Estruturas de Dados e Algoritmos II

Licenciatura em Engenharia Informática

3º Trabalho

Card Exchange

André Gonçalves, 58392 | André Zhan, 58762 | Mooshak: g322

Departamento de Informática
Universidade de Évora
Maio 2025

Conteúdo

1 Problema

O problema *Card Exchange* consiste em determinar se é possível organizar uma troca de cartas entre N participantes, de forma que todos recebam uma nova carta de que gostam, sem ficar com a que trouxeram.
Cada participante traz uma única carta e pode indicar quais cartas de outros participantes gostaria de receber. As trocas podem ocorrer em pares, trios ou ciclos maiores, desde que todos os envolvidos estejam satisfeitos com a carta que recebem. O objetivo é verificar se é possível realizar trocas cíclicas de modo que todos os participantes fiquem satisfeitos e ninguém fique de fora da troca.

1.1 Rede de Fluxos e Rede Residual

No método `main`, o programa começa por ler os dados da entrada padrão. Os dados lidos são:

- O número de participantes (N),
- E o número de declarações de interesse (M).

1.1.1 Construção da Rede de Fluxos

A seguir da leitura dos dados, é construída a estrutura da rede de fluxos com `size = 2 * N + 2` nós, onde:

- `S = 2 * N` representa o nó *source* (origem do fluxo),
- `T = 2 * N + 1` representa o nó *sink* (destino do fluxo),
- Os nós de `0` a `N - 1` representam os participantes como **doadores** de carta,
- Os nós de `N` a `2N - 1` representam os mesmos participantes como **receptores** de carta.

Depois disso:

- São adicionadas arestas do *source* para cada participante do lado esquerdo (`S → i`) com capacidade 1.
- São adicionadas arestas de cada participante do lado direito para o *sink* (`N+i → T`) com capacidade 1.
- Para cada declaração de interesse (a, b), é criada uma aresta de `a` (do lado esquerdo) para `N + b` (do lado direito), com capacidade 1.

1.1.2 Construção da Rede Residual

Durante a execução do algoritmo de Edmonds-Karp, é construída a rede residual, que representa a rede de fluxo atualizada com as capacidades restantes e possíveis reversões de fluxo.
Para cada aresta (u → v) na rede original com capacidade `c` e fluxo atual `f`, a rede residual contém:

- Uma aresta direta (u → v) com capacidade residual `c - f`, permitindo enviar mais fluxo se ainda houver capacidade disponível.
- Uma aresta reversa (v → u) com capacidade `f`, permitindo reverter parte do fluxo já enviado, caso seja necessário reajustar decisões anteriores.

Esta estrutura é fundamental para identificar caminhos aumentantes (isto é, caminhos com capacidade residual positiva do nó de origem `S` até ao nó de destino `T`). O algoritmo itera até que não existam mais caminhos aumentantes na rede residual, garantindo que se atingiu o fluxo máximo.

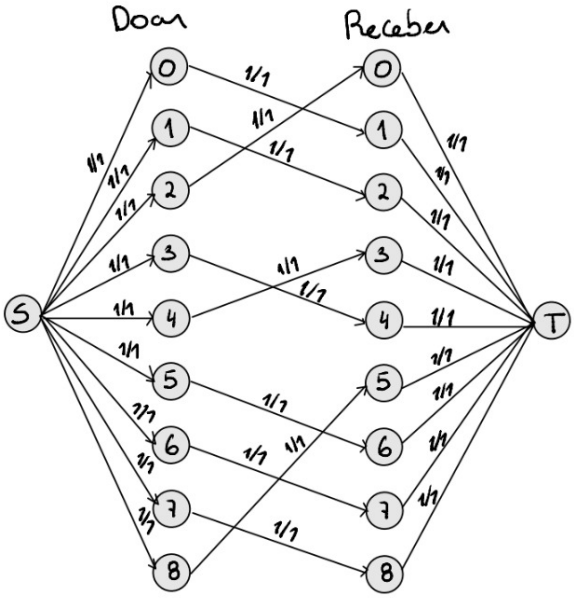
1.1.3 Rede de Fluxos Exemplo 1

Sample Input 1

```
9 9
0 1
1 2
2 0
3 4
4 3
5 6
6 7
7 8
8 5
```

Sample Output 1

YES



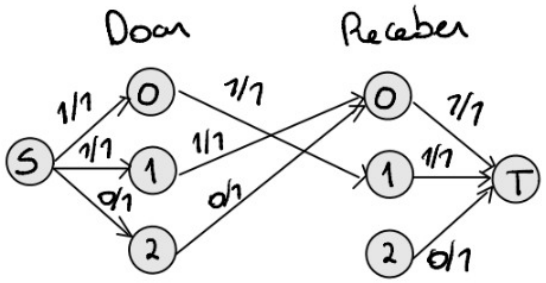
1.1.4 Rede de Fluxos Exemplo 2

Sample Input 2

```
3 3
0 1
1 0
2 0
```

Sample Output 2

YES



1.2 Solução para o Problema

A estratégia adotada foi transformar o problema num problema clássico de fluxo máximo em grafo bipartido, mais especificamente um problema de emparelhamento perfeito.
A ideia principal é:

- Garantir que cada participante consiga **dar** a sua carta a alguém interessado.
- E que cada participante **receba** uma carta de alguém cujo dono esteja disposto a trocá-la.

Isso é modelado como uma rede de fluxo onde cada participante está representado dos dois lados da troca: como *doador* e *receptor*.
Utiliza-se o algoritmo de **Edmonds-Karp**, que usa ***BFS*** para encontrar caminhos aumentantes. O algoritmo:

- Repetidamente encontra caminhos com capacidade residual positiva entre a origem (*source*) e o destino (*sink*).
- Aumenta o fluxo ao longo desses caminhos até não haver mais caminhos disponíveis.

Se o **fluxo máximo final** for igual a N, significa que todos os participantes conseguiram realizar uma troca satisfatória — logo, a resposta é ‘‘`YES`’’. Caso contrário, a resposta é ‘‘`NO`’’.

1.3 Algoritmo de Edmonds-Karp Usado

O algoritmo de Edmonds-Karp foi implementado com base numa estrutura de rede de fluxo orientada e com capacidades unitárias. A seguir apresenta-se o pseudocódigo comentado de cada uma das partes principais do algoritmo.

1.3.1 Base do Algoritmo

```
    // Algoritmo de Edmonds-Karp: encontra o fluxo máximo de S para T
public int edmondsKarp() {
    // Constrói a rede residual inicial com base na rede de fluxo original
    FlowNetwork r = buildResidualNetwork();

    int flowValue = 0; // Valor total do fluxo
    int[] p = new int[nodes]; // Vetor de predecessores para reconