UNIVERSIDAD CATÓLICA SAN PABLO

PROGRAMACIÓN COMPETITIVA

Problemas de flujo máximo, Sweep Line y Algoritmos de String

Mauricio Carazas Segovia Ángel Loayza Huarachi
Luis Huachaca Vargas Hugo Manchego Paredes

28 November 2022

Abstract

En este artículo presenta la descripción de 8 algoritmos que cubren el problema del flujo máximo, Sweep Line, y Algoritmos de String. Asimismo, cada explicación del algoritmo traerá un ejemplo en un problema de programación competitiva, aplicando dicho algoritmo en su solución.

1 Algoritmos de flujo de grafos

1.1 Algoritmo de Dinic

1.1.1 Descripción del algoritmo

Propuesto por Yefim Dinitz en 1969, es un algoritmo polinomial para resolver el problema de flujo máximo con tiempo de ejecución O(V*V*E), que incluye conceptos como nivel de grafo, flujo bloqueo y usa diferentes técnicas de recorrer un grafo, incluyendo DFS y BFS, es un algoritmo que trabaja mejor en grafos bipartitos y es capaza de manejar grandes proporciones de grafos.

Un dato curioso es que el algoritmo se popularizó como 'Dinic's' por una mala pronunciación de Shimon Even, un profesor en computación.

El punto básico para entender la idea del algoritmo es esta situación, imagina que tienes una cita con un amigo en una cafetería cercana, sin embargo, no sabes exactamente el lugar al que debes ir, lo único que sabes es que está al este, una persona común y corriente lo que haría es preguntar o averiguar donde queda exactamente. Pero tú eres programador y te gusta hacer las cosas por ti solo, el paso que seguirías es tomar la dirección del este, al final, estarás un poco arriba o un poco abajo, pero cerca de tu objetivo.

Sin embargo, sigue habiendo conceptos adicionales en el Algoritmo de Dinic que es importante mencionar.

1.1.2 Grafo de flujo residual

Un grafo de flujo es un grafo dirigido con aristas que tienen la cierta capacidad máxima, cada una tiene un valor de flujo asociado, indica cuanto contenido(agua en un sistema de drenaje) pasa en la arista.

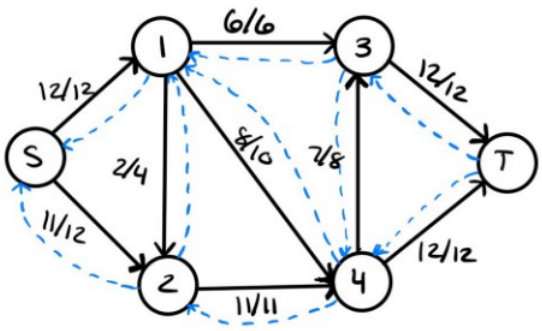

Figure 1: Grafo residuo (Aristas residuos en azul).

Un grafo es residual si nosotros podemos modificar el flujo desde ciertos puntos con el objetivo de obtener la mayor cantidad de flujo en el nodo final. Los nodos de flujo tienen un nodo raíz(s), donde sale el flujo, y un nodo sumidero(t) donde termina el flujo.

1.1.3 Ruta aumentada

Una ruta aumentada es la ruta de la arista en un grafo residual, con capacidad mayor que 0 desde "s" hasta "t". Significa que hay un recorrido que no está lleno, es decir, tiene espacio para alguna carga más. Siempre habrá un camino que tenga cuello de botella, que será la arista más chica.

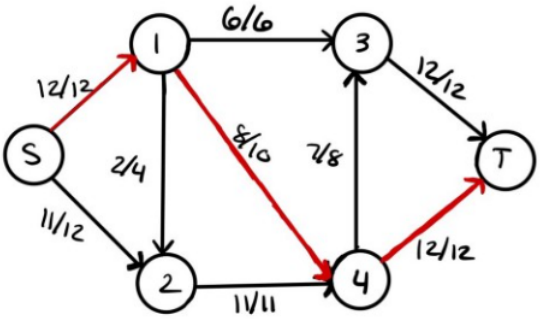

Figure 2: Ruta aumentada (en rojo).

Cuanto recorramos el grafo necesitaremos aumentar el valor de flujo por cada iteración, sin embargo, en caso estemos en una situación donde debamos volver, no tenemos por qué tener problemas, es simplemente un backtrack, para solucionar el problema.

1.1.4 Nivel de grafo

Asignamos un nivel a cada nodo, los cuales podemos lograr con un BFS, desde el nodo raíz hasta los demás, esto nos ayuda para evitar retrocesos(backtracking) innecesarios, estos vértices son deben tener capacidad mayor a 0, porque la capacidad restante de una arista es calculado por su capacidad menos el flujo sobre este.

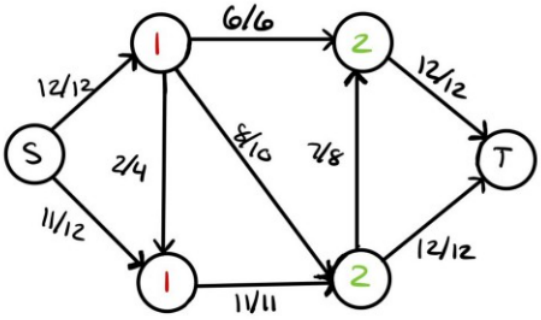

Figure 3: Grafo con niveles (marcados de colores).

Este algoritmo cambió la forma de ver los problemas de flujo máximo, ya que introdujo conceptos nuevos, por ello es una clara mejora a otros, además de tener un tiempo de ejecución más que aceptable.

1.1.5 Codigo Dinic

```
#include <iostream>
#include <atcoder/maxflow>
using namespace std;
typedef long long ll;
const ll inf = 1e18;
int h, w;
ll a[105][105];
ll rsum[105], csum[105];
int main(void)
{
  cin >> h >> w;
  for(int i = 1; i <= h; i++) for(int j = 1; j <= w; j++) cin >>
      a[i][j];
  atcoder::mf_graph<ll> g(h+w+2);
  int S = 0, T = h+w+1; ll ans = 0;
  for(int i = 1; i <= h; i++){
    for(int j = 1; j <= w; j++){
      if(a[i][j] >= 0) g.add_edge(i, h+j, a[i][j]), ans += a[i][j];
      else g.add_edge(h+j, i, inf), rsum[i] -= a[i][j], csum[j] -=
          a[i][j];
    }
  }
  for(int i = 1; i <= h; i++) g.add_edge(S, i, rsum[i]);
  for(int j = 1; j <= w; j++) g.add_edge(h+j, T, csum[j]);
  cout << ans - g.flow(S, T) << endl;
  return 0;
}
```

1.2 Algoritmo Ford-Fulkerson

1.2.1 Descripción del algoritmo

El Algoritmo de Ford-Fulkerson calcula un flujo máximo de manera iterativa, comenzando con un flujo y luego haciendo ajustes que cumplen con las restricciones y aumenten el flujo. Es un algoritmo que soluciona de problema con las redes que transportan algún tipo de recurso de un punto a otro, como puede ser agua, electricidad, datos, etc.

En general, este tipo de modelo se denomina problema de flujo. Este se define por el grafo que representa la estructura de la red. Para cada arista del grafo, es necesario especificar la cantidad máxima de recursos que se pueden desplazar a lo largo de él, su capacidad.

El algoritmo comienza con un flujo vacío y luego encuentra distintas rutas en el grafo, desde el origen (source) hasta el destino (sink). Añadiendo el flujo necesario a lo largo del camino hasta saturar uno de los nodos, que es el que tiene la capacidad más baja, mantiene cumplidas las restricciones sobre el flujo y lo aumenta, en caso sea válido. Estos se llaman caminos de aumento (augmenting paths).

- Algunos lo denominan método, ya que abordaje e para encontrar los caminos de aumento en un grafo no fue completamente especificado.
- Tiene una complejidad de O(|E| x f), donde "f" es el flujo máximo, este mejora si se implementa usando BFC para encontrar los caminos de aumento O(V^2 x E), V siendo los vértices y E las aristas.

