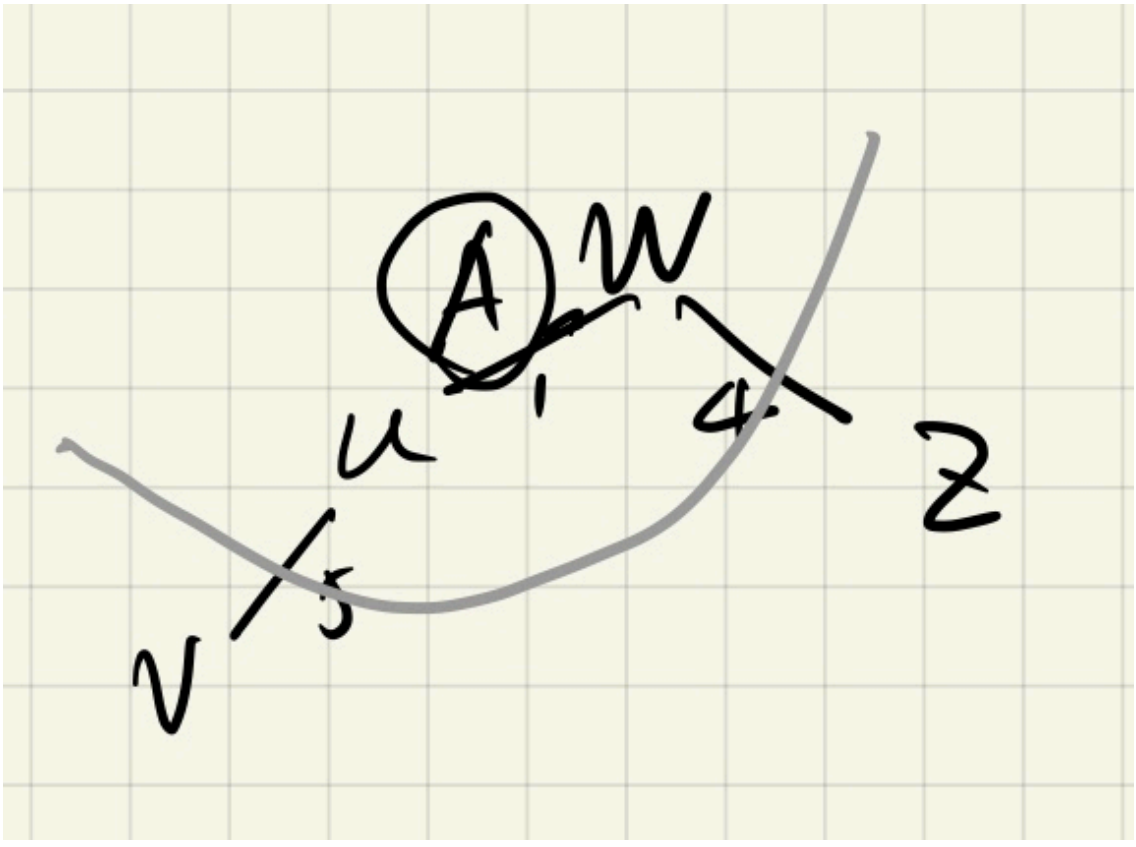
Assignment 6

Jiadao Zou --- jxz172230

Q1

23.1-2

> Professor Sabatier conjectures the following converse of Theorem 23.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.



Suppose we have w, u, z, v four vertices.

- let edge $A = (u, w)$ be S, so (u, w) respect the cut.
- Since the whole graph is a tree which itself could be a Minimum Spanning Tree, (u, v) should be a safe edge. Moreover, every edge is safe over there.
- To the "light edge", obviously, the (w, z) is smaller than (u, v). So, the conjunction is False.

23.1-3

> Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

For the specific MST, after removing (u, v) from it, we get two seperate tree T_1, T_2.

- Suppose (u, v) is not the light edge of some cut. Then, we could construct a new MST by using the light edge in this cut to connect T_1 and T_2. According to this, the new MST has smaller weight than the origin one, which contradicts the premise.

23.1-4

> Give a simple example of a connected graph such that the set of edges $\{(u, v)$: there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing $(S, V - S)\}$ does not form a minimum spanning tree.

We suppose G has 3 vertices. Each vertex connects with two others and all the 3 edges share the same weight.

- Obviously, every edge is a light edge since their weights are same.
- But, if we containing such above light edge, we get a cycle but not a tree.

Q2

23.2-8

> Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.
>
> Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

The algorithms fails.

- Since the partition is not well defined. Only makes difference of numbers of two sets be 1. So suppose we have a set $\{a, b, c\}$, where weight of $(a, b) = (a, c) = 1$, and $(b, c) = 5$. If the partition makes two sets $\{a\}$ and $\{b, c\}$.
- According to above algorithms, it won't give out the minimum spanning tree cause (b, c) is in the result tree.

Q3

23-4

> In this problem, we give pseudocode for three different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Also describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.

```
function MAYBE-MST-A(G,w)
    sort the edges into nonincreasing order of edge weight w
    T = E
    for each edge e, taken in nonincreasing order by weight w
        if T-{e} is a connected graph
            T = T-{e}
    return T
```

```
function MAYBE-MST-B(G,w)
    T = ∅
    for each edge e, taken in arbitrary order
        if Tu{e} has no cycles
            T = Tu{e}
    return T
```

```
function MAYBE-MST-C(G,w)
    T = ∅
    for each edge e, taken in arbitrary order
        T = Tu{e}
        if Tu{e} has a cycle c
            let e' be a maximum-weight edge on c
            T = T-{e'}
    return T
```

- (a)
  - The algorithms return the MST because it is Kruskal's Algorithm B. The algorithm only remove the edge won't affect the tree's connection, which means it only remove edge of

a cycle. While it remove edge in nonincreasing order, any of the other edges' weights would not larger than weight of the edge it removed.
    - To implement this algorithms, first we have to sort the edge in $O(E \lg E)$. Then for each edge, we use DFS to check whether $T - e$ is connected, that takes $O(|V| + |E|)$. The total cost would be $O(|E|^2)$ which will dominate the algorithms time complexity
- (b)
    - The algorithms will not return the MST because it doesn't arrange the weight in some order which could cause it discard lighter edge afterwards. For example, if we have $W_{(a,b)} = 3,\ W_{(a,c)} = 2,\ W_{(b,c)} = 1$, then if the algorithms takes the edge in order of above sequence, it would dismiss (b, c).
    - To implement this algorithms, just like Kruskal's Algorithms A, instead of replacing nondecreasing order to arbitrary order. Also, we have to keep a set recording current connected vertices. While the algorithms try to connect two components under same set would make a cycle. The time complexity is $O(|E|\alpha(|E|, |V|))$
- (c)
    - This algorithm works because it always try to remove the maximum weight in a cycle. The iteration would finally make a cycle becomes a tree. In some point, this algorithm serves like random version of Kruskal's algorithms B.
    - To implement this algorithms, just like question (b), except here it gonna find the maximum weight edge of a cycle. Using DFS in a cycle with $|V|$ vertices, it takes $O(|V|)$, because at that time, one cycle is the most this tree could have. So the time cost is $O(|E||V|)$.

