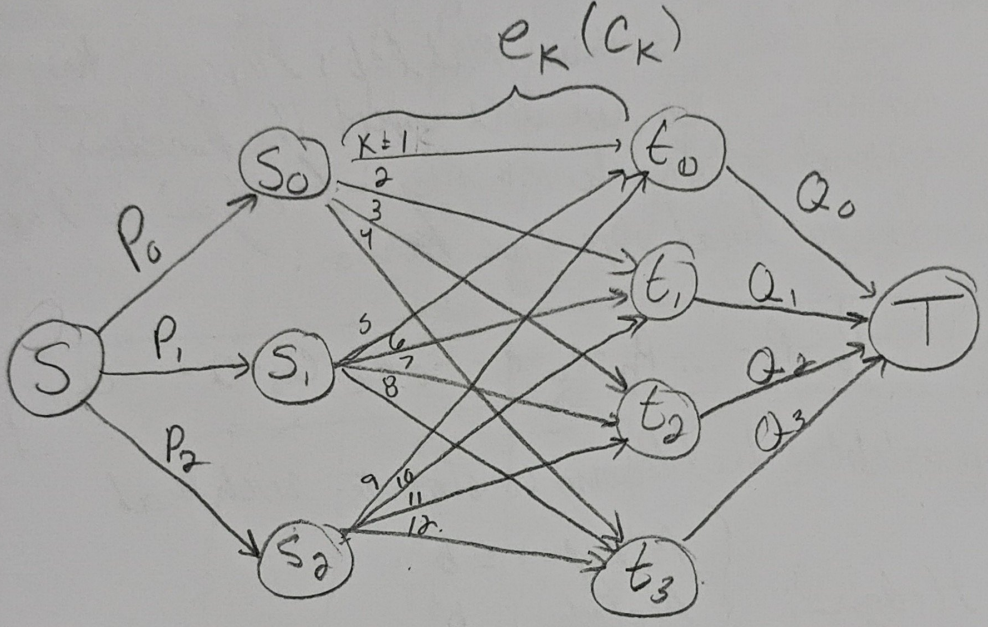
```
visited[v.index] = true. // visit.

for(each edge e ∈ v.edges)
    if(!visited[e.to.index])
        dfs(e.to, visited)
```

NOTE: following is some implementation specifications as mentioned in pseudocode above.

G (graph)	V (vertex)	E (edge)
V = a set/list/etc of vertices.	edges = adjacency list of edges from vertex. index = convinience ID that is a distinct integer [0, ∞)	to = the destination vertex from = the from/start vertex.

(7) We can frame this problem as a maximum flow problem. We will simplify the problem to a min-flow-problem, but w/ our objective function inverted.

To set up what this could look like visually (as a rudimentary start); lets look at a network with 3 families and 4 cars.



→ Let S & T be dummy nodes to represent a single source and sink respectively.

→ Let S_n be a node representing a family going on the road trip, where $n \in [0, \text{number of families})$.

→ Let t_m be a node representing a car to be used on the road trip, where $m \in [0, \text{number of cars})$

→ Let e_K be the directed edges from each family (S_n) to each car (t_m). These will have an associated capacity, c_K, that (in our simplified version of a solution) will be set to 1. Note that $K \in [0, (\text{number of families}) * (\text{number of cars}))$.

→ Let P_n & Q_m represent edges from $S \to S_n$ and $T \to t_m$ respectively. They also have a capacity associated: P_n capacity is number of family

members associated with its destination node (t_n) (Q_n) capacity will be the sum of edge e_k capacities at its start node (t_n).

We can even begin to dream up a linear programming set up to consider solutions. To consider an objective function, we can simply tally the edges (capacity details will be listed in our constraints):

$$f(\text{edges}) = P_0 + P_1 + \ldots + P_n + e_0 + e_1 + \ldots + e_k + Q_0 + Q_1 + \ldots + Q_n.$$

However, using most tools (such as Matlab's linprog), they are taking a function to minimize. So, we will invert the function ($-f(\text{edges})$) such that the optimal solution to minimize $-f(\text{edges})$ will maximize $f(\text{edges})$. Therefore, our final objective function:

$$f(\text{edges}) = -P_0 - P_1 - \ldots - P_n - e_0 - e_1 - \ldots - e_k - Q_0 - Q_1 - \ldots - Q_n.$$

Further, we can establish the following constraints, such that:

$$\max f(\text{edges}) \begin{cases} A(\text{edges}) \leq b \\ Aeq(\text{edges}) = b \\ lb \leq \text{edges} \leq ub \end{cases}$$

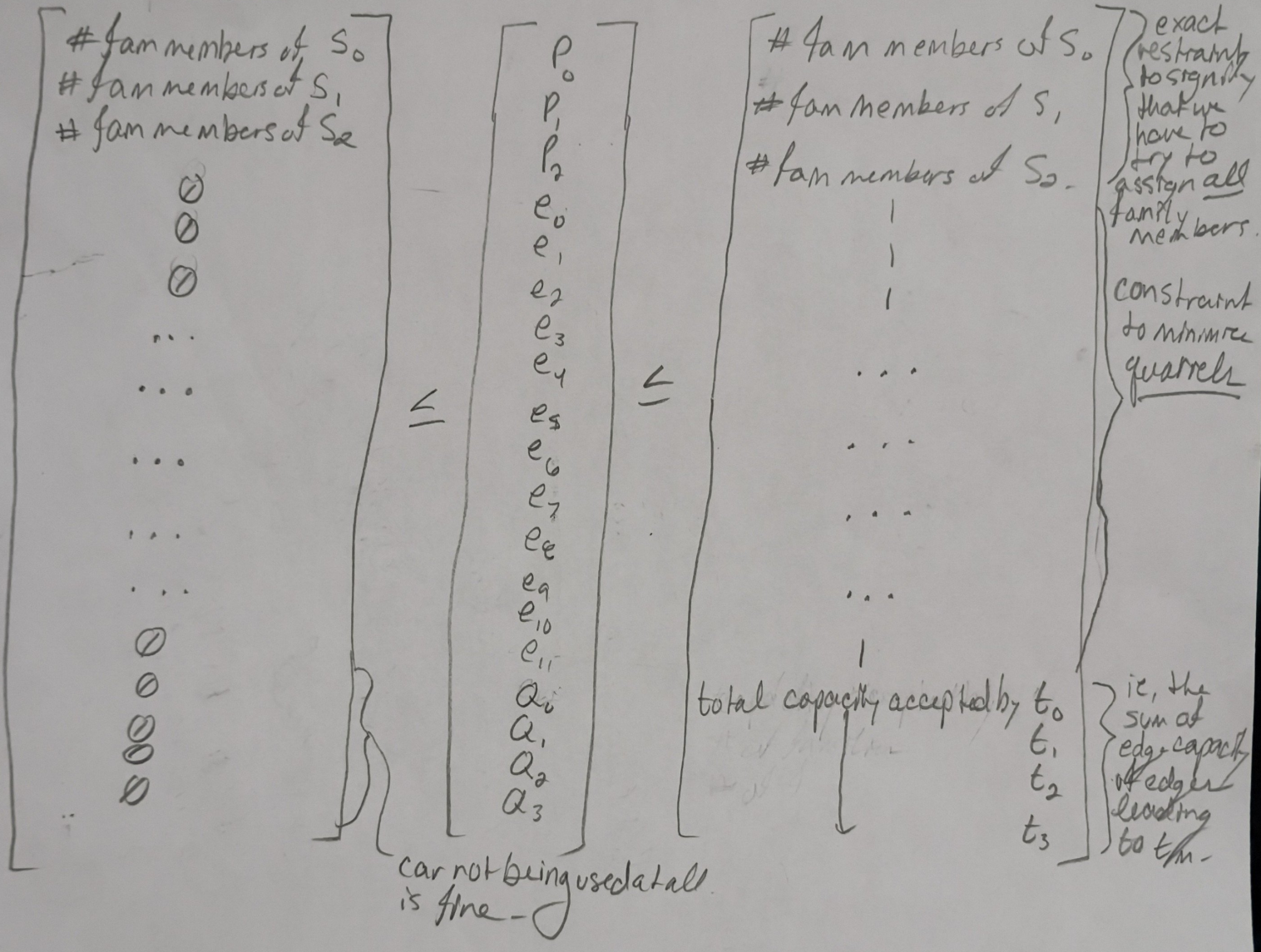
We can set the following constraints (using our 3 family, 4 car example for simplicity) using some linear programming tools for simplicity:

$$A: \begin{cases} e_0 + e_1 + e_2 + e_3 \leq P_0 \\ e_4 + e_5 + e_6 + e_7 \leq P_1 \\ e_8 + e_9 + e_{10} + e_{11} \leq P_2 \end{cases} \quad Aeq: \begin{cases} e_0 + e_4 + e_8 = Q_0 \\ e_1 + e_5 + e_9 = Q_1 \\ e_2 + e_6 + e_{10} = Q_2 \\ e_3 + e_7 + e_{11} = Q_3 \end{cases}$$

General Inflow / Outflow constraints. We will not include our dummy S, T nodes here since they are only dummy nodes.

→ Note that I am allowing $P_v \geq \sum e_k$ at its destination node. This is an important note on the set up: it essentially represents the idea that there may be some family members unassigned to a car by the algorithm. (I send out 4 members for assignment, only 3 cars, only 3 members are assigned a car).

And now bounds:



The above set up will maximize the number of members assigned to cars. As noted above, a limitation in this set up is that some members could potentially not be assigned to a car, in which case we would have to put some members in the same car. We also don't consider realities of # of car seats, but this is a general starting set up to general network flow framing with some additional linear programming style constraints to start considering a tangible solution to the problem.