1.2.2 Código Ford-Fulkerson

```
#include <iostream>
#include <queue>
#include <cstring>
#define MAX 101
using namespace std;

int graph[MAX][MAX];

bool DFS(int rGraph[MAX][MAX], int* parent, bool* visited, int n, int
    x, int t) {
  if (x == t) return true;
  visited[x] = true;
  for (int i = 0; i < n; ++i) {
    if (rGraph[x][i] > 0 && !visited[i]) {
      parent[i] = x;
      if (DFS(rGraph, parent, visited, n, i, t))
        return true;
    }
  }
  return false;
}

int FordFulkerson(int n, int s, int t) {

  int rGraph[MAX][MAX];
  int parent[MAX];
  bool visited[MAX];

  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      rGraph[i][j] = graph[i][j];

      int max_flow = 0;

      while (DFS(rGraph, parent, visited, n, s, t)) {
        // while (DFS(rGraph, parent, visited, n, s, t)){
        memset(visited, false, sizeof(visited));
        int path_flow = 0xfffffff;
        for (int v = t; v != s; v = parent[v]) {
          int u = parent[v];
          path_flow = min(path_flow, rGraph[u][v]);
        }
        for (int v = t; v != s; v = parent[v]) {
          int u = parent[v];
          rGraph[u][v] -= path_flow;
          rGraph[v][u] += path_flow;
        }
        max_flow += path_flow;
      }
      return max_flow;
}

void addEdge(int ini, int fin, int capacidad){
  graph[ini][fin] = capacidad;
```

```
}

int main (){
  int V = 6;

  addEdge(0, 1, 16);
  addEdge(0, 2, 13);
  addEdge(1, 2, 10);
  addEdge(2, 1, 4);
  addEdge(1, 3, 12);
  addEdge(2, 4, 14);
  addEdge(3, 2, 9);
  addEdge(3, 5, 20);
  addEdge(4, 3, 7);
  addEdge(4, 5, 4);

  int s = 0, t = 5;
  cout << "Max Flow: " << FordFulkerson(V, s, t) << endl;
}
```

1.3 Algoritmo Edmonds-Karp

1.3.1 Descripción del algoritmo

El algoritmo de Edmonds-Karp es una implementación específica del algoritmo de Ford-Fulkerson. Al igual que Ford-Fulkerson, Edmonds-Karp también es un algoritmo que se ocupa del problema de corte mínimo de flujo máximo.

Ford-Fulkerson a veces se denomina método porque algunas partes de su protocolo no se especifican. Edmonds-Karp, por otro lado, proporciona una especificación completa. Lo que es más importante, especifica que la búsqueda primero en amplitud debe usarse para encontrar las rutas más cortas durante las etapas intermedias del programa.

- Publicado por Yemin Dinitz en 1970 y luego por Jack Edmonds - Richard Karp en 1972, de ahí el nombre.
- Es una implementación del método Ford-Fulkerson.
- La diferencia entre este algoritmo es que el de ford fulkerson usa DFS y este usa BFS
- BFS(breadth-first search) para siempre escoger la trayectoria de aumento con menor cantidad de aristas.
- Tiene una complejidad de $O(VE^2)$, donde V es el número de vértices y E es el número de aristas.
- La complejidad es independiente al valor del flujo máximo, ya que en el otro si dependía del flujo máximo y eso podría ocasionar que sea bastante costoso, ya que el flujo máximo puede tomar valores bastante altos.

El Algoritmo de Edmonds-Karp es una implementación del método de Ford-Fulkerson para calcular el flujo máximo en una red de flujo(i.e. computer network) con complejidad O(V E2).

El algoritmo es idéntico al algoritmo de Ford-Fulkerson, excepto porque el orden para ir buscando los caminos "aumentables" está definido. El camino encontrado debe ser el camino más corto que tiene capacidad disponible. Esto se puede encontrar mediante una búsqueda en anchura, ya que dejamos que los bordes tengan unidad de longitud.

1.3.2 Código Edmonds-Karp

```
#include <iostream>
#include <queue>
#include <cstring>
#define MAX 101
using namespace std;

int graph[MAX][MAX];

bool BFS(int ways[MAX][MAX], int* parent, bool* visited, int n, int
    s, int t) {
  queue<int> q;
  parent[s] = -1;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < n; ++i) {
      if (!visited[i] && ways[u][i] > 0) {
        q.push(i);
        parent[i] = u;
        visited[i] = true;
      }
    }
  }
  return visited[t];
}

int FordFulkerson(int n, int s, int t) {

  int rGraph[MAX][MAX];
  int parent[MAX];
  bool visited[MAX];

  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      rGraph[i][j] = graph[i][j];

      int max_flow = 0;

      while (BFS(rGraph, parent, visited, n, s, t)) {
        // while (DFS(rGraph, parent, visited, n, s, t)){
        memset(visited, false, sizeof(visited));
        int path_flow = 0xffffffff;
        for (int v = t; v != s; v = parent[v]) {
          int u = parent[v];
          path_flow = min(path_flow, rGraph[u][v]);
        }
        for (int v = t; v != s; v = parent[v]) {
          int u = parent[v];
          rGraph[u][v] -= path_flow;
          rGraph[v][u] += path_flow;
        }
        max_flow += path_flow;
      }
```

```
        return max_flow;
}

void addEdge(int ini, int fin, int capacidad){
  graph[ini][fin] = capacidad;
}

int main (){
  int V = 6;

  addEdge(0, 1, 16);
  addEdge(0, 2, 13);
  addEdge(1, 2, 10);
  addEdge(2, 1, 4);
  addEdge(1, 3, 12);
  addEdge(2, 4, 14);
  addEdge(3, 2, 9);
  addEdge(3, 5, 20);
  addEdge(4, 3, 7);
  addEdge(4, 5, 4);

  int s = 0, t = 5;
  cout << "Max Flow: " << FordFulkerson(V, s, t) << endl;
}
```

1.4 Algoritmo Push - Relabel

En la optimización matemática, el algoritmo push-relabel, también se considera un algoritmo para calcular los "flujos máximos" o "maximum flow" en una red de flujo. Este tiene la característica de mantener un pre-flujo (un grafo con el que gradualmente se irá modificando), el cual ira moviendo localmente dicho flujo entre nodos vecinos, utilizando 2 operaciones básicas que hacen único a este algoritmo: operaciones de "empuje" o "push", y operaciones de "reetiquetado" o "relabel"; hasta que al final del algoritmo, tengamos el flujo máximo.

El algoritmo push - relabel, se considera uno de los algoritmos más eficientes para resolver este problema. Puesto que tiene una complejidad de $O(V^2E)$, el cual es asintóticamente más eficiente que el algoritmo Edmonds-Karp ($O(VE^2)$). Y justamente estas variantes del algoritmo push - relabel, logran obtener complejidades mucho menores e igual de interesantes.

1.4.1 Origen

Primero que todo, el concepto de pre-flujo fue diseñado originalmente por Alexander V. Karzanov en 1974 en Siviet Mathematical Dokladi 15. 12 años después, Andrew V. Goldberg y Robert Tarjan implementaron el algoritmo Push - Relabel (noviembre, STOC '86), y luego oficialmente en octubre de 1988 como un artículo en el "Journal of the ACM"

1.4.2 Lógica del algoritmo

La lógica detrás el algoritmo de push-Relabel hace mucha referencia a un típico problema de flujos de fluidos, es decir, consideramos a las aristas como tuberías de agua y a los nodos como las uniones entre estos. Ahora, existen ciertos conceptos que debemos entender, puesto que los usamos frecuentemente para el desarrollo de este algoritmo; entre estos están:

- Altura: es un dato de tipo entero adicional a la información del nodo, que indica a qué altura se ubica de sus nodos adyacentes. Este cumplirá la función de permitir o no que

se haga un "push" a otro nodo. ¿Cómo así?, Se sigue la lógica de que el nodo/fuente que está en el nivel más alto, envía (push) agua a todos los nodos adyacentes que tienen nivel menor. Esto se realizará siempre cuando un nodo tenga exceso de agua

- Relabel o volver a etiquetar: Esto sucede cuando el agua se queda atrapada localmente en un vértice; es decir, los nodos adyacentes a dicho nodo son MAYORES, lo que es imposible hacer un "push". La solución a esto es el mismo reetiquetado, el cual significa aumentarle la altura.

1.4.3 Pseudo Algoritmo

```
Algoritmo Push - Relabel:
    Pre-flow()        // Función que inicializa todos los
                      // vértices y nodos
    Mientras:
        Sea posible hacer un Push()
        Sea posible hacer un Relable()
        Si (Exista un exceso de flujo) then Push() o Relabel()
    Return flujo de retorno     // Una vez que termine el bucle,
                                // tenemos la certeza que todos
                                // los vértices tiene exceso de
                                // flujo 0. Lo que permite hacer
                                // el flujo de retorno, es
                                // decir, retornar el flujo máximo
```

Adicionalmente, se explicará el paso previo antes del bucle que busca el flujo máximo; es decir, la función Pre-flow()

```
Funcion Pre-Flow:
    1) Inicializar la altura y el flujo de cada vértice como 0
    2) Inicializar la altura del nodo de inicio N. N siendo el
       número de vértices en el gráfico
    3) Inicializar el flujo de cada arista como 0
    4) Para los vértices/nodos adyacentes al nodo de inicio,
       el flujo y el exceso de flujo serán igual a la capacidad
       que tienen inicialmente
```

Para un mejor entendimiento, se comparte gráficamente dichos conceptos en una imagen con un ejemplo de problema de flujo máximo.

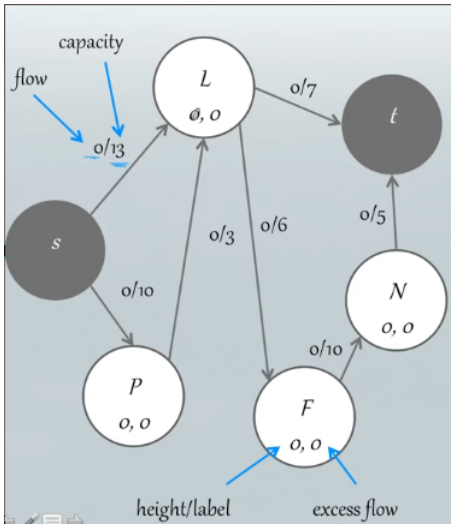

Figure 4: Partes de un grafo utilizando Push-Relabel

1.4.4 Código Push-Relabel

```
#include <bits/stdc++.h>
using namespace std;

class Graph
{
  int V;
  vector<Vertex> ver;
  vector<Edge> edge;

  bool push(int u){
    for (int i = 0; i < edge.size(); i++)
    {
      if (edge[i].u == u)
      {

        if (edge[i].flow == edge[i].capacity)
          continue;

        if (ver[u].h > ver[edge[i].v].h){
          int flow = min(edge[i].capacity - edge[i].flow,
              ver[u].e_flow);
          ver[u].e_flow -= flow;
          ver[edge[i].v].e_flow += flow;
          edge[i].flow += flow;
          updateReverseEdgeFlow(i, flow);
          return true;
        }
      }
    }
    return false;
  }

  void relabel(int u)
  {
    int mh = INT_MAX;
```

```
    for (int i = 0; i < edge.size(); i++){
      if (edge[i].u == u)
      {
        if (edge[i].flow == edge[i].capacity)
          continue;

        if (ver[edge[i].v].h < mh)
        {
          mh = ver[edge[i].v].h;

          ver[u].h = mh + 1;
        }
      }
    }
  }

  void preflow(int s) {
    ver[s].h = ver.size();
    for (int i = 0; i < edge.size(); i++){
      if (edge[i].u == s)
      {
        edge[i].flow = edge[i].capacity;
        ver[edge[i].v].e_flow += edge[i].flow;
        edge.push_back(Edge(-edge[i].flow, 0, edge[i].v, s));
      }
    }
  }

  public:
    int getMaxFlow(int s, int t){
      preflow(s);

      while (overFlowVertex(ver) != -1){
        int u = overFlowVertex(ver);
        if (!push(u))
          relabel(u);
      }

      return ver.back().e_flow;
    }
};
```

1.5 Problema (259.E)

1.5.1 Definición del problema

Este problema se llama G - Grid Card Game, y trata sobre una cuadrilla que contienen cuadrados de dimensiones H x W donde j e i representan la posición de la grilla y una carta con cierto número, como una matriz, se le muestra como, A(i,j). El planteamiento consiste en los siguientes pasos; primero, Takahashi escoje algunas filas H, posiblemente ninguna, y coloca un token rojo en cada carta de las seleccionadas. Segundo, aoki escoge algunas de las columnas W, posiblemente ninguna o todas, y coloca un token azul en cada carta seleccionada. Ahora se ejecuta de esta forma; si hay una carta con valor negativo que tiene token azul y rojo en él, entonces es una falla y su score es -10^{100}, si no, el selecciona todas las cartas y tiene más tokens

a su disposición, el puntaje es la suma de números pertenecientes a las cartas coleccionadas. El objetivo final es hallar el puntaje máximo posible.

1.5.1.1 Problema del corte mínimo

El problema del corte mínimo, es encontrar un corte de mínimo costo en un grafo, imaginemos que tengamos un grafo donde todas las aristas tengan valor 1, el problema se reduciría a encontrar un corte con la menor cantidad de nodos posibles. Así mismo, el problema, ayuda a muchas orientaciones con el problema de flujo máximo, ya que si lográramos identificar un flujo máximo, sería más sencillo obtener el corte mínimo, ya que viendo al grafo residuo el conjunto de nodos que pueden alcanzarse desde S al nodo consultado, inducen un corte mínimo. Por tanto, por definición del grafo residuo, el costo de este corte es igual al valor del flujo máximo y este es un min(s,t) cut.

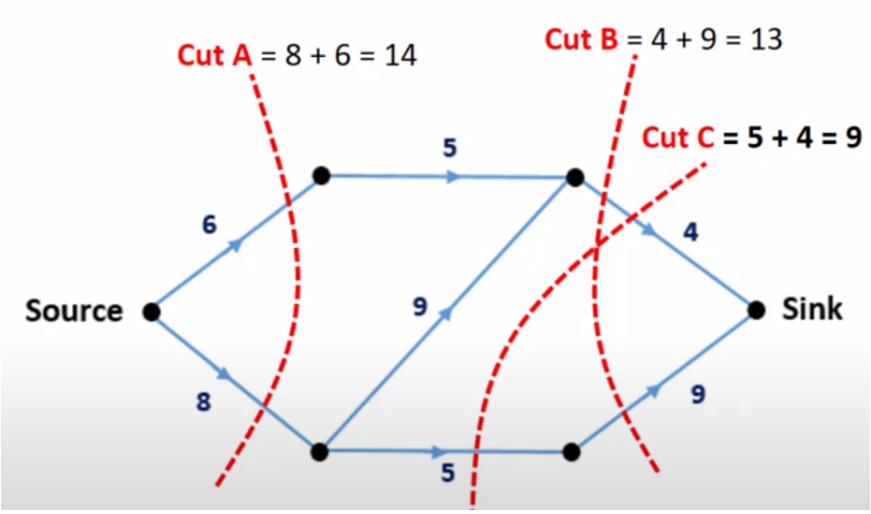

Figure 5: Cortes mínimos en un grafo de flujo máximo

1.5.1.2 Entrada

La entrada se dará del siguiente formato:

$$
\begin{array}{llll}
H & W & & \\
A_{1,1} & A_{1,2} & \ldots & A_{1,W} \\
A_{2,1} & A_{2,2} & \ldots & A_{2,W} \\
\vdots & & & \\
A_{H,1} & A_{H,2} & \ldots & A_{H,W}
\end{array}
$$

Figure 6: Imagen del formato de entrada del problema

Entrada con datos de prueba:

```
2 3
-9 5 1
6 -2 4
```

Figure 7: Entrada real del problema

1.5.1.3 Salida

La salida es el máximo puntaje posible, respecto a la entrada numérica, si Takahashi escoge solo la segunda fila y tercera columna, coleccionaría 4 cartas, 6 + (- 2) + 1 + 4 = 9.

1.5.2 Solución del problema

Si no escogemos columnas con filas, entonces tendremos 0, por lo que la opción de falla total no es posible, entonces no debemos considerarla en el código. Trataremos de reducir el problema a los conceptos de flujo máximo que hemos visto, el programa intentara buscar la mayor cantidad de cartas con enteros positivos y trataremos de minimizar la reducción, empezando desde el estado donde ganamos la suma de todos los enteros positivos, buscaremos la cantidad de reducciones si cambiáramos y optimizaremos ese proceso.

Si una carta en (i,j) con entero positivo no se recogió, entonces es una reducción al óptimo resultado de esa posición, si una carta en una posición cualquiera con valor negativo, ha sido escogida, también hay deducción de puntos. Por ello, en j e i hacemos una suma de los valores negativos, tendremos la deducción total que se deseaba.

Entonces, una solución precisa sería representar el grafo(en grafo de flujo), donde podremos obtener nuestra respuesta como la capacidad del mínimo S-T partiendo del grafo mostrado.

El grafo tiene vértices R1, R2, Rh correspondientes a las filas, y C1,C2 ... correspondientes a la columna, la raíz S y el sumidero T e n un total de (H+ W+2) vértices. Para cada i en H agregamos una arista S al nodo R(i) con la capacidad de la sumatoria hasta ese nodo asimismo con los de las columnas.

- Si A(i,j) es mayor que 0, agrega una arista desde el vértice Ri hasta el vértice Cj con la capacidad A(i,j).
- Si A(i,j) es menor que 0, agrega una arista desde el vértice Ci hasta el vértice Rj con la capacidad máxima(infinito).

1.5.3 Resultados de Ejecuciones

En este caso, optamos por usar el mismo ejemplo usado por la plataforma Atcoder - Sample Input 2. Como se muestra a continuación:

Sample Input 2 Copy

```
15 20
-14 74 -48 38 -51 43 5 37 -39 -29 80 -44 -55 59 17 89 -37 -68 38 -16
14 31 43 -73 49 -7 -65 13 -40 -45 36 88 -54 -43 99 87 -94 57 -22 31
-85 67 -46 23 95 68 55 17 -56 51 -38 64 32 -19 65 -62 76 66 -53 -16
35 -78 -41 35 -51 -85 24 -22 45 -53 82 -30 39 19 -52 -3 -11 -67 -33 71
-75 45 -80 -42 -31 94 59 -58 39 -26 -94 -60 98 -1 21 25 0 -86 37 4
-41 66 -53 -55 55 98 23 33 -3 -27 7 -53 -64 68 -33 -8 -99 -15 50 40
66 53 -65 5 -49 81 45 1 33 19 0 20 -46 -82 14 -15 -13 -65 68 -65
50 -66 63 -71 84 51 -91 45 100 76 -7 -55 45 -72 18 40 -42 73 69 -36
59 -65 -30 89 -10 43 7 72 93 -70 23 86 81 16 25 -63 73 16 34 -62
22 -88 27 -69 82 -54 -92 32 -72 -95 28 -25 28 -55 97 87 91 17 21 -95
62 39 -65 -16 -84 51 62 -44 -60 -70 8 69 -7 74 79 -12 62 -86 6 -60
-72 -6 -79 -28 39 -42 -80 -17 -95 -28 -66 66 36 86 -68 91 -23 70 58 2
-19 -20 77 0 65 -94 -30 76 55 57 -8 59 -43 -6 -15 -83 8 29 16 34
79 40 86 -92 88 -70 -94 -21 50 -3 -42 -35 -79 91 96 -87 -93 -6 46 27
-94 -49 71 37 91 47 97 1 21 32 -100 -4 -78 -47 -36 -84 -61 86 -51 -9
```

Sample Output 2 Copy

```
1743
```

Figure 8: Sample Input 2 - AtCoder

1.5.3.1 Usando Algoritmo Dinic

Con estos puntos a considerar podemos usar el algoritmo de Dinic para encontrar el mínimo corte del grafo (donde habría menos restas), dándonos una complejidad polinomial.

