IE501-2-Grp

Atharva Bendale

September 2023

Contents

1 Question 1

Project Problem Statement : Lights Out

1.1 Motivation for the project problem

The motivation behind addressing the problem described in this project proposal stems from the need to tackle complex scenarios that involve toggling the states of interconnected elements, such as a grid of lights. The problem can have wide-ranging practical applications like:

- Puzzle Solving and Games
- Efficient Resource Utilization
- Localized Actions with Global Impact
- Customer Differentiation and Market Segmentation:
- Effect of actions of social media influencers on social media.

In summary, the motivation behind addressing this project problem lies in its relevance to a diverse range of real-world applications. By finding efficient solutions to this problem, we can gain valuable insights into resource allocation, network connectivity, localized actions, and optimization strategies that can benefit both theoretical research and practical decision-making processes.

1.2 Detailed description of the project

The problem statement of this project is as follows:

> Given a n x n grid where each block can be either on (1) or off (0), we can toggle the state of each block but doing so will change the state of all the blocks that share a border with that particular block. This counts as one operation. Find whether we can turn off all the blocks in the grid using some series of operations and if it is possible, return the minimum number of operations required for this.

This problem for a n x n grid can be extended to a general graph where each block is a node and blocks sharing a border would have an edge between them. This problem can also be extended to include 'k' states instead of just 2 (on and off). These sorts of problems have a lot of real life applications that we have discussed in the further sections.

1.3 Tentative solution methodology ideated

To find the solution to this problem, we would first need to formulate a linear program for this and then solve that by coding that up in python (or by using some solver)

- **Formulation:** As switching a block two times is equivalent to not doing anything at all, the maximum number of times an operation should be done on one block is 1. From this we can conclude that our decision variables would be binary and for each of the blocks, would store 0 if it doesn't need to be toggled, 1 if it does need to be toggled. Thus, the total number of operations would just be the summation of all these decision variables, and our objective function would be to minimize this summation. We would need to come up with proper constraints to set up the border-sharing condition.
- **Solving:** For solving this problem statement, we can write up a python code and then enforce the constraints that we formulated in our linear program to minimize the required summation or we can use any of the available LP solvers.

1.4 Tentative deliverables

There are a lot of real life problems which can be simplified to this problem like targeted advertising, customer differentiation i.e. any problem which consist of analysing the effect of a localised action on an isolated group of objects. Our solution can then be used to find the minimum localities an entity must focus on to ensure that it gets convered by the entire population.

1.5 Distribution of work amongst team members

There are two major parts of this project (at least for the n x n grid) :

- Formulating a Linear Program to which simulates the property of the "Lights Out" game consisting of linear constraints (might be non-linear for a general geometry grid).
- Using the derived constraints to solve the problem using any one of the available LP solvers.

As both of these parts cannot be done simultaneously we intend to brainstorm on both parts together, we might distribute the work (more like each member can derive linear constraint which will simulate one of the properties of the problem) after we have a detailed idea of how our problem can be best approached.

2 Question 2

We are asked to :

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \end{aligned}$$

We can calculate the certificate of infeasibility for a LP in standard form using Farkas' Lemma, so it would be convenient to convert the above LP into a standard form. This linear program is in the canonical form, we can convert it into the standard form as follows. Consider the linear program formed by:

$$\begin{aligned} \text{minimize} \quad & c^T x' \\ \text{subject to} \quad & A'x' = b \qquad x' \geq 0 \end{aligned}$$

Where ,

$$A' = \begin{bmatrix} A & -A & I \end{bmatrix} \qquad x' = \begin{bmatrix} x_1 \\ x_2 \\ s \end{bmatrix}$$

We can show that the above two linear program are equivalent, since for every x, we can always find two positive numbers such that x_1, x_2, such that both are non-negative and $x = x_1 - x_2$, and since if there is a solution x such that $Ax \leq b$, then there exists a non-negative number s such that $Ax + s = b$. So finding the solution of one LP is equivalent to finding the other.

