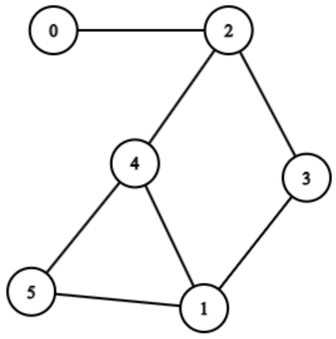
Problem Set 5

Name: Gabriel Chiong

Collaborators: None

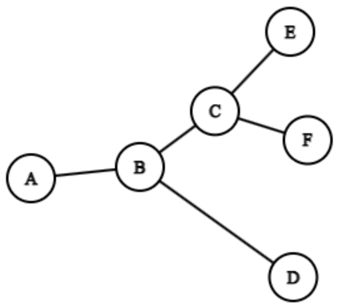
Problem 5-1.

(a) Answer:

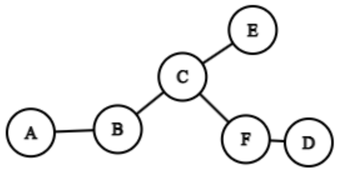


(b)
```
Adj = {
      'A': ['B'],
      'B': ['C', 'D'],
      'C': ['E', 'F'],
      'D': ['E', 'F'],
      'E': [],
      'F': ['D', 'E']
}
```

(c) BFS: A, B, C, D, E, F



DFS: A, B, C, E, F, D



(d) There is only one cycle in the graph, therefore removing either (D, F) or (F, D) would result in a DAG.

If we remove edge (F, D), there are two valid topological orderings, based on running DFS: `[A, B, D, C, F, E]` and `[A, B, C, D, F, E]`.

If we remove edge (D, F), there is a unique topological ordering from running DFS: `[A, B, C, F, D, E]`.

Problem 5-2. If we construct a graph G with plants and buildings as vertices and wires as edges, this problem is equivalent to finding the largest connected component (by vertex count). We know that each plant belongs to at most one connected component since each building is powered by no more than one plant. Therefore, the plant belonging to the largest connected component should receive the backup generator.

Graph G has $n^2 + n$ vertices (number of buildings and number of plants). Therefore $|E|$ is upper-bounded by $O(n^2 + n) = O(n^4)$ as required by the problem, since there is at most one wire connecting each pair of vertices.

The algorithm proceeds by running Full-BFS or Full-DFS to count the size of each connected component and return the largest connected component's plant. Constructing G takes $O(n^4)$, running Full-BFS or Full-DFS takes $O(n^4)$, and iterating over the connected components to find the largest count takes $O(n)$ time. Therefore, the entire algorithm runs in $O(n^4)$ time as required.

Problem 5-3. Construct a (potentially disconnected) undirected graph G, with each vertex representing a friend in the n-length short-circuiting pairs list, and an edge between any pair of vertices of short-circuiting friend vertices.

The problem of determining whether two parties are sufficient for inviting all their friends is then equivalent to finding a 2-coloring of graph G. Each color represents a party, and if adjacent vertices can be divided into two sets, then two parties is possible. If there is an odd-cycle in the graph (two adjacent vertices have the same color; two short-circuiting friends attending the same party), then more than two parties will be needed.

The algorithm is to run Full-BFS on G to obtain the shortest path distances from an arbitrary starting vertex from each connected component. Along the way, assign each vertex one of two colors, with all even-length shortest distances being of the same color, and all odd-length shortest distances being of another color. Next, iterate over all edges of the graph, and for each edge check if both adjacent vertices are of different color. If there is at least one pair of adjacent vertices with the same color, then return that two parties is not sufficient. Otherwise, at the end of iterating through all edges in G, return that two parties is possible.

G has n edges, and at most $2n$ vertices. Therefore it takes at most $O(n)$ time to construct. Using Full-BFS to color the vertices takes $O(|V| + |E|) = O(2n + n) = O(n)$ time, and iterating over edges takes $O(n)$ time (since there is only a constant amount of work to check each edge). Therefore, the algorithm runs in $O(n)$ time in the worst-case.

Problem 5-4. Let M be the $n \times n$ labeled map, where $M[r][c]$ is the label of the grid square in row r and column c. Construct a graph G with $(n+1)^2$ vertices, where each vertex is labeled (r, c) for all $r, c \in \{0, \ldots, n\}$. We place an undirected edge between two vertices under two scenarios.

Firstly, between vertices $(r, c-1)$ to (r, c); left to right of the map, on the condition that r is not a boundary edge (i.e. $r \in \{0, n\}$ is true) and $M[r][c-1] \neq M[r][c]$ (edge does not go between two squares owned by the same farmer).

Secondly, between vertices $(r-1, c)$ to (r, c); top to bottom of the map, on the condition that c is not a boundary edge (i.e. $c \in \{0, n\}$ is true) and $M[r-1][c] \neq M[r][c]$ (edge does not go between two squares owned by the same farmer).

This graph has the property that a path between two vertices corresponds to a route on the map that does not trample crops. For each square $M[r][c]$ that is part of the Euphris river, mark the vertices $(r, c), (r+1, c), (r, c+1), (r+1, c+1)$ as "Euphris squares". Do likewise with squares belonging to the Tigrates river.

Using BFS on each of the $\Omega(n^2)$ Euphris-labeled vertex would incur a time complexity of $\Omega(n^2)$ for each run of BFS. This exceeds the specified time complexity of the required algorithm. Therefore, add a supernode s to G with an undirected edge to each Euphris-labeled vertex to construct graph G'. Now any possible trade route will correspond to a path from s to a Tigres-labeled vertex in G' (discounting the supernode), and return the shortest path to any Tigres-labeled vertex by traversing parent pointers back to the source.

Graph G' has $(n+1)^2 + 1$ vertices (including the supernode), and $O(n^2) + O(n^2)$ edges, so can be constructed in $O(n^2)$ time. Running BFS once from s also takes $O(n^2)$ time, so this algorithm runs in $O(n^2)$ time overall.

Problem 5-5. Since the doors that Liza can enter depends on which key cards she currently has, we use graph duplication to enumerate the different scenarios along her journey. Since there are four types of key cards, there are at most $2^4 = O(1)$ different possible states from key cards. At any location, there is also either a pizza or not, which gives $2^2 = O(1)$ different possibilities. We therefore construct a graph G with 2^5 vertices, representing the different combinations of key cards and pizza that Liza has along the journey.

Let $T = \{SeeSail, TOPS, S3C, DPW\}$, and let $C(l, k_t)$ equal 1 if $l = l_t$ and k_t otherwise, and let $P(l, p)$ equal 1 if location l contains a pizza and p otherwise. Each of the 2^5 vertices will store a 6-tuple $(l, p, k_{SeeSail}, k_{TOPS}, k_{S3C}, k_{DPW})$, where $p \in \{0, 1\}$ and $k_t \in \{0, 1\}$. This represents the state that Liza is currently in when she arrives location l. If Liza enters any location l having at least p pizzas and k_t key cards of type t, she can leave with at least $P(l, p)$ pizzas and $C(l, k_t)$ key cards of type t. In other words, she can only gain items along the way.

For each edge in the graph (doors) represented by (l_1, l_2), if the door does not require card access or Liza currently has the requisite card i.e. $C(l_1, k_t) = 1$, then add a directed edge from:

•vertex $v(l_1, p, k_{SeeSail}, k_{TOPS}, k_{S3C}, k_{DPW})$ to

•vertex $v(l_2, P(l_1, p), C(l_1, k_{SeeSail}), C(l_1, k_{TOPS}), C(l_1, k_{S3C}), C(l_1, k_{DPW}))$.

For all $p \in \{0, 1\}$ and $k_t \in \{0, 1\}$. If a door goes two ways, implement an edge in the opposite direction in the same way.

A path from vertex $s = v(e, 0, 0, 0, 0, 0)$ to any vertex $v(e, 1, k_{SeeSail}, k_{TOPS}, k_{S3C}, k_{DPW})$ represents a path that enters and leaves the Statum Center at e while also obtaining a pizza. To minimize the amount of doors opened, running BFS from s results the shortest path to any other vertex in G (the minimum number of doors opened), so we can return the shortest path using parent pointers.

Graph G has $|L| \cdot 2^5 = O(|L|)$ vertices and at most $|D| \cdot 2^5 = O(|D|)$ edges, so can be constructed in $O(|L| + |D|)$ time, and running BFS once from s takes time linear to the size of the graph, so this algorithm runs in $O(|L| + |D|)$ time.

Problem 5-6.

(a) Only sliders can move throughout the entire game, therefore this problem is equivalent to placing s non-distinct sliders (it does not matter which slider reaches the target, only that one does) onto a board of $n^2 - b$ empty spaces. There are $n^2 - b$ ways to place the first slider, $n^2 - b - 1$ to place the second, and so on until the $n^2 - b - s + 1$ places for the n-th slider. However, the product of these overcounts by a factor of $s!$ since the sliders are not distinct, therefore the total number of board configurations is $C(n, b, s) = \frac{1}{s!}\prod_{i=0}^{s-1}(n^2 - b - i)$.

(b) There are only four directions to move for any board configuration, therefore there can only be four successors to a board configuration B.

In the extreme case where B has no obstacles and $s \le n$, with each slider placed in the bottom cell of distinct columns, the number of predecessors would be lower bounded by $\Omega(n^2)$. This board configuration is reachable from any "down" predecessor with s sliders in the same columns. Each slider in the predecessor can be in any of the n column squares. Therefore, there must be at least $\Omega(n^s)$ predecessors of B.

(c) Run BFS starting from B, constructing vertices (a.k.a. board configurations) for any reachable configuration along the way. Use the `move(B,d)` function to compute each successor in $O(n^2)$ time. Return the (shortest) path at the first occurrence of `B[yt][xt]='o'`, where $B*$ is a target state. Let the number of moves to reach B^* be k. Using $r = 4$ as the number of successors for a board configuration, the total number of configurations generated along the way to a successful solution is $\sum_{i=0}^{k} r^i < r^{k+1} = O(r^k)$.

In the event that B is not solvable, then every possible configuration may be explored along the way, which is $C(n, b, s)$ configurations. Since it takes $O(n^2)$ time to generate each configuration, the overall worst-case time complexity of this search is $O(n^2 \min\{r^k, C(n, b, s)\})$.

(d) Submit your implementation to `alg.mit.edu`.