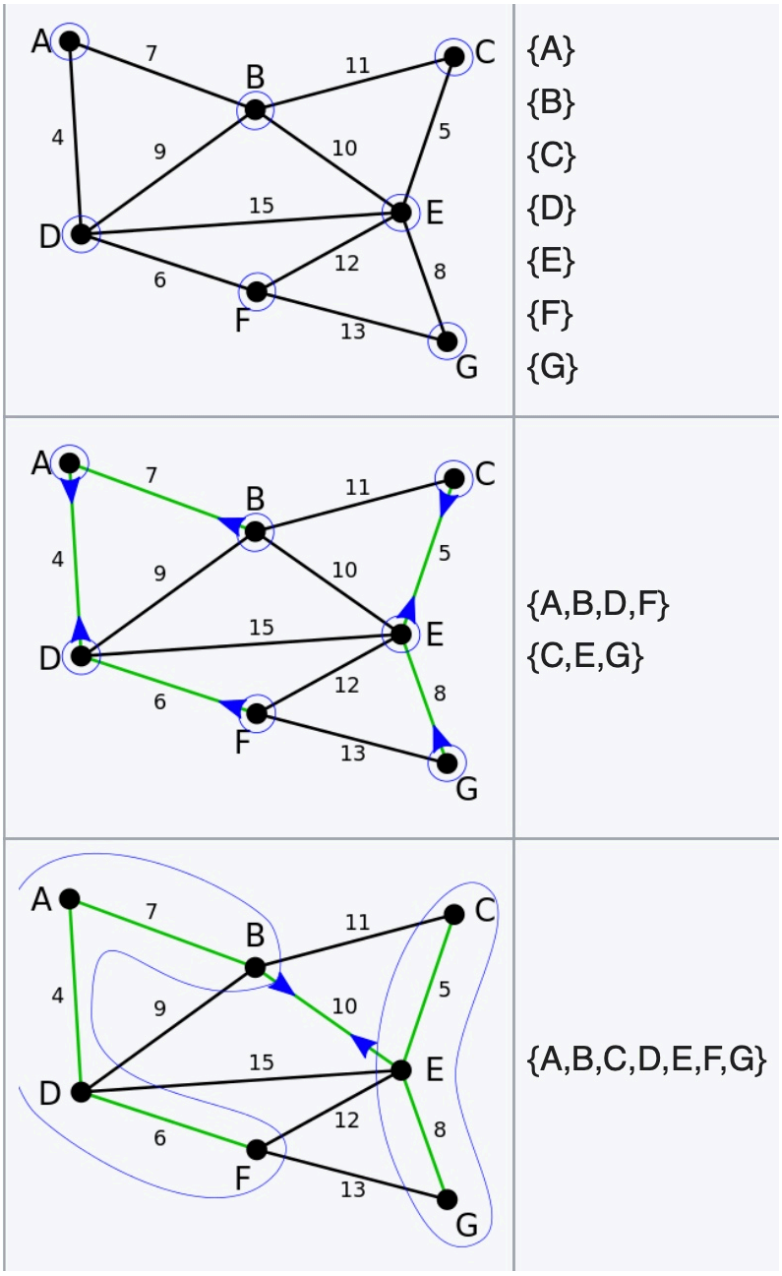
Q4

> Consider the minimum spanning tree problem on a connected undirected graph. Show that Boruvka algorithm produces a spanning tree if all weights are distinct (equally if they are totally ordered). Give a counter-example to show that if edges have equal weight, we may not get a spanning tree.

```
 function Boruvka-Algorithms:
     Input: A graph G whose edges have distinct weights
     Initialize a forest F to be a set of one-vertex trees, one for each vertex of
the graph.
     While F has more than one component:
       Find the connected components of F and label each vertex of G by its compone
nt
       Initialize the cheapest edge for each component to "None"
       For each edge uv of G:
         If u and v have different component labels:
           If uv is cheaper than the cheapest edge for the component of u:
             Set uv as the cheapest edge for the component of u
           If uv is cheaper than the cheapest edge for the component of v:
```

```
            Set uv as the cheapest edge for the component of v
        For each component whose cheapest edge is not "None":
          Add its cheapest edge to F
      Output: F is the minimum spanning forest of G.
```

- To prove the correctness of Boruvka Algorithm:



  - It looks like a divide-and-conquer version of Kruskal's Algorithms A, since each step, one component would connect to another one by the smallest weight edge if and only if those two components are disconnected.
  - At the first step, the vertex would only extend one edge, the smallest weight one out.
  - At the following steps till the end, when components try to connect with others, they would only extend one edge between two components. So, suppose we have n separate components, current step would only extend $n-1$ edges which is not enough for all the components connected while making a cycle. So, it won't generate any cycle when input is distinct edge.
- Counter example when edge has equal weights:
  - When we have three vertices with three equally weighted edges, at step 1, each component would select a different edges to make a cycle.

Q5

Let $G = [V, E]$ be a directed graph and $w[e]$ $(= w[u, v]$ if $e = (u, v))$ (not necessarily nonnegative) be weight on edge $e \in E$. Let K be a constant satisfying the condition that

$$r[e] = w[e] + K > 0 \qquad \forall e \in E$$

(a) Give an example to show that the shortest path in G from s to all other nodes depends on whether we use the weights $w[e]$ or $r[e]$.

(b) We know that lengths of the shortest path from s satisfy the relations:

$$\delta(s, v) \leq \delta(s, u) + w(u, v) \qquad \forall (u, v) \in E$$

Suppose $\{x_v\}; v \in V$ satisfy the relations:

$$x_v \leq x_u + w(u, v) \qquad \forall (u, v) \in E$$

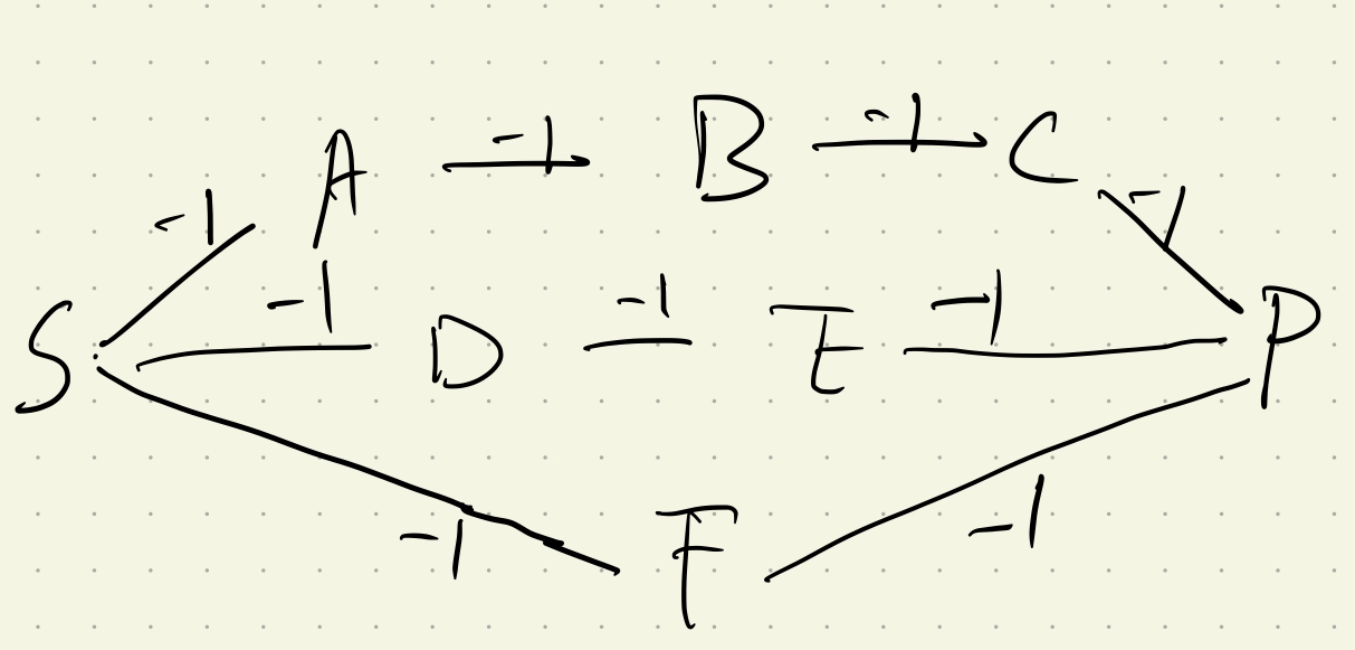
Does this imply $x_v = \delta(s, v)$ for all $v \in V$?

(c) Let $r[u, v] = w[u, v] + x_u - x_v$ for $(u, v) \in E$ with the above $\{x_v\}$. Now $r[u, v] \geq 0$ for all $(u, v) \in E$. So we can apply Dijkstra algorithm to the problem with r. Are these paths also shortest paths with w?

(d) In case (c), do we get to do less work in determining the shortest paths from s to all other nodes?

(e) In case (c), there was no mention of negative cycles in the problem with w – how come?

- (a)



- use $w[e]$, the shortest path is $S-A-B-C-P$
- use $r[e]$:
  - when $K=1$, all the paths are same,
    $S-A-B-C-P=S-D-E-P=S-F-P$.
  - When $k>1$, shortest path is $S-F-P$.