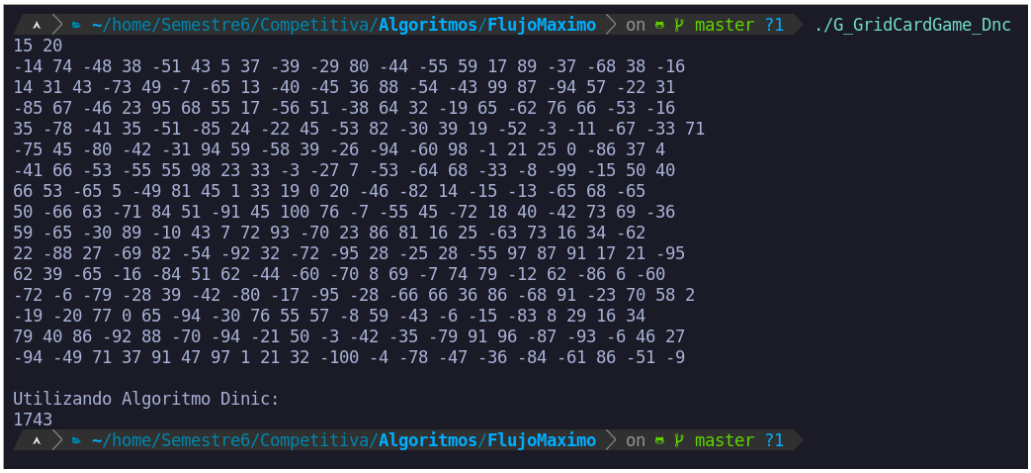

Figure 9: Resultado - Algoritmo Dinic

1.5.3.2 Usando Algoritmo Ford-Fulkerson

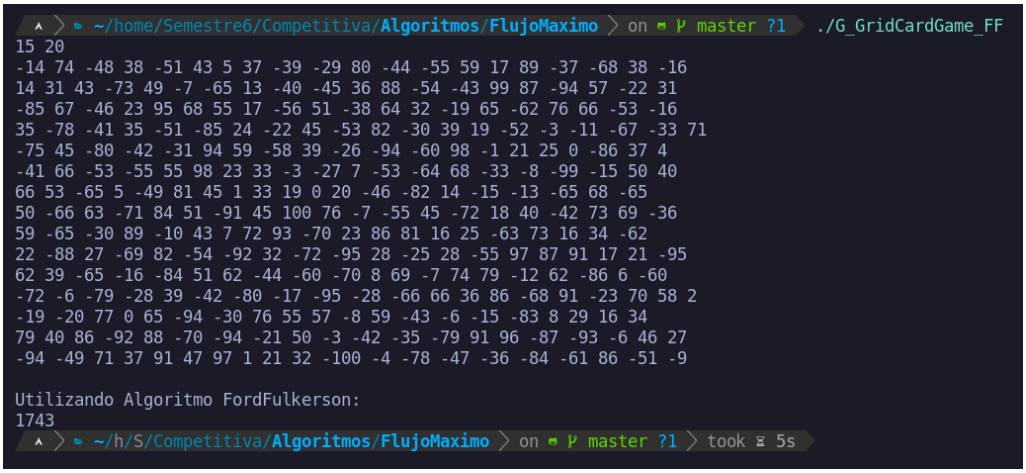

Figure 10: Resultado - Algoritmo Ford-Fulkerson

1.5.3.3 Usando Algoritmo Edmonds-Karp

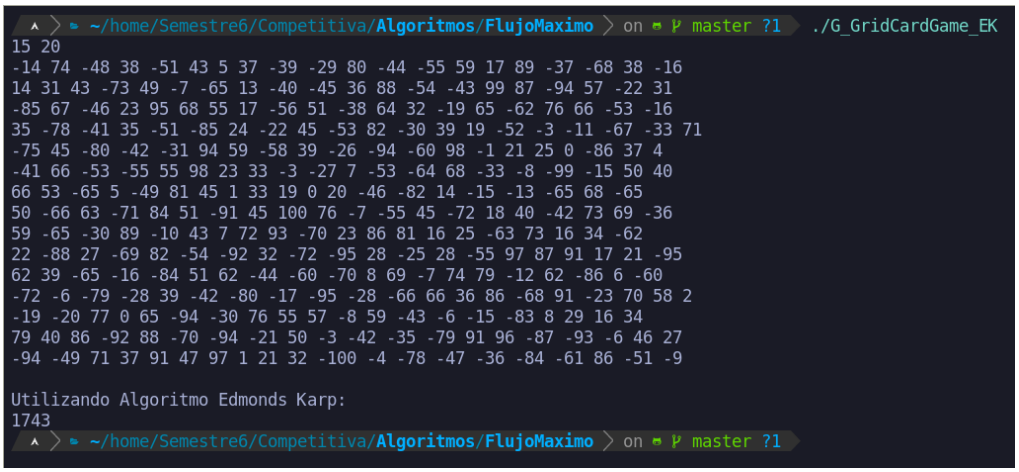

Figure 11: Resultado - Edmonds-Karp

1.5.3.4 Usando Algoritmo Push-Relabel

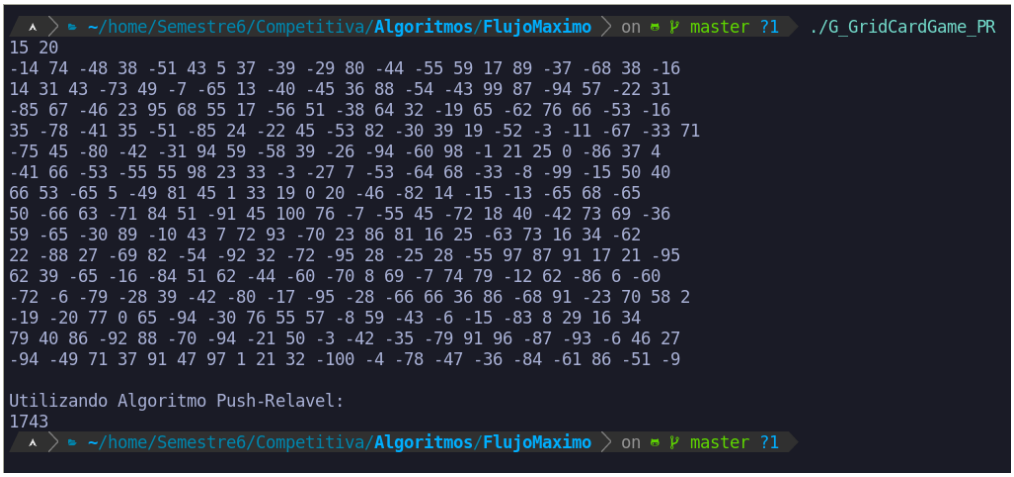

Figure 12: Resultado - Push-Relabel

2 Sweep Line

El algoritmo de Sweep Line, se basa en la idea de usar una línea vertical imaginaria que va hacia una dirección, encontrando ocurrencias de eventos que nosotros definiremos en nuestro problema, este algoritmo se usa para resolver problemas como CLosest Pair, Line Segment intersections, UNion of rectangles, COnvex Hull y más.
Nosotros nos enfocaremos en la unión de rectángulos; sin embargo, debemos entender bien de que se trata el Sweep LIne. En principio es una línea vertical que barre el plano, podemos visualizarlo como una línea vertical en una hoja de papel. Barre en la dirección dada basándose en la ocurrencia de ciertos eventos. Además, usamos términos como la distancia Euclidiana y Distancia de Manhattan, la más conocida es la distancia Euclidiana.

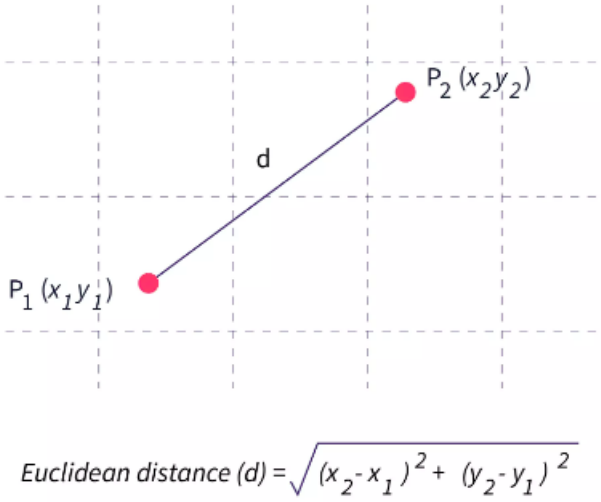

Figure 13: Fórmula de distancia Euclidiana

La distancia de manhattan es la distancia que se recorre mientras moviendo solamente verticalmente u horizontalmente, se da de la forma -|x1-x2|+|y2-y2|

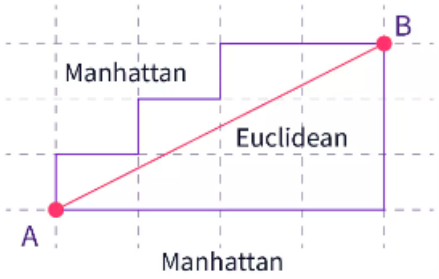

Figure 14: Forma de distancia de Manhattan

Ahora discutiremos el tema específico, UNion de Rectángulos. Este puede ser resuelto con sweep line, la definición del problema es: Dado un conjunto de rectángulos de N ejes alineados (los nodos de los rectángulos son paralelos en el eje x o eje y), requerimos buscar el área de unión de todos estos rectángulos.

Un rectángulo se representa por dos puntos, el menor izquierdo y el mayor derecho. El evento a analizar seria cuando un rectángulo empieza o termina, si encontramos una parte del rectángulo, necesitaremos analizar situaciones precisas para conseguir el área, por ejemplo, si encontramos un rectángulo lo pondremos en un conjunto de rectángulos activos donde encontraremos las intersecciones y las consideraremos sin que se repitan.

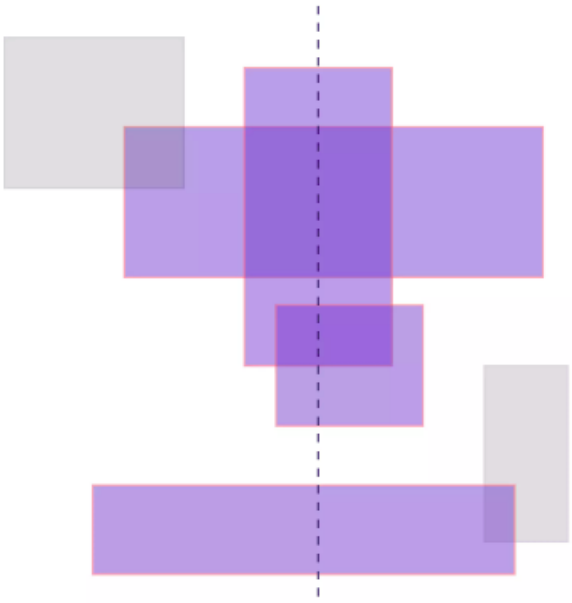

Figure 15: Representación de rectángulos e intersecciones

Como podremos ver, los rectángulos con líneas rojas serán parte del set activo, a diferencia de los otros que por el momento y porque no pertenecen a la línea no son considerados. Para calcular el área de intersección necesitaremos saber el ancho que corta el sweep line, que se multiplicará con la distancia horizontal en cada evento.

Los eventos serían los bordes horizontales de los rectángulos activos (un rectángulo activo sería un rectángulo cortado por la línea de barrido vertical). Al encontrarnos con el borde inferior de un rectángulo activo, incrementaríamos un contador (esta variable contador mantendría la cuenta de rectángulos que se superponen actualmente) y decrementaríamos este contador cuando nos encontremos con el borde horizontal superior del rectángulo activo.

Cuando el contador cambie su valor a cero desde algún valor distinto de cero, habríamos encontrado la longitud del área que se corta en la línea de barrido vertical (el segmento de línea azul sólido como se ha comentado anteriormente) y así es como añadimos el área a nuestra respuesta final.

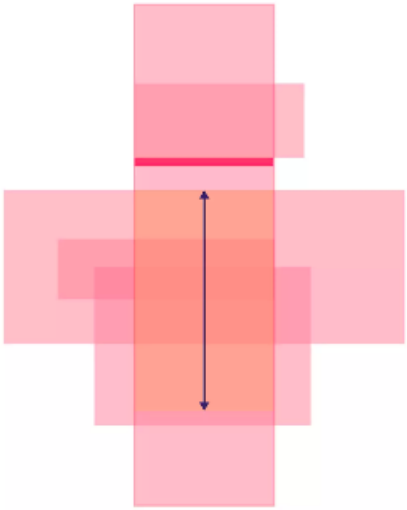

Figure 16: Resultado de sweep line para hallar el área de intersección

2.1 Problema (1083.E)

El problema nos indica que La Nuez Justa (Fair Nut) se atascó y debe resolver una tarea para salir, plantea que tenemos n rectángulos con vértices (0,0), (xi,0), (xi,yi), (0,yi) para cada rectángulo, también te dan un número ahi. Nos pide que escojamos algunos tal que el área de unión, menos la suma de ahi, sea el máximo.

2.1.0.1 Entrada

La primera línea, índica el número de rectángulos, las n líneas siguientes son xi, yi coordenadas de los rectángulos. Se garantiza que no hay rectángulos dentro de otros.

2.1.0.2 Salida

Requiere que imprima el máximo valor que se puede lograr. Imaginémonos la entrada del caso 1.

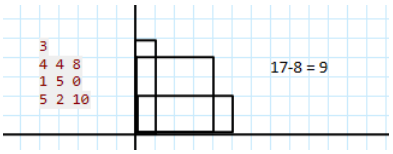

Figure 17: Salida del problema

2.1.0.3 Solucion

En tal caso debemos usar las heurísticas del algoritmo de sweep line, ejecutamos la línea de scanner, para que evalúe la suma de las áreas en unión, asimismo consideramos el elemento a para la iteración, considerando que el cuadro empieza desde (0,0), podemos tomar de abajo hacia arriba, entonces tomamos los n rectángulos que nos den de entrada y sumamos el valor de su área, guardamos en una lista con el valor a sumado, también guardamos su diferencia.
Podemos almacenar los rectángulos escaneados con una lista enlazada, ya que,si al encontrar una mejor diferencia (que la resta sea mayor a la anterior), eliminamos el cuadro con un pop y guardamos el nuevo con un front, una forma sencilla de actualizar los valores.
Esto nos daría un valor optimo cada vez que itera el algoritmo, así que no tendríamos que preocuparnos en intentar sacar una comprobación con otra iteración

2.1.0.4 Codigo

```
struct Rect {
    int x, y;
    long long a;

    bool operator < (const Rect &other) const {
        return x < other.x;
    }
} r[1000000];
```

Figure 18: Struct que define el rectángulo con una sobrecarga para comparar el valor nuevo

```
struct Line {
    long long m, c;

    long long eval(int x) {
        return m * x + c;
    }

    double intersect(Line &l) {
        return (double) (c - l.c) / (l.m - m);
    }
};
```

Figure 19: Struct que define la línea con método de unión más el área total

```
for (int i=0; i<n; i++) {
    while (dq.size() >= 2 && dq.back().eval(r[i].y) <= dq[dq.size()-2].eval(r[i].y))
        dq.pop_back();

    long long val = dq.back().eval(r[i].y) + (long long) r[i].x * r[i].y - r[i].a;
    ret = max(ret, val);
    Line cur = {-r[i].x, val};
    while (dq.size() >= 2 && cur.intersect(dq.front()) >= dq.front().intersect(dq[1]))
        dq.pop_front();
    dq.push_front(cur);
}
```

Figure 20: Loop principal con la ejecución de la solución planteada

3 Algoritmos de String

3.1 Trie

Un trie es una estructura de datos que también se conoce como árbol de búsqueda digital o árbol de prefijos. Puede utilizarse para la recuperación rápida de grandes conjuntos de datos, como la búsqueda de palabras en un diccionario. El término trie se inventó a partir de la frase "Information Retrieval" de Fredkin (1960).

Como un árbol de prefijos, el trie es un árbol ordenado que se utiliza para representar un conjunto de palabras (o strings) sobre un conjunto alfabético finito. Permite que las palabras con prefijos comunes utilicen los mismos datos de prefijo y almacena solo el final de las palabras para indicar palabras diferentes. Cada nodo del trie está etiquetado con un carácter, excepto el nodo raíz, que es un nodo vacío. El número máximo de hijos que puede tener un nodo del trie es igual al número de caracteres del conjunto de alfabetos utilizado para el trie. Los hijos de un nodo están ordenados alfabéticamente y los nodos hermanos deben representar caracteres distintos. El nodo que contiene el último carácter de una palabra se llama nodo hoja. El camino desde la raíz hasta un nodo hoja representa una palabra en el trie.

Consideremos el conjunto C = ’A’, ’C’, ’G’, ’T’, entonces un trie para este conjunto puede tener potencialmente 4 hijos para cada nodo; uno correspondiente a cada letra. La siguiente figura muestra un trie para 6 cadenas diferentes compuestas a partir de C. Las cadenas son: ’AAAA’, ’ACGT’, ’CGA’, ’GTA’, TA’ y ’TTTT’. Observe que "A", "AA", "AAA", "AC", "CG", "ACG", "C", "G", "GT", "T", "TT", "TTT" son prefijos comunes. Además, la altura del trie es la misma que la palabra más larga del trie, que en este caso es 4.

3.1.1 Problema: ADAINDEX - Ada and Indexing

Ada la mariquita tiene muchas cosas que hacer y casi nada de tiempo. Quiere ahorrar tiempo mientras busca algo, por lo que ha decidido crear un motor de búsqueda.

Tiene muchas palabras en su lista de cosas por hacer. Le cuesta un tiempo precioso averiguar si hay una palabra en él, por lo que busca tu ayuda. Obtendrá una lista y algunas consultas. Se le pedirá que averigüe cuántas palabras en la lista tienen otra palabra como prefijo.

Entrada La primera línea contiene N, Q: las palabras numéricas en la lista de cosas que hacer y el número de consultas.

Siguen N líneas, con palabras (de la lista) que consisten en letras minúsculas. La suma de sus longitudes no será mayor que 10 6

Siguen las líneas Q, con palabras (consultas) que consisten en letras minúsculas. La suma de sus longitudes no será mayor que 10 6

3.1.2 Solución

Para solucionar este problema, implemente un trie con una función que cuenta la cantidad de hijos que tiene el nodo que corresponde a la última letra de la palabra sufijo que se consulta e imprimo el resultado.

```
#include<bits/stdc++.h>
#define ll long long
using namespace std;
struct trie
{
  struct trie* arr[26];
  long long count;
};
trie* create()
{
  struct trie *root=NULL;
  root=(trie *)malloc(sizeof(struct trie));
  for(int i=0;i<26;i++)
    root->arr[i]=NULL;
  root->count=0;
  return root;
}
void add(trie *root,string s)
{
  long long len=s.length();
  if(len==0)
    return ;
  long long index=0;
  for(index=0;index<len;index++)
  {
    if(root->arr[s[index]-'a']==NULL)
```

```
    {
      root->arr[s[index]-'a']=create();
    }
    root=root->arr[s[index]-'a'];
    root->count++;
  }
}
long long find(trie *root,string s)
{
  long long len=s.length();
  if(len==0)
    return 0;
  long long index;
  for(index=0;index<len;index++)
  {
    if(root->arr[s[index]-'a']==NULL)
      return 0;
    root=root->arr[s[index]-'a'];
  }
  return root->count;
}
int main()
{
  string str;
  long long N,Q;
  trie *root=create();


  cin>>N>>Q;
  for(long long i=0 ;i<N; ++i){
    cin>>str;
    add(root,str);
  }
  for(long long i=0; i<Q; ++i){
    cin>>str;
    cout<<find(root, str)<<endl;
  }
  return 0;
}
```

3.2 Shift and, shift or

Los algoritmos Shift-And y Shift-Or están relacionados, Shift-Or es solo una mejora en la implementación.

Los algoritmos mantienen el conjunto de todos los prefijos de p que coinciden con un sufijo del texto leído. Usan bitparallelism para actualizar este conjunto para cada nuevo carácter de texto. El conjunto está representado por una máscara de bits D = dm,. . . , d1.

Shift-And se clasifica como un algoritmo basado en prefijos, similar a KMP, pero lo que hace Shift-And es mantener un conjunto de todos los prefijos porque coincidan con un sufijo del texto leído hasta el momento. Tenga en cuenta que es el sufijo del texto leído hasta ahora, no un sufijo de todo el texto.

El conjunto de prefijos se actualiza mediante el paralelismo de bits, es decir, el conjunto de prefijos se mantiene en una máscara de bits denominada D. Se define como: d m . . . d 1, donde

mes la forma canónica de referirse a la longitud del patrón. Por ejemplo, si tenemos un patrón de seis caracteres, y hasta ahora nuestro primer carácter coincide con el texto, entonces Sería 000001.

Con cada carácter leído del texto, Dse actualizará, si el bit d m está activo, entonces tenemos una coincidencia. ¿Es posible actualizar este conjunto en tiempo constante? Sí, utilizando operaciones de bits en paralelo.

Este algoritmo primero construirá una tabla B, manteniendo una máscara de bits b m . . . b 1 para cada carácter en el patrón. Entonces, si estamos buscando la palabra announce, la entrada para el carácter se verá así: 00100110, es decir, la palabra announcetiene una en las posiciones 2, 3 y 6, que están marcadas en la máscara de bits de derecha a izquierda. Entonces, para recapitular: el algoritmo escaneará el patrón que estamos buscando y luego construirá esta tabla, actualizando la máscara de bits a medida que avanza.

El algoritmo Shift OR utiliza técnicas de bit a bit para verificar si el patrón dado está presente en la cadena o no. Es eficiente si la longitud del patrón es menor que el tamaño de la palabra de memoria de la máquina. Nos dan la cuerda, la longitud de la cuerda y el patrón. Nuestro trabajo es devolver el índice inicial del patrón si el patrón existe en la cadena y -1 si no existe.

Algoritmo Los siguientes son los pasos del algoritmo Shift OR para la coincidencia de cadenas:

1. Toma la cadena y el patrón como entrada.
2. Cree una matriz llamada patternmask de tamaño 256 (número total de caracteres ASCII) e inicialícela en 0.
3. Ahora recorra el patrón e inicialice el i-ésimo bit de patternmask[pattern[i]] de derecha a 0
4. Ahora inicialice una variable llamada R que contiene 1.
5. Atraviesa la cuerda de izquierda a derecha.
6. R es igual a R shift patternmask[test[i]].
7. Desplace R hacia la izquierda en 1
8. Si el índice mth (longitud del patrón) en R desde la derecha es igual a 0, entonces la cadena se encuentra en el índice i - m + 1.
9. Si no existe tal m entonces devuelve -1.

3.2.1 Problema Kattis String Matching

La entrada consta de varios casos de prueba. Cada caso de prueba consta de dos líneas, primero un patrón no vacío, luego un texto no vacío. La entrada finaliza al final del archivo.

Para cada caso de prueba, genere una línea que contenga las posiciones de todas las ocurrencias del patrón en el texto, desde la primera hasta la última, separadas por un solo espacio.

3.2.2 Solución

El algoritmo de shift or es una mejora del shift and, como el problema solo es un string matching; es una implementación del shift or con su lógica de bitmaps y en el main un while para mantenerse recibiendo entradas mientras el usuario las siga pasando. y luego por cada pattern una llamada a la función Find Pattern

3.2.3 Algoritmo Shift Or

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int shift_or(string t, string p) {
  int m = p.length();//length of pattern.
  long pattern_mask[256];//creation of array.
  long R = ~1;

  if (m == 0)
    return -1;//If no pattern has been given.
  if (m >63) {
    cout<<"Pattern is too long!";//if pattern is too long
    return -1;
  }
  for (int i = 0; i <=255; ++i){
    pattern_mask[i] = ~0;//initializing pattern mask array.
  }
  for (int i = 0; i < m; ++i){
    pattern_mask[p[i]] &= ~(1L << i);
  }
  for (int i = 0; i < t.length(); ++i) {
    R |= pattern_mask[t[i]];
    R <<= 1;
    if ((R & (1L << m)) == 0)
      return i - m + 1;
  }
  return -1;
}

void findPattern(string t, string p) {
  int position = shift_or(t, p);//invoking shift_or function.
  if (position == -1)
    cout << "\nNo Match\n";
  else
    //cout << "\nPattern found at position: " << position;
    cout << position;
}

int main() {
  string texto, patron;
   while(getline(cin,texto)){
      getline(cin,patron);
      findPattern(texto,patron);
   }
   return 0;



  /*string t;
  t="you speek a bootiful language";
  string p;
  p="peek a boo";
```

```
  findPattern(t, p);*/
}
```

3.3 BMH

El algoritmo BMH, es una solución al problema de string matching, donde dado un patronP(1..n) y un texto T(.1.m) definido en un alfabeto, encuentra las ocurrencias de P en T, tiene aplicaciones en búsquedas web, queries de DB y detectar copia en documentos, la idea principal es tener una 'ventana' que va recorriendo el texto de búsqueda, respeta las reglas del algoritmo, haciendo un pseudo-escaneno.

El algoritmo usa la búsqueda por sufijos, que toma el elemento más a la derecha del substring. leyendo el sufijo y comparándolo con nuestro texto, esto nos permite evitar caracteres del texto que definitivamente no serían parte de la solución, propuesto por Boyeer-Moore, tuvo una modificación por Horspool que le dio el nombre.

El algoritmo nos hace generar dos tablas, que son parte de un preprocesamiento, una de ellas indica cuanto debe desplazarse el substring cuando hubo discordancia en la búsqueda, se da un número de desplazamientos a cada letra, este número se da de la forma: valor = ancho - índice -1, el índice corresponde a la última ocurrencia del substring es la última ocurrencia.

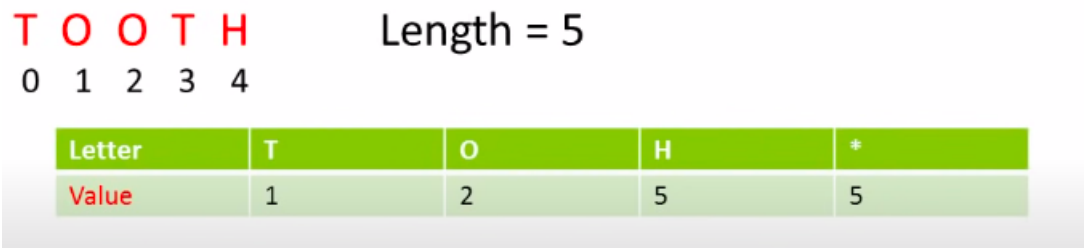

Figure 21: Inicio de la búsqueda

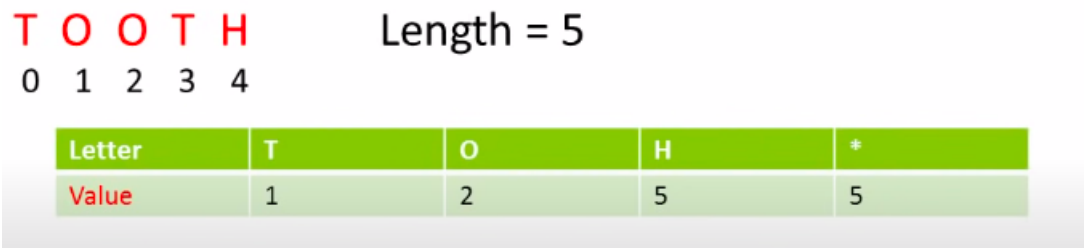

Figure 22: Segundo paso haciendo caso a los saltos

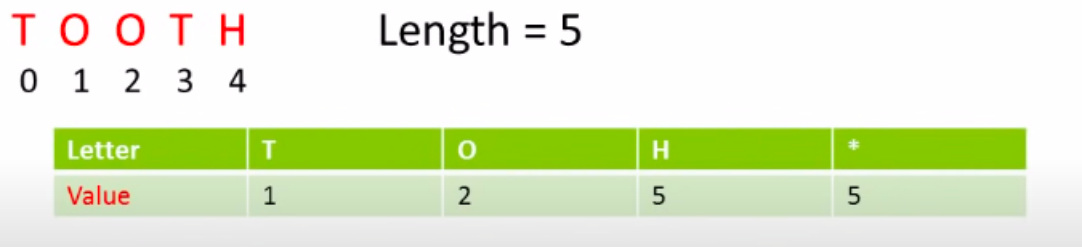

Figure 23: Estado final con el resultado encontrado

Si el primer carácter del texto que se revisa en la actual ventana de búsqueda (esto es, el que se compara con Pm) al revisar el texto es uno que sabemos no aparece en P, eso descarta cualquier ocurrencia del patrón en la ventana de texto, y luego podemos desplazar la ventana m caracteres a la derecha, sin ningún riesgo de pasar por alto una ocurrencia de P en T.

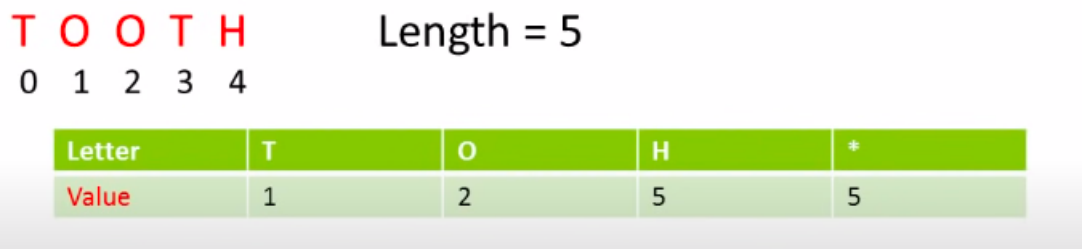

Figure 24: Tabla de índices generada con la formula

Este algoritmo, mejora un recorrido lineal de búsqueda, gracias a los saltos de su tabla. El peor caso de este algoritmo será el ancho del substring por el ancho completo, O(n*m) (este es el caso del algoritmo simple). El caso promedio de este algoritmo es O(M/E), E son las letras del alfabeto, este sumando la existencia de letras, sin embargo, tal E se reemplaza normalmente por n, que será el número de saltos que hará en la tabla de desplazamientos.

3.3.1 Problema Kattis String Matching

La entrada consta de varios casos de prueba. Cada caso de prueba consta de dos líneas, primero un patrón no vacío, luego un texto no vacío. La entrada finaliza al final del archivo.

Para cada caso de prueba, genere una línea que contenga las posiciones de todas las ocurrencias del patrón en el texto, desde la primera hasta la última, separadas por un solo espacio.

3.3.2 Solución

El algoritmo BMH es una búsqueda de atrás hacia adelante, como el problema solo es un string matching; es una implementación sencilla del BMH. Primero su función Bad Heuristic para el preprocesamiento y luego llamamos a esta en la función 'search'. En el main un while para mantenerse recibiendo entradas mientras el usuario las siga pasando.

3.3.3 Algoritmo BMH

```
#include <iostream>
#include <string>
#include <vector>
```

```
using namespace std;
# define NO_OF_CHARS 256

void badCharHeuristic( string str, int size,int badchar[NO_OF_CHARS]){
  int i;

  for (i = 0; i < NO_OF_CHARS; i++)
    badchar[i] = -1;

  for (i = 0; i < size; i++)
    badchar[(int) str[i]] = i;
}

void search( string txt, string pat){
  int m = pat.size();
  int n = txt.size();

  int badchar[NO_OF_CHARS];

  badCharHeuristic(pat, m, badchar);

  int s = 0;
  while(s <= (n - m))
  {
    int j = m - 1;

    while(j >= 0 && pat[j] == txt[s + j])
      j--;

    if (j < 0)
    {
      cout << "pattern occurs at shift = " << s << endl;

      s += (s + m < n)? m-badchar[txt[s + m]] : 1;

    }

    else
      s += max(1, j - badchar[txt[s + j]]);
  }
}

int main()
{
   string texto, patron;
   while(getline(cin,texto)){
      getline(cin,patron);
      search(texto,patron);
   }

}
```

References

[1] GeeksforGeeks. (2022, 30 junio). Push Relabel Algorithm | Set 1 (Introduction and Illustration). Extraido de https://www.geeksforgeeks.org/push-relabel-algorithm-set-1-introduction-and-illustration/

[2] GeeksforGeeks. (2022, 30 junio). Push Relabel Algorithm | Set 2 (Implementation). Extraido de https://www.geeksforgeeks.org/push-relabel-algorithm-set-2-implementation/

[3] LCA, Teoría de grafos, Dávila Méndez Damián (2015, 11 Noviembre). UANL-México.

[4] Algoritmo de flujo máximo push-reetiquetado (s.f.). Recuperado en: https://hmong.es/wiki/Push–relabel$_m aximum_f low_a lgorithm$

[5] Mathur, T. (2022, septiembre 21). Line Sweep Algorithm. Scaler Topics. https://www.scaler.com/topics/data-structures/line-sweep-algorithm/

[6] Us, A. (s/f). Line Sweep Technique. HackerEarth. Recuperado el 28 de noviembre de 2022, de https://www.hackerearth.com/practice/math/geometry/line-sweep-technique/tutorial/

[7] Boyer Moore algorithm for pattern searching. (2012, mayo 26). GeeksforGeeks. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

[8] Std::Boyer$_m oore_h orspool_s earcher.(s/f).Cppreference.com.Recuperadoel28denoviembrede$2022, $dehttps : //en.cppreference.com/w/cpp/utility/functional/boyer_moore_horspool_searcher$

[9] Horspool algorithm for string matching. (s/f). DaniWeb. Recuperado el 28 de noviembre de 2022, de https://www.daniweb.com/programming/software-development/threads/43356/horspool-algorithm-for-string-matching

[10] Raghavendra, A. C. (2021, mayo 16). Shift OR algorithm for String Matching. OpenGenus IQ: Computing Expertise Legacy. https://iq.opengenus.org/shift-or-algorithm-for-string-matching/

[11] Shift-And. (s/f). Helsinki.fi. Recuperado el 28 de noviembre de 2022, de https://www.cs.helsinki.fi/u/tpkarkka/opetus/11s/spa/lecture04.pdf

[12] Shift-AND algorithm for exact pattern matching. (s/f). Tung M Phung's Blog. Recuperado el 28 de noviembre de 2022, de https://tungmphung.com/shift-and-algorithm-for-exact-pattern-matching/

[13] Shift Or algorithm. (s/f). Univ-mlv.fr. Recuperado el 28 de noviembre de 2022, de https://www-igm.univ-mlv.fr/ lecroq/string/node6.html