به نام خدا

تمرین سری اول
درس نظریه و الگوریتم های گراف
دکتر فرزانه غیور باغبانی

فرزان رحمانی
۹۹۵۲۱۲۷۱

سوال اول

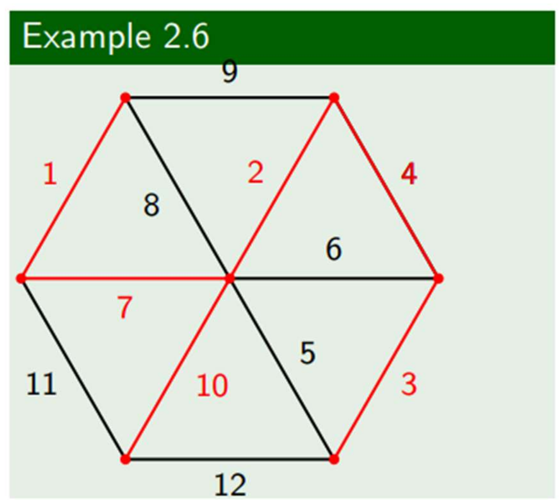
الگوریتم Kruskal برای پیدا کردن MST به شکل زیر است که آن را اجرا میکنیم:

Minimum Cost Spanning Tree

Suppose G is a graph and $c : E(G) \to \mathbb{N}$ is a cost function. The cost of a subgraph H of G is $\sum_{e \in E(H)} c(e)$. We want to find a minimum-cost spanning tree T of G.

Algorithm 2.5 (Kruskal)

- *Start with $V(T) = V(G)$ and $E(T) = \emptyset$.*
- *Order the edges of G so that their costs are non-decreasing.*
- *Proceed with each edge of G, one by one, in the above order: if its joins two components of T, add it to T; otherwise do nothing.*



حال با اجرای این الگوریتم به حل سوال می پردازیم.

Subject
Date

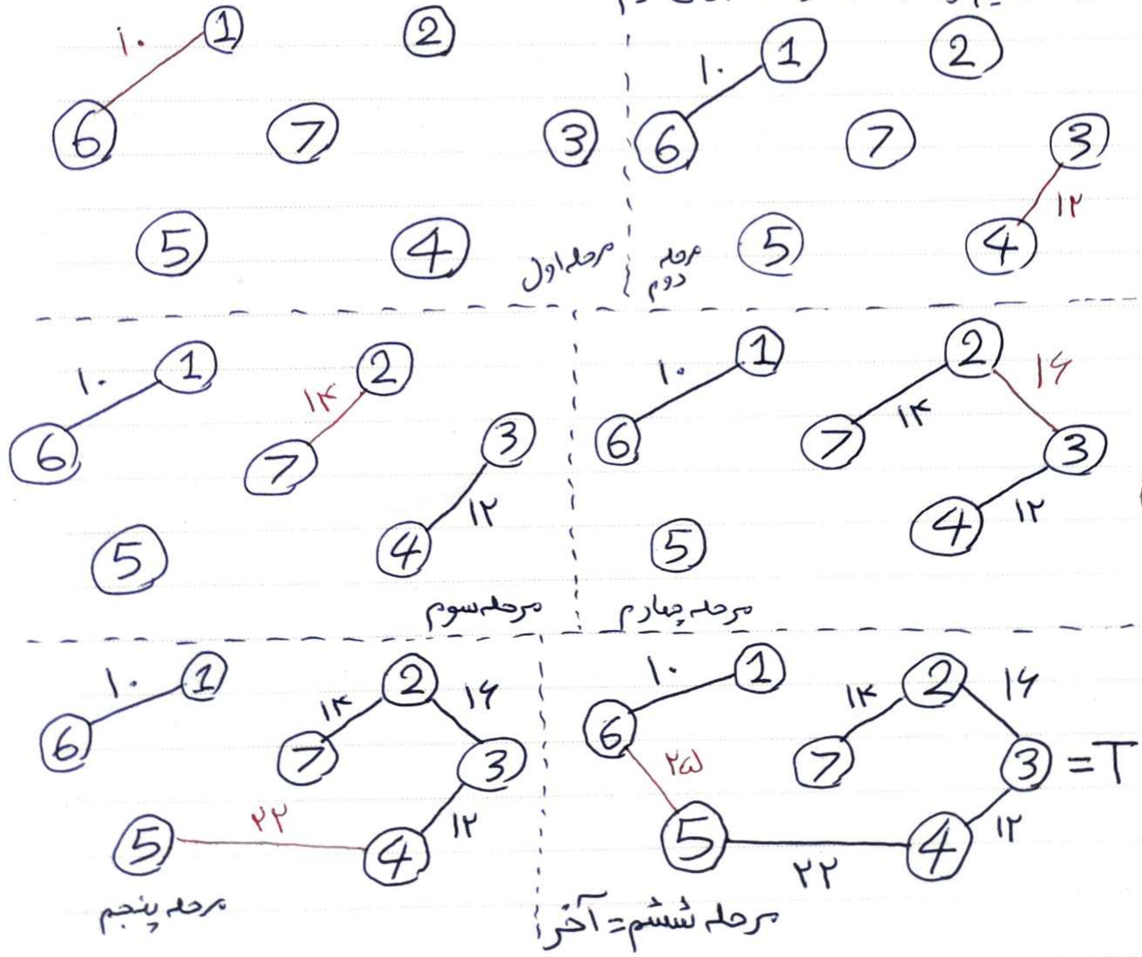
درخت کمینه پوشا را T می‌نامیم. $MST(G) = T$

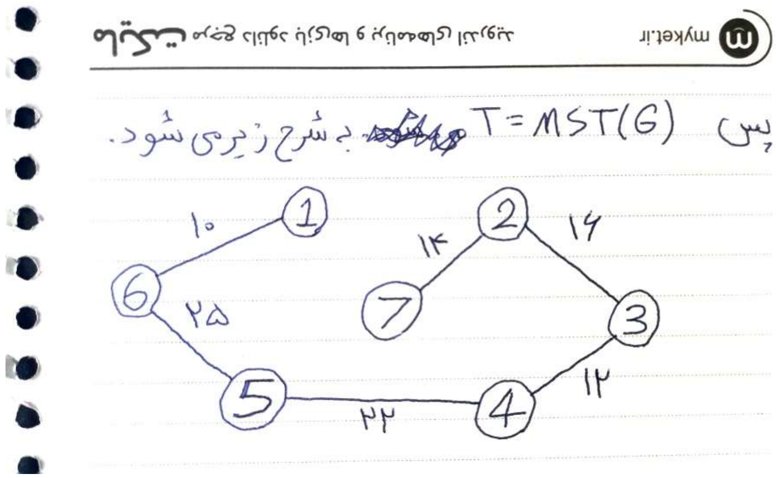
با $V(T)=V(G)$ و $E(T)=\varnothing$ شروع می‌کنیم.

حال یال های G را به صورت غیرنزولی مرتب می‌کنیم.

۱۰, ۱۲, ۱۴, ۱۶, ۱۸, ۲۲, ۲۴, ۲۵, ۲۸

حال الگوریتم را مرحله به مرحله اجرا می‌کنیم.





سوال دوم

الگوریتم آن به شرح زیر است:

Trees from Sequences

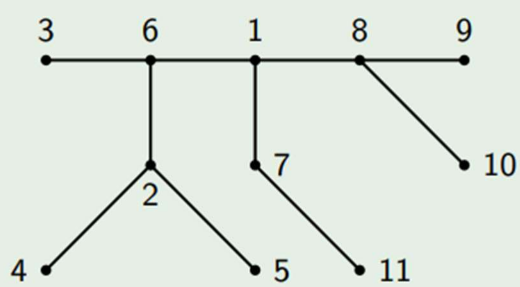
Now we describe how to produce a tree from a Prüfer sequence.

- Begin with a forest having n isolated vertices labeled 1, 2, $\ldots$, n.
- Proceed with all $n-2$ elements of the sequence, and, at the ith step,
  - let x be the label in position i.
  - let y be the smallest label that does not appear at the ith or later position and has not yet been marked as "finished".
  - add the edge xy, and
  - mark y as finished.
- Join the two remaining unfinished vertices with an edge.

Example 2.10

Sequence: 6, 2, 2, 6, 1, 8, 8, 1, 7

Finished: 3, 4, 5, 2, 6, 9, 10, 8, 1



حال به حل سوال با اجرای الگوریتم می پردازیم:

Prüfer Sequence: 1, 1, 1, 1, 6, 5

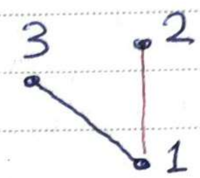
n-۲ = num of elements in sequence

n-۲ = ۴ → $\boxed{n=۸}$ → ۸ رأس داریم.

مرحله اول

Finished:

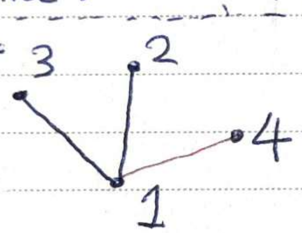
مرحله دوم



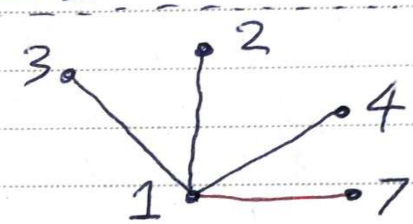
Finished: 2

مرحله دوم

مرحله سوم

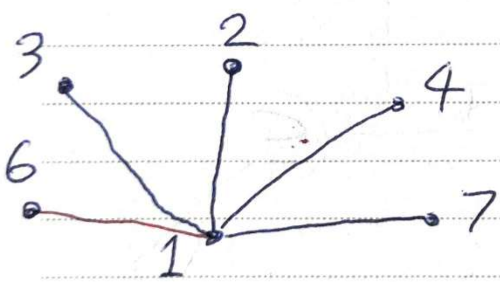


Finished: 2, 3



Finished: 2,3,4

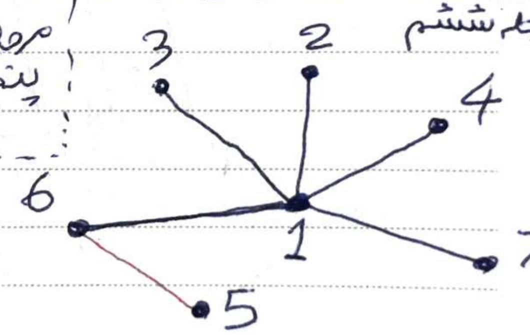
مرحله چهارم



Finished: 2, 3, 4, 7

مرحله پنجم

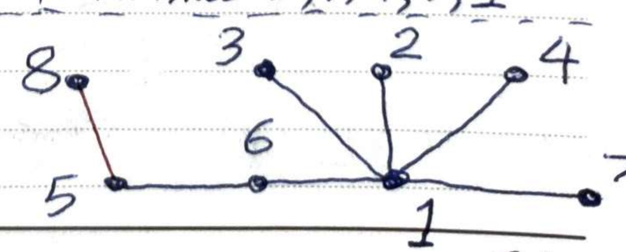
مرحله ششم



Finished: 2,3,4,7,1

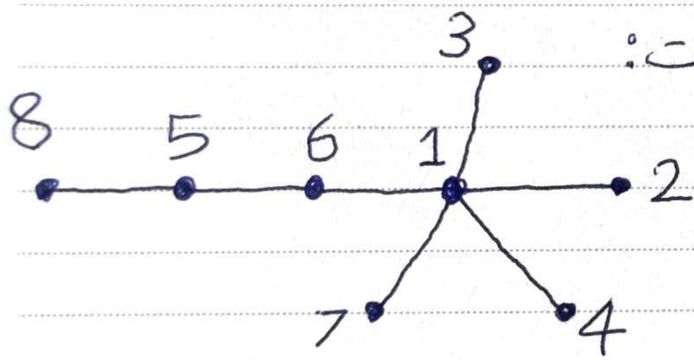
مرحله آخر = مرحله هفتم

Finished: 2,3,4,7,1,6



~~[illegible]~~ پس درخت متناظر با دنباله

Prufer به شکل زیر است:



سوال اول انگلیسی

ابتدا تعریف و روش پیدا کردن isomorphic را بیان میکنیم:

Isomorphism

Definition 1.7

The graphs $G_1 = (V_1, E_1, \mathcal{I}_1)$ and $G_2 = (V_2, E_2, \mathcal{I}_2)$ are isomorphic, written $G_1 \cong G_2$, if there are bijections $\varphi : V_1 \to V_2$ and $\psi : E_1 \to E_2$ such that $(v, e) \in \mathcal{I}_1$ if and only if $(\varphi(v), \psi(e)) \in \mathcal{I}_2$. Such a pair of bijections is an isomorphism.

Note 1.8

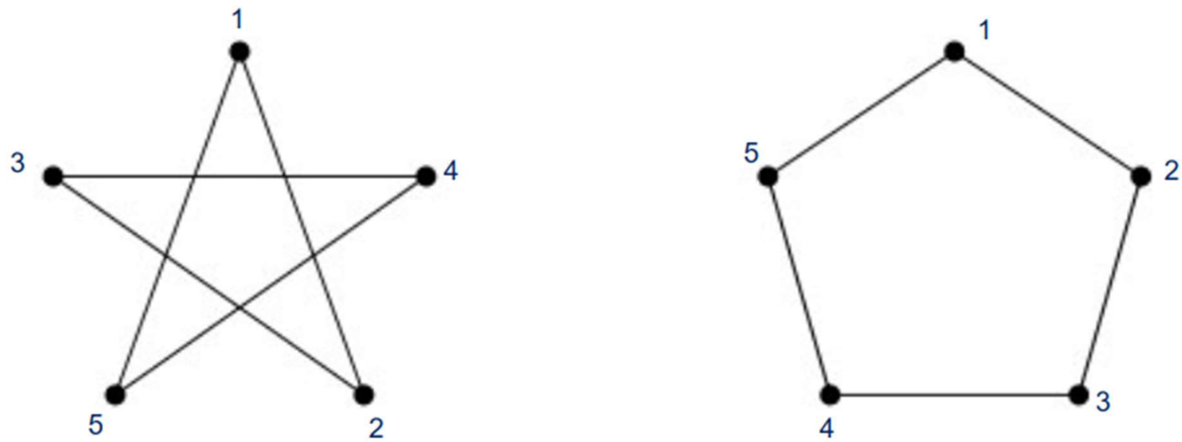
- *If G_1 and G_2 are simple, then an isomorphism may be defined as a bijection $\varphi : V_1 \to V_2$ such that u and v are adjacent in G_1 if and only if $\varphi(u)$ and $\varphi(v)$ are adjacent in G_2.* بین شون یال باشه
- *Isomorphic graphs are usually considered "the same".*

Theorem 1.9 (Babai, 2015–2016)

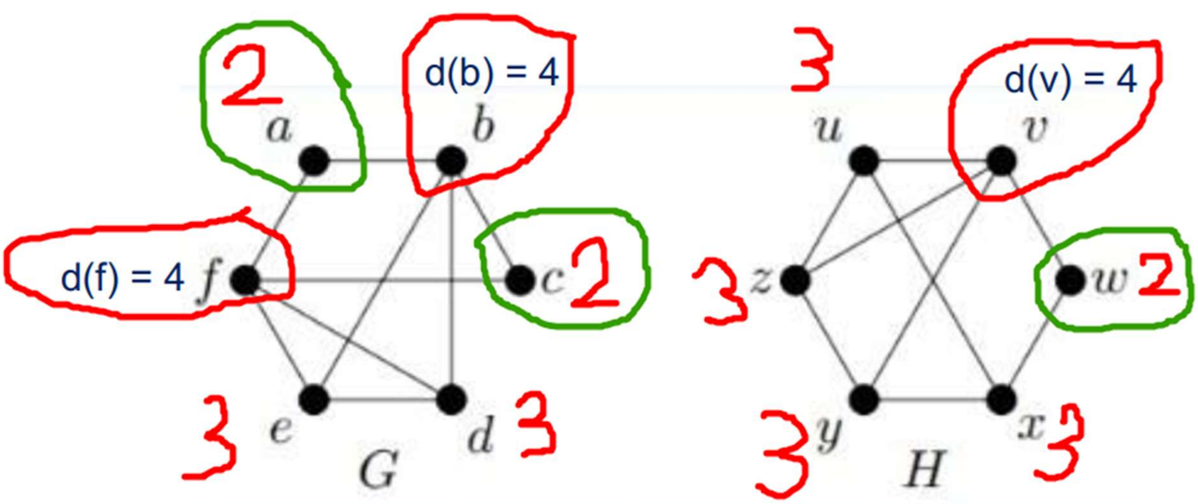
Graph isomorphism problem can be solved in quasi-polynomial time.
There is a constant c and an algorithm that can decide whether two graphs on n vertices are isomorphic or not in at most $2^{\mathcal{O}((\log n)^c)}$ steps.

حال هر یک را به صورت جدا بررسی میکنیم:

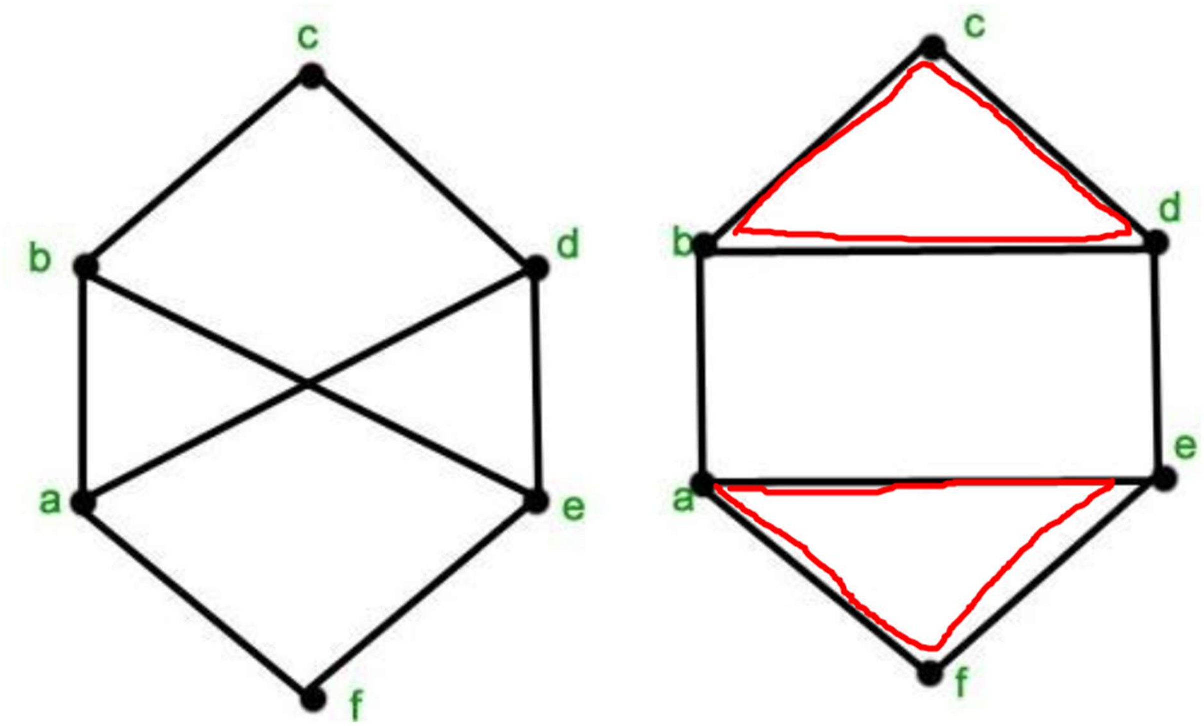
۱. Isomorphic هستند چرا که هر دو همان C5 هستند و با شماره گذاری رئوس به شکل زیر هر دو رأس در گراف سمت چپ مجاور هستند اگر و تنها اگر در گراف سمت راست مجاور باشند. (If G1 and G2 are simple, then an isomorphism may be defined as a bijection φ : V1 → V2 such that u and v are adjacent in G1 if and only if φ(u) and φ(v) are adjacent in G2.)



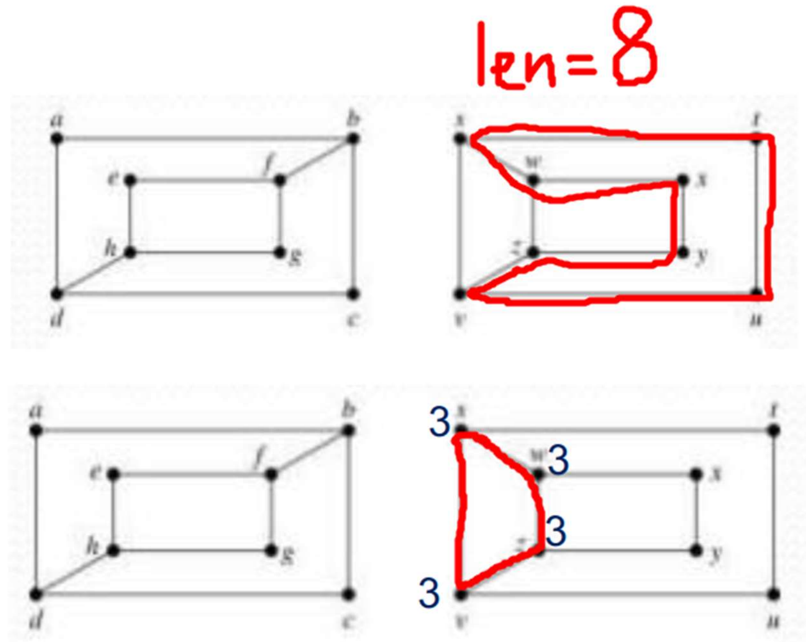
۲. Isomorphic نیستند. چرا که در گراف H فقط یک رأس با درجه ۴ (v) ولی در گراف G دو رأس با درجه ۴ (b, f) داریم. همچنین در گراف H فقط یک رأس با درجه ۲ (w) ولی در گراف G دو رأس با درجه ۲ (a, c) داریم. این برای درجه ۳ هم صدق می‌کند. بنابراین نمیتوانیم رئوس را طوری به هم نگاشت کنیم که هر دو رأس در گراف G مجاور هستند اگر و تنها اگر در گراف H مجاور باشند.



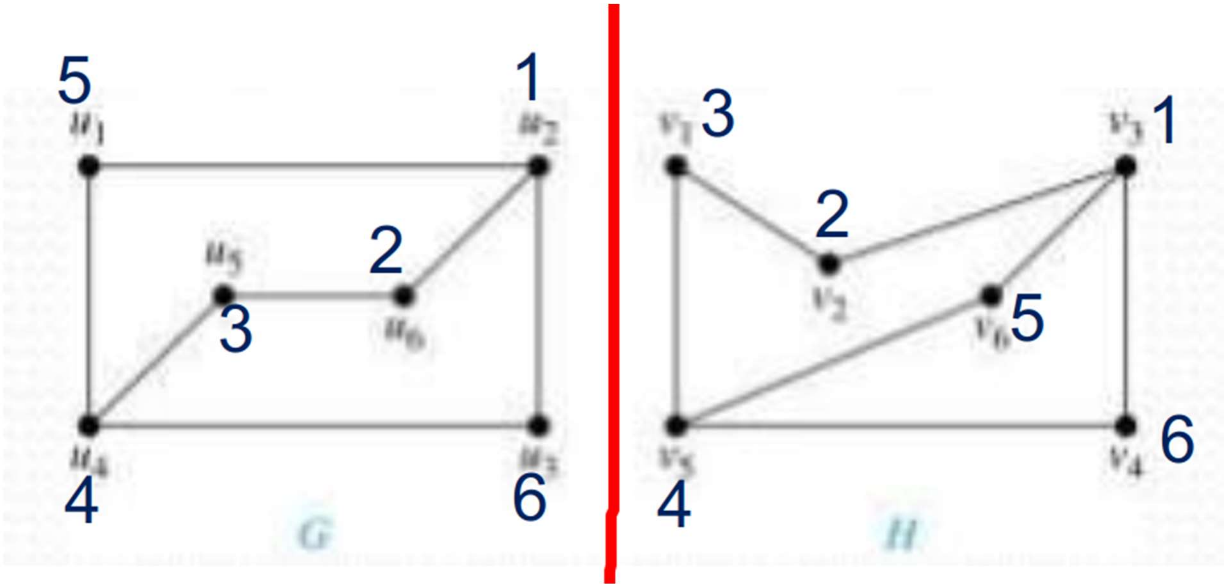
۳. Isomorphic نیستند. چراکه گراف سمت راست دارای دو دور به طول ۳ (cbdc, aefa) هست (گراف سمت راست دور به طول فرد دارد) ولی گراف سمت چپ دور به طول سه (فرد) ندارد.



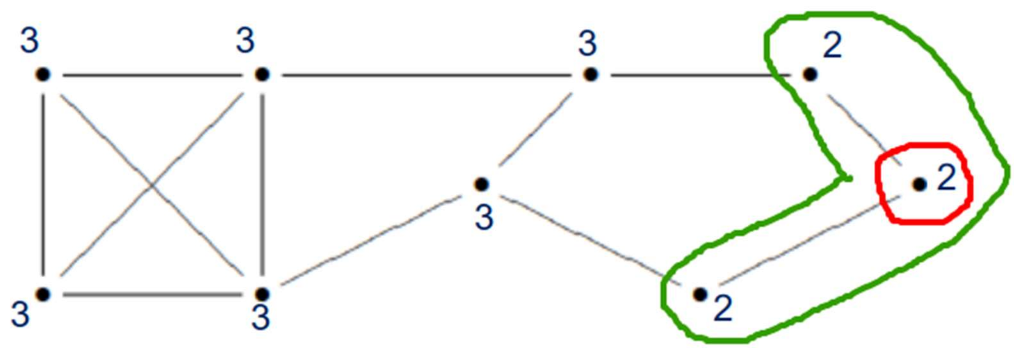
۴. Isomorphic نیستند. چراکه گراف سمت راست دارای دور به طول ۸ هست ولی گراف سمت چپ دور به طول ۸ ندارد. همچنین گراف سمت راست دارای یک دور به طول ۴ هست که درجه تمامی رئوس آن ۳ است ولی در گراف سمت چپ چنین دوری وجود ندارد.

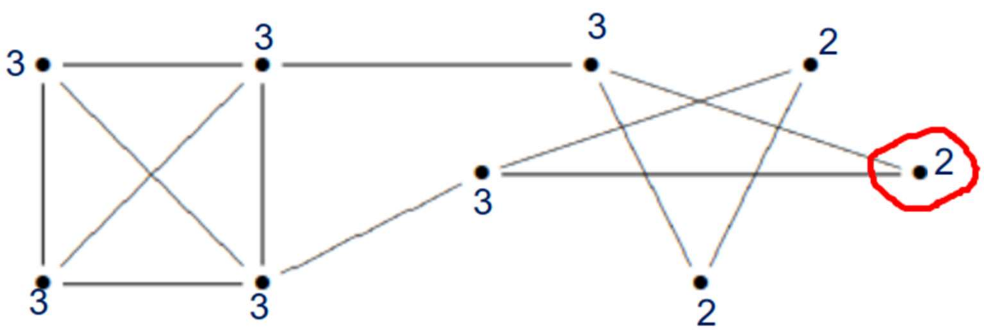


۵. Isomorphic هستند چرا که با شماره گذاری رئوس به شکل زیر هر دو رأس در گراف سمت چپ مجاور هستند اگر و تنها اگر در گراف سمت راست مجاور باشند. (If G_1 and G_2 are simple, then an isomorphism may be defined as a bijection $\varphi : V_1 \to V_2$ such that u and v are adjacent in G_1 if and only if $\varphi(u)$ and $\varphi(v)$ are adjacent in G_2.)

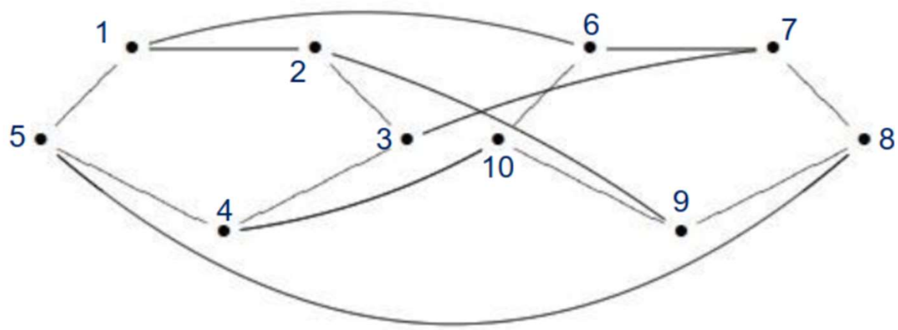


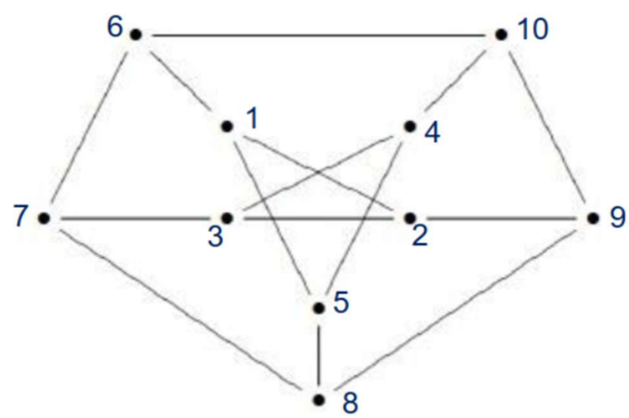
۶. Isomorphic نیستند. چرا که در گراف بالایی در شکل یک رأس وجود دارد که درجه آن ۲ است و دو همسایه دارد که درجه های آن‌ها ۲ است. (یک مسیر به شامل سه رأس با درجه ۲ دارد.)
ولی در گراف پایینی در شکل چنین چیزی برقرار نیست. سه رأس با درجه ۲ دارد. یکی از آنها دو همسایه با درجه ۳ دارد. دو رأس دیگر یک همسایه با درجه ۲ و یک همسایه با درجه ۳ دارند.





۷. Isomorphic هستند. چرا که با شماره گذاری رئوس به شکل زیر هر دو رأس در گراف سمت چپ مجاور هستند اگر و تنها اگر در گراف سمت راست مجاور باشند. (If $G1$ and $G2$ are simple, then an isomorphism may be defined as a bijection $\varphi : V1 \to V2$ such that u and v are adjacent in $G1$ if and only if $\varphi(u)$ and $\varphi(v)$ are adjacent in $G2$.)





سوال دوم انگلیسی

برای اثبات این گزاره روش های مختلفی وجود دارد. ما در زیر به هر یک از آنها اشاره خواهیم کرد:

اثبات با برهان خلف:

Proof idea: If a vertex is repeated, then part of the walk can be deleted so that a shorter walk from x to y is obtained. This shortest walk from x to y must be a path.

Proof. Let $P : (x = x_0); x_1; x_2; \ldots ; (x_k = y)$ be a shortest walk from x to y. If it is not a path, then there is a repeated vertex. Therefore there exist subscripts i and j such that $0 \le i < j \le k$ such that $x_i = x_j$. But then $(x = x_0); x_1; x_2; \ldots ; (x_i = x_j); x_{j+1}; \ldots ; (x_k = y)$ is a shorter walk from x to y, which is a contradiction. Therefore P is a path.

اثبات با استقرای ریاضی:

We use induction on the length of the walk.

Let $\boldsymbol{W}$ be a walk between $\boldsymbol{x}$ and $\boldsymbol{y}$.
Base step: if $|\boldsymbol{W}|=1|=1$, then $\boldsymbol{W}$ is just the edge $\boldsymbol{xy}$ and it is a $\boldsymbol{x}$-$\boldsymbol{y}$ path.
Induction step: Now assume the statement is true for all $\boldsymbol{a}$-$\boldsymbol{b}$ walks of smaller size than $\boldsymbol{W}$. If all the vertices in $\boldsymbol{W}$ are distinct, then $\boldsymbol{W}$ is $\boldsymbol{x}$-$\boldsymbol{y}$ path and we are done. Otherwise, $\boldsymbol{W}$ has a repeated vertex say $\boldsymbol{u}$. Let $\boldsymbol{W'}$ be the walk obtained by suppressing the section of $\boldsymbol{W}$ between the two repetition of $\boldsymbol{u}$. Obviously $\boldsymbol{W'}$ is $\boldsymbol{x}$-$\boldsymbol{y}$ walk of smaller length than $\boldsymbol{W}$. By induction hypothesis, $\boldsymbol{W'}$ has $\boldsymbol{x}$-$\boldsymbol{y}$ path which means that $\boldsymbol{W}$ has $\boldsymbol{x}$-$\boldsymbol{y}$ path.

اثبات با روش(الگوریتم) سازنده:

We can also do it using the Constructive method:

Our algorithm input = walk (sequence of edges and vertices) Our algorithm output = path (distinct sequence of edges and vertices)

1. pick an element (starting from index 0) and compare it with all other elements in the string*
2. if any repeating element is found, then remove the complete sub-array between repetitive elements and also remove one of the repetitive elements.
3. repeat step 3 until the last element is picked.

*strings are arrays of characters

مراجع:

https://www.math.uvic.ca/faculty/gmacgill/guide/M222Graphs.pdf
https://math.stackexchange.com/questions/699765/prove-that-if-there-is-a-walk-from-u-to-v-then-there-is-also-a-path-from-u-to-v

سوال عملی

ابتدا به طور مختصر الگوریتم Kruskal و جزئیات پیاده سازی را توضیح می‌دهیم.

Kruskal's Algorithm

- Algorithm: repeatedly add to X the next lightest edge e that doesn't produce a cycle
- At any point of time, the set X is a forest, that is, a collection of trees
- The next edge e connects two different trees—say, T_1 and T_2
- The edge e is the lightest between T_1 and $V - T_1$, hence adding e is safe

Implementation Details

- use disjoint sets data structure
- initially, each vertex lies in a separate set
- each set is the set of vertices of a connected component
- to check whether the current edge $\{u, v\}$ produces a cycle, we check whether u and v belong to the same set

در ادامه شبه کد(pseudocode) این الگوریتم آمده است:

Kruskal(G) using DSU: Disjoint Sets Union

```
for all u ∈ V:
  MakeSet(v)
X ← empty set
sort the edges E by weight
for all {u, v} ∈ E in non-decreasing
  weight order:
  if Find(u) ≠ Find(v):
    add {u, v} to X
    Union(u, v)
return X
```

پیچیدگی محاسباتی(Time Complexity) این الگوریتم نیز به شرح زیر است:

Running Time

- Sorting edges: at most

$$O(|E|\log|E|) = O(|E|\log|V|^2) = O(2|E|\log|V|) = O(|E|\log|V|)$$

- Processing edges: connected

Min: $|V|-1$

Max: $|V|^2$

$$2|E| \cdot T(\texttt{Find}) + |V| \cdot T(\texttt{Union}) = O((|E|+|V|)\log^{*}|V|) = O(|E|\log|V|)$$

heuristic

- Total running time: $O(|E|\log|V|)$

ابتدا این سوال را با زبان پایتون پیاده‌سازی کردیم ولی به مشکل time limit exceeded برخوردیم. سپس آن را با زبان C پیاده سازی نمودیم. در فایل MST.c تمامی خطوط، struct ها و توابع را به طور کامل توضیح دادیم که می‌توانید مشاهده کنید. با این حال به طور مختصر روند کار را در اینجا به شکل مختصر توضیح می‌دهیم.

این کد مربوط به پیاده سازی الگوریتم کروسکال برای پیدا کردن کم هزینه ترین درخت پوشا در یک گراف وزن دار است.

تعریف ساختارها:

- **Edge (لبه):** این ساختار برای نمایش یال‌های گراف استفاده می‌شود. شامل سه فیلد است:
  - u: راس اول یال
  - v: راس دوم یال
  - w: وزن یال
- **Graph (گراف):** این ساختار برای نمایش کل گراف به کار می‌رود. شامل سه فیلد است:
  - V: تعداد راس‌های گراف
  - E: تعداد یال‌های گراف
  - edge: آرایه‌ای از یال‌های گراف

توابع کمکی:

- **createGraph(V, E):** این تابع یک گراف جدید با V راس و E یال ایجاد می‌کند و یک اشاره گر به آن برمی‌گرداند.
- **find(parent, i):** این تابع برای پیاده‌سازی مجموعه‌های مجزا[1] به کار می‌رود. با گرفتن یک راس (i) به عنوان ورودی، مجموعه (یا نماینده) آن راس را پیدا می‌کند.
- **Union(parent, rank, x, y):** این تابع نیز برای مجموعه‌های مجزا است. با گرفتن دو راس (x و y) به عنوان ورودی، مجموعه‌های آن‌ها را با هم ادغام می‌کند.
- **compare(a, b):** این تابع برای مرتب کردن یال‌ها بر اساس وزنشان در الگوریتم quicksort استفاده می‌شود. دو یال را به عنوان ورودی می‌گیرد و بر اساس وزنشان (کدام بزرگتر است) خروجی می‌دهد.

تابع اصلی (kruskalMST):

این تابع الگوریتم کروسکال را برای پیدا کردن کم هزینه ترین درخت پوشا گراف اجرا می‌کند. مراحل کار به شرح زیر است:

۱. مرتب کردن یال‌های گراف بر اساس وزن به شکل غیر نزولی (از کمترین به بیشترین)

۲. ایجاد آرایه‌ای برای ذخیره کردن یال‌های درخت پوشا

۳. ایجاد و مقداردهی اولیه آرایه‌های parent و rank برای مجموعه‌های مجزا (هر راس در ابتدا یک مجموعه جداگانه است)

٤. پیدا کردن یال‌ها تا زمانی که تعداد یال‌های درخت پوشا به V-1 برسد (کمترین درخت پوشا شامل V-1 یال است)

- بررسی اینکه آیا اضافه کردن یال جاری باعث ایجاد دور در درخت پوشا می‌شود (با استفاده از مجموعه‌های مجزا)

[1] Disjoint Sets

- در صورتی که دور ایجاد نمی‌شود، یال به درخت پوشا اضافه شده و مجموعه‌های راس‌های یال با هم ادغام می‌شوند.

۵. محاسبه و برگرداندن وزن کل یال‌های درخت پوشا

تابع main:

این تابع به عنوان نقطه شروع برنامه عمل می‌کند. وظایف آن به شرح زیر است:

۱. خواندن تعداد راس‌ها و یال‌های گراف از ورودی

۲. ایجاد گراف

۳. خواندن اطلاعات یال‌ها (راس‌های مبدا و مقصد و وزن) از ورودی

٤. صدا زدن تابع kruskalMST برای پیدا کردن کمترین درخت پوشا

۵. چاپ وزن کل یال‌های درخت پوشا

نکات مهم:

- این کد از ساختار داده‌های مجموعه‌های مجزا برای اطمینان از اینکه هیچ دوری در درخت پوشا ایجاد نمی‌شود، استفاده می‌کند.
- یال‌ها بر اساس وزنشان از کمترین به بیشترین مرتب می‌شوند تا اطمینان حاصل شود که کمترین درخت پوشا پیدا می‌شود.

مرجع:

https://www.coursera.org/specializations/data-structures-algorithms

همچنین در ادامه کد پیاده سازی شده را مشاهده می‌کنید.

```
#include <stdio.h> // printf, scanf
#include <stdlib.h> // malloc, free, qsort

struct Edge {
    /* u, v: vertices of the edge
     * w: weight of the edge
     */
    int u, v, w;
};

struct Graph {
    /* V: number of vertices
     * E: number of edges
     * edge: array of edges of the graph (pointer to the first element of the
array)
     */
    int V, E;
    struct Edge* edge;
};
```

```c
struct Graph* createGraph(int V, int E) {
    /* Allocates memory for a graph with V vertices and E edges
     * and returns a pointer to the graph
     */
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph)); //
Allocates memory for the graph
    graph->V = V; // Sets the number of vertices
    graph->E = E; // Sets the number of edges
    graph->edge = (struct Edge*) malloc(graph->E * sizeof(struct Edge));
    return graph; // Returns a pointer to the graph
}

int find(int parent[], int i) {
    /* disjoint set find operation
     * Finds the set of the element i
     * and performs path compression
     * using disjoint set data structure
     */
    if (parent[i] == i) // If i is the root
        return i; // Returns i
    return find(parent, parent[i]); // Else returns the root of the set of i with
recursive call
}

void Union(int parent[], int rank[], int x, int y) {
    /* disjoint set union operation
     * Unites the sets of x and y
     * using disjoint set data structure
     */
    int xroot = find(parent, x); // Finds the set of x
    int yroot = find(parent, y); // Finds the set of y

    if (rank[xroot] < rank[yroot]) // If the rank of xroot is less than the rank
of yroot
        parent[xroot] = yroot; // Then yroot becomes the parent of xroot
    else if (rank[xroot] > rank[yroot]) // If the rank of xroot is greater than
the rank of yroot
        parent[yroot] = xroot; // Then xroot becomes the parent of yroot
    else { // If the ranks are the same
        parent[yroot] = xroot; // Then yroot becomes the parent of xroot
        rank[xroot]++; // And the rank of xroot increases by 1
    }
}
```

```c
int compare(const void* a, const void* b) {
    /* Comparison function for qsort
     * Compares the weights of two edges
     */
    struct Edge* a1 = (struct Edge*)a; // Casts a to a pointer to an Edge
    struct Edge* b1 = (struct Edge*)b; // Casts b to a pointer to an Edge
    return a1->w > b1->w; // Returns 1 if the weight of a is greater than the
weight of b, 0 otherwise
}

int kruskalMST(struct Graph* graph) {
    /* Kruskal's algorithm for finding the minimum spanning tree of a graph
     * Returns the weight of the minimum spanning tree
     * using disjoint set data structure and qsort
     */
    int V = graph->V; // Number of vertices
    struct Edge result[V]; // Array of edges of the minimum spanning tree
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compare); // Sorts the
edges of the graph by weight using qsort(quicksort)

    int* parent = (int*) malloc(V * sizeof(int)); // Array of parents of the
vertices
    int* rank = (int*) malloc(V * sizeof(int)); // Array of ranks of the vertices
    for (int v = 0; v < V; ++v) { // Initializes the arrays
        parent[v] = v; // The parent of each vertex is itself
        rank[v] = 0; // The rank of each vertex is 0
    }

    int e = 0; // Number of edges of the minimum spanning tree
    int i = 0; // Index of the edge being considered
    while (e < V - 1 && i < graph->E) { // While the minimum spanning tree has
less than V - 1 edges and there are edges left
        struct Edge next_edge = graph->edge[i++]; // The next edge to be
considered
        int x = find(parent, next_edge.u); // The set of the first vertex of the
edge
        int y = find(parent, next_edge.v); // The set of the second vertex of the
edge

        if (x != y) { // If the vertices are not in the same set
            result[e++] = next_edge; // The edge is added to the minimum spanning
tree
            Union(parent, rank, x, y); // The sets of the vertices are united
        }
    }
```

```c
    int minCost = 0; // Weight of the minimum spanning tree
    for (i = 0; i < e; ++i) // Calculates the weight of the minimum spanning tree
        minCost += result[i].w; // Adds the weight of the edge to the weight of
the minimum spanning tree
    return minCost; // Returns the weight of the minimum spanning tree
}

int main() {
    /* Reads the input and calls the kruskalMST function
     * Prints the weight of the minimum spanning tree
     */
    int V, E; // Number of vertices and edges
    scanf("%d %d", &V, &E); // Reads the number of vertices and edges

    struct Graph* graph = createGraph(V, E); // Creates a graph with V vertices
and E edges

    for (int i = 0; i < E; ++i) { // Reads the edges
        scanf("%d %d %d", &graph->edge[i].u, &graph->edge[i].v, &graph-
>edge[i].w); // Reads the vertices and weight of the edge
        graph->edge[i].u--; // Vertices are 0-indexed (0 <= u, v < V)
        graph->edge[i].v--; // Vertices are 0-indexed (the reason for the
decrement)
    }

    printf("%d\n", kruskalMST(graph)); // Calls the kruskalMST function and
prints the weight of the minimum spanning tree

    return 0; // Returns 0 to the operating system
}
```

پایان