- (b)
  - No. Since by the definition, we only know that $x_v-x_u\leq w(u,v)$. Though we suppose $\forall i\in V, x_i>0$, we could only tells $x_v\leq w(u,v)$. So when $x_v\leq \min\limits_{u,v}(w(u,v))$, the equation always hold.
- (c)
  - Since from the definition, we know $r[u,v]=w(u,v)+x_u\geq x_v\geq 0$ is always increasing. So Dijkstra algorithm would works on this situation.
  - Suppose form starting vertex S till ending vertex P, the shortest path of r is:

$$\begin{aligned}Cost_r&=w(S,s_1)+x_S-x_{s_1}+w(s_1,s_2)+x_{s_1}-x_{s_2}+\ldots+w(S_n,P)+x_{s_n}-x_P\\&=\sum_{i=s_n,j=P}^{i=S,j=s_1}w(i,j)+x_S-x_P\end{aligned}$$

  As we could see, the cost of r has nothing to do with r, so this shortest path could also be append to w.
- (d)
  - Dijkstra algorithm's time complexity is $O((|E|+|V|)\cdot\lg|V|)=O(|E|\cdot\lg|V|)$. Thus, if there is any other algorithms faster than that, we has a faster implements. Situation depends on $|E|$ and $|V|$.
- (e)
  - According to question (c), for any cycle:

$$\begin{aligned}Cost_{cycle}&=w(c_1,c_2)+x_{c_1}-x_{c_2}+w(c_2,c_3)+x_{c_2}-x_{c_3}+\ldots+w(c_n,c_1)+x_{c_n}-x_{c_1}\\&=\sum_{i,i+1\in cycle}w(c_i,c_i+1)\end{aligned}$$

As long as $\forall w \geq 0$, we won't get a negative cycle.

Q6

> Suppose that both f and f' are flows in a network G and we compute flow $f \uparrow f'$. Does the augmented flow satisfy the flow conservation property? Does it satisfy the capacity constraint?

- The augmented flow satisfies the flow conservation property. To a graph, flow into a vertex and out of a vertex can be viewed as two sums, f and f'. If the parts are equally separated, their sums are also equal.
- The capacity constraint is not satisfied. Suppose we only have the vertices s and t, and have a single edge from s to t of capacity 1. If there is a flow of value 1 from s to t, however, augmenting this flow with itself ends up putting two units along the edge from s to t, which is greater than the capacity we can send.

Q7

26-1

> A$n \times n$ ***grid*** is an undirected graph consisting of n rows and n columns of vertices, as shown in Figure 26.11. We denote the vertex in the ith row and the jth column by (i, j). All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points (i, j) for which $i = 1$, $i = n$, $j = 1$, or $j = n$.
>
> Given $m \leq n^2$ starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ in the grid, the ***escape problem*** is to determine whether or not there are m vertex-disjoint paths from the starting points to any m different points on the boundary. For example, the grid in Figure 26.11(a) has an escape, but the grid in Figure 26.11(b) does not.
>
> **a.** Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show that determining the maximum flow in a network with edge and vertex capacities can be reduced to an ordinary maximum-flow problem on a flow network of comparable size.
>
> **b.** Describe an efficient algorithm to solve the escape problem, and analyze its running time.