Therefore the problem of finding the certificate of infeasibility is equivalent to finding the same for other. We can easily find the certificate for the second LP by applying Farkas' lemma to the second LP to get the conclude that , if there exists a y, such that ,:

$$y^T A' \geq 0 \text{ and } y^T b < 0$$

Then the LP is infeasible. Also $y^T A' \geq 0$ implies $y^T A \geq 0, -y^T A \geq 0, y^T \geq 0$ or $y^T A = 0$ and $y \geq 0$. Hence the final certificate of infeasible is:

$$\exists y \text{ such that } \quad y^T A = 0, y \geq 0 \text{ and } y^T b < 0$$

3 Question 3

We first constructed a graph $G(E,\mathcal{N})$ from the map of the IIT Bombay, where each node represents a point on the map from where we can go in at least three directions. And the roads connecting two nodes are represented as edges of the graph.

We have solved the problem as follows. We have applied two sets of constraints to the set of edges selected. the first constraint ensures that for every node that gets selected due to a selected edge, the number of edges incident on that node is even. This ensures that for every node if we enter the node through an edge, we can exit the node from a different edge. The second set of constraints ensures that every node that gets selected due to the selected edges is connected; in the selected edges, we can reach every node from every other such selected node.

We claim that under these constraints, we can always find a closed connected path so no edge is repeated. First, if we begin from a point, we always have to come back since there are even numbers of edges incident on the. Since our constraints don't allow disjoint circuits, the set of edges will form a connected circuit, where it is always possible to enter and leave a node from distinct edges and it is ensured that we return to the starting point

3.1 Parameters

Let us define the sets

$$E_i = \{(i,j) | (i,j) \in E \text{ and the edge } (i,j) \text{ is incident on the } i^{th} \text{ vertex of} \mathcal{N}\}$$

The distance between of the roads is stored in another set of variables w_{ij}, where

$$w_{ij} = \text{The distance between of the road between vertices i and j}$$

We also will define P to be the powerset of the set of all vertices V.

$$P = \text{Powerset}(\{x : x \ \in \ \mathbb{N}, \ 1 \leq x \leq |\mathcal{N}|\})$$

3.2 Decision Variables

I have considered the following decision variables. Note: Here, we have used *integer linear program*, that is, the decision variables can take only integer values.

$$\forall \ i \in [1, |\mathcal{N}|], \ \forall \ j \in [1, |\mathcal{N}|] \ : \ x_{ij} = \begin{cases} 1 & \text{If the edge (i,j) is selected in the path} \\ 0 & \text{otherwise (Note: } x_{ii} \text{ is also 0)} \end{cases}$$

$$y_i \ : \ y_i \in \mathbb{N} \qquad \forall \ i \in \{1, 2, 3 \ldots, |\mathcal{N}|\} \tag{1}$$

Some non-decision variables are :

$$\begin{aligned} T_i = \{0,1\} \qquad & \forall \ i \in \{1, 2, 3 \ldots, |\mathcal{N}|\} \\ U_\alpha = \{0,1\} \qquad & \text{where } \alpha \in P \\ V_\alpha = \{0,1\} \qquad & \text{where } \alpha \in P \end{aligned}$$

3.3 Objective Function

$$\max \sum_{i \in \mathcal{N}} \sum_{j:(i,j) \in E_i} (x_{ij} w_{ij})$$

3.4 Constraints

- $y_k \geq 0 \qquad \forall k$
- We need to ensure that for any node which is selected, in our path the number of edges incident on it will be even. That is, we want that if we enter a node, we can always exit from a different edge than we entered. This can be done as follows:

$$\sum_{(i,j)\in E_i} x_{ij} = 2 \cdot y_i \qquad \forall i$$

Now we will define a series of binary variables (none of which are decision variables) to ensure that our LP never chooses a set of variables which have two or more disjoint vertices.

- For the binary variable T_i, we will apply the following constraint which will ensure $T_i = 1$ if i^{th} node is visited at least once in our path else becomes 0. To ensure this we need to relate it to x_{ij} as below, $\quad \forall\ i\ \in\ [\ 1,\ |\mathcal{N}\ |]$,

$$T_i\ \leq \sum_{j:\ (i,j)\in E_i} (x_{ij})$$

$$T_i\ \geq\ x_{ij} \qquad \forall\ j:\ (i,j) \in E_i$$

- For U_α, we will apply constraints which will ensure $U_\alpha = 1$ if atleast one node in $\alpha\ \in\ P$ is visited atleast once else it is 0. $\quad \forall\ \alpha \in P$,