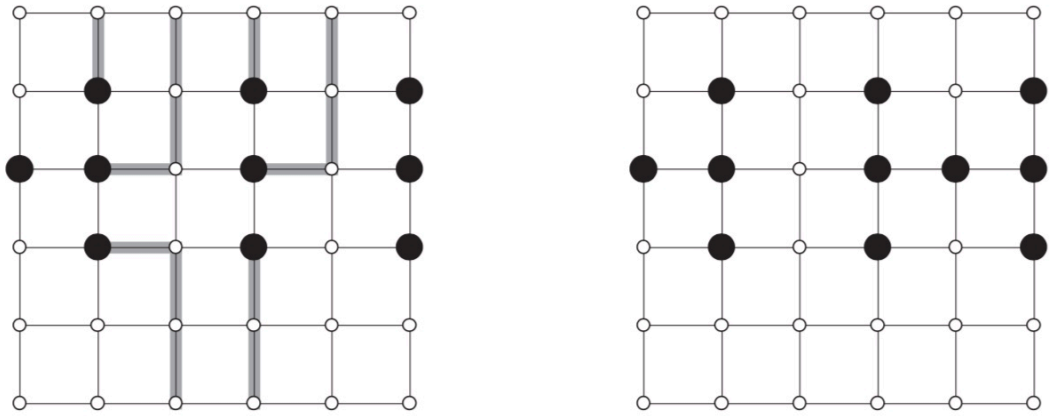
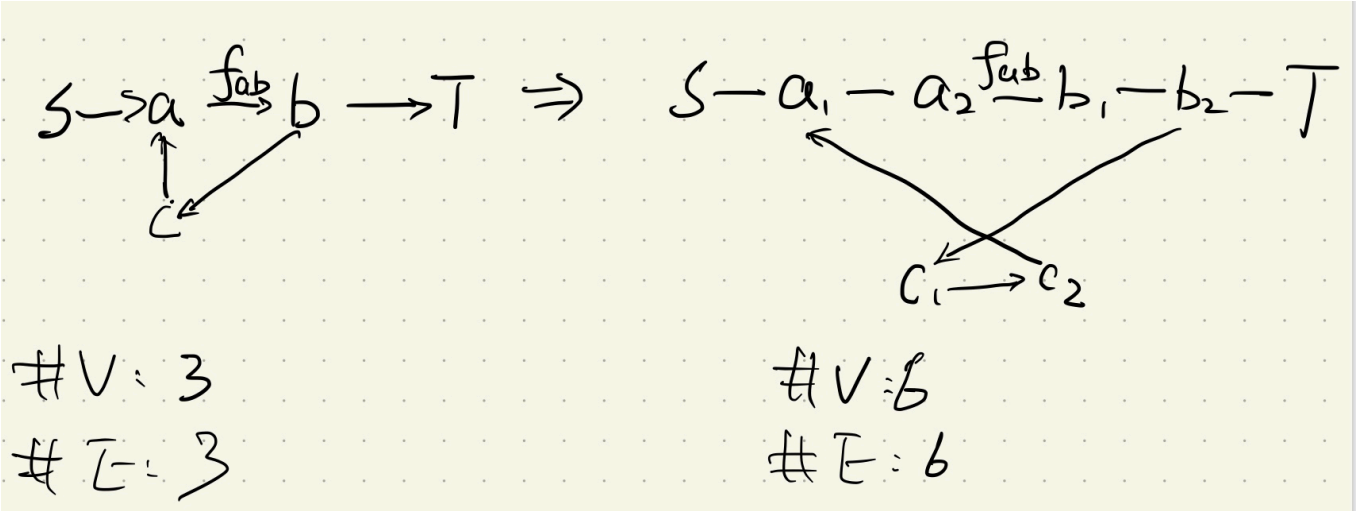


Figure 26.11 Grids for the escape problem. Starting points are black, and other grid vertices are white. **(a)** A grid with an escape, shown by shaded paths. **(b)** A grid with no escape.

- (a)
  - We can capture the vertex constraints by splitting out each vertex into two, where the edge between the two vertices is the vertex capacity. This new flow network will have $2|V|$ vertices and have $|V| + |E|$ edges.



- (b)
  - For each intersection of grid lines, we construct a vertex with 1 capacity. And for each pair of adjacent vertices in the grid, we construct a bidirectional edge with 1 capacity. Then, put a unit capacity edge going from s to each of the distinguished black vertices, and a unit capacity edge going from each vertex on the sides of the grid to t.
  - Thus, we implement the idea of path taking because all of the augmenting paths will be a unit flow, and every edge has unit capacity. We would have the escaping paths if the total flow is equal to m. And, if the max flow is less than m, the escape path is not present, because otherwise we could construct a flow with value m from it.

Q8

26-4

> Let $G = (V, E)$ be a flow network with source s, sink t, and integer capacities. Suppose that we are given a maximum flow in G.
>
> **a.** Suppose that we increase the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$ -time algorithm to update the maximum flow.
>
> **b.** Suppose that we decrease the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

- (a)
  - If there exists a minimum cut on which (u, v) doesn't lie then the maximum flow can't be increased.
  - Otherwise it does cross a minimum cut, and we can possibly increase the flow by 1 by finding an augmenting path from s to t and increasing the edge capacity by 1. For that, we use a BFS, which runs in $O(|V| + |E|)$.
- (b)
  - If the edge's flow was already at least 1 below capacity then nothing changes.
  - Otherwise, find the path from u to v by BFS. Decrease this direction's flow by 1 and increasing opposite direction flow (from v to u) by 1 to maintain the maximum flow.
  - As we have talked before, BFS takes $O(|V| + |E|)$ time.