$$U_\alpha\ \leq\ \sum_{i\in\alpha} (T_i)$$

$$U_\alpha\ \geq\ T_i \quad \forall\ i\ \in\ \alpha$$

- For the binary variable V_α, we will define constraints which will ensure that $V_\alpha = 1$ if both $U_\alpha = 1$ and $U_{\alpha^c} = 1$ else $V_\alpha = 0$. $\qquad \forall \alpha \in P$,

$$V_\alpha \leq U_\alpha$$

$$V_\alpha \leq U_{\alpha^c}$$

$$V_\alpha + 1 \geq U_\alpha + U_{\alpha^c}$$

Now using these defined variables we can finally employ a constraint which eliminates the possibility of any two disjoint loops occuring in our predicted path.

$$\sum_{i\in\alpha} \sum_{j\in\alpha^c} (x_{ij})\ \geq\ V_\alpha \qquad \forall\ \alpha\ \in\ P$$

Because of these constraints, we can't have two disjoint vertices because consider the subset α which contains exactly one of these cycles, for this α, V_α will be 1, but the summation on the left side will be zero, hence such a selection won't be allowed.

3.5 Our Approach and Solution

To solve this problem, we wrote a python code which takes in the input graph and outputs the maximum possible closed path in the graph. The code uses DFS to recursively generate all paths which satisfy the given constraints and then find the largest path(i.e. max of the our decision variable). The DFS can be run choosing any of the nodes as the starting point and we get the longest loop containing that node.
though the above was feasible(and we did do it later just for checking) but it would have taken a long time to give ans so we **greedily** choose the node 1 as the starting as the total length of roads out of that is approx 2 km which is the highest. We then ran our code and the ans we got differed from the total length of roads by nearly 2 kms only so we were able to say with certainty that a path that did not contain 1 would definitely smaller

The results we got from this node were:

```
The max length of a closed path is:  8713.0 metres
The path for the max length is:  [1, 23, 21, 29, 28, 27, 26, 25, 24, 29, 30, 31,
16, 17, 11, 14, 12, 7, 4, 5, 2, 6, 12, 15, 35, 34, 33, 32, 22, 30, 18, 20, 19, 10,
8, 9, 3, 2, 1]
The total length of all the roads is:  10818.0 metres
```

The path highlighted in blue indicates the optimal path.

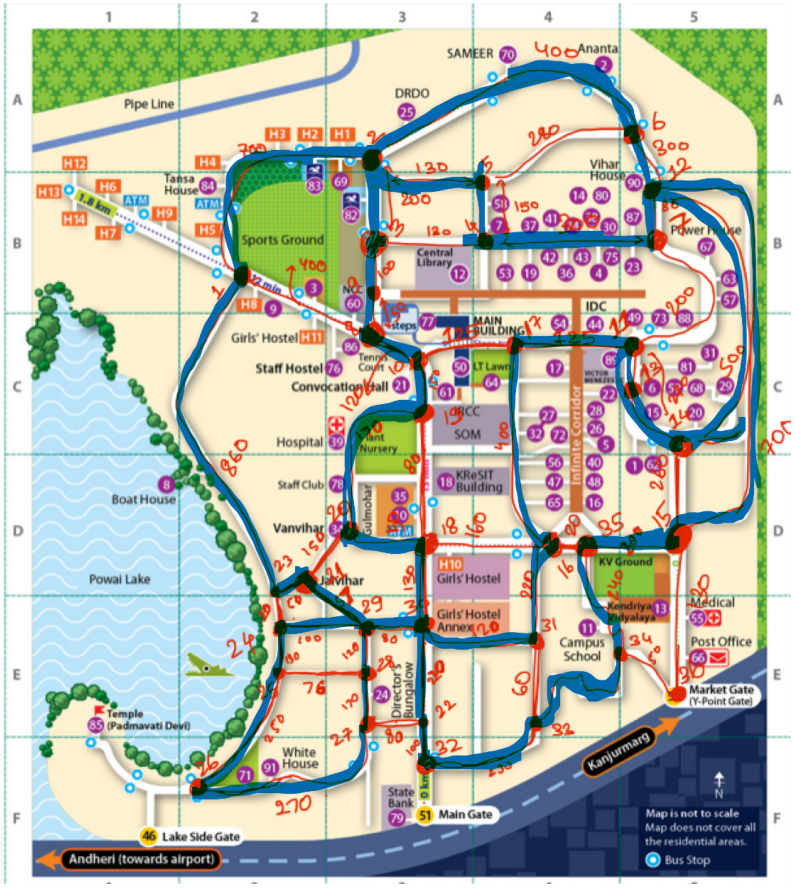

Figure 1: Tracing this path on the IITB map provided

```
import numpy as np
def DFS(start,cur,gra,max,n):
    niv=[]                # List to store our path
    if(start==cur and max):     # Telling our previous call that the current path can loop back to the start
        max=max+1               # Small increase for conditional statement
        niv.append(cur)         # Updating the path list for the current iteration
        return max,niv

    m=0                         # To store the max after this point

    for i in range(1,n):
        if(gra[cur][i]):        # If there is a road, then proceed down that road
            st=[]               # Path when going down that road
            a=gra.copy()
            l=gra[cur][i]       # Storing the length of the current road
            le=gra[cur][i]      # Storing the length of the current road for comparison later on
            a[cur][i]=0
            a[i][cur]=0         # Removing the road after going down it

            # Iterating down to find the biggest path that starts here and loops back to the start
            l,st=DFS(start,i,a,l,n)
            if (l!=le):         # Finding the max out of all the feasible paths
                if(l>m):
                    m=l
                    niv=st

    max=max+m
    niv.append(cur)
    return max,niv

if __name__ == '__main__':
    n=37        # Number of intersections + 1
    a=np.zeros((n,n))       # Initialising an array containing road between any two vertices
    start=1                 # Picking an initial node
    max_len=0
    graph = [[] for i in range(n)]   # Initialising our graph

    # Adding the nodes and weights
    graph[1].append([2, 700])
    graph[1].append([8, 400])
    graph[8].append([9, 50])
    graph[9].append([3, 100])
    graph[3].append([2, 200])
    graph[2].append([6, 400])
    graph[5].append([6, 280])
    graph[2].append([5, 130])
    graph[4].append([5, 150])
    graph[3].append([4, 130])
    graph[4].append([7, 300])
    graph[6].append([12, 300])
    graph[12].append([14, 500])
    graph[12].append([7, 80])
    graph[7].append([11, 200])
    graph[11].append([17, 125])
    graph[8].append([10, 120])
    graph[11].append([14, 270])
    graph[11].append([13, 100])
    graph[13].append([14, 170])
    graph[15].append([36, 130])
    graph[34].append([36, 50])
    graph[34].append([35, 240])
    graph[16].append([35, 20])
```

```
    graph[33].append([34, 270])
    graph[31].append([33, 60])
    graph[16].append([31, 220])
    graph[16].append([18, 160])
    graph[18].append([30, 130])
    graph[18].append([20, 50])
    graph[19].append([20, 170])
    graph[18].append([19, 80])
    graph[20].append([21, 150])
    graph[21].append([23, 50])
    graph[21].append([29, 57])
    graph[29].append([24, 100])
    graph[23].append([24, 20])
    graph[24].append([25, 130])
    graph[25].append([28, 76])
    graph[28].append([29, 120])
    graph[29].append([30, 80])
    graph[27].append([28, 170])
    graph[26].append([27, 270])
    graph[25].append([26, 250])
    graph[22].append([27, 80])
    graph[22].append([32, 100])
    graph[22].append([30, 220])
    graph[30].append([31, 120])
    graph[32].append([33, 230])
    graph[16].append([17, 400])
    graph[10].append([19, 150])
    graph[15].append([35, 200])
    graph[34].append([36, 50])
    graph[12].append([15, 700])
    graph[1].append([23,860])

    # Converting our graph into a 2D numpy array
    for i in range(1,n):
        for j in range(len(graph[i])):
            a[i][graph[i][j][0]]=graph[i][j][1]
            a[graph[i][j][0]][i]=graph[i][j][1]

    # Finding the total length of all the roads
    len=0
    for i in range(1,n):
        for j in range(1,n):
            len+=a[i][j]

    max_len,b=DFS(start,start,a,max_len,n)
    print("The max length of a closed path is: ",max_len)
    print("The path for the max length is: ", b)
    print("The total length of all the roads is: ", len/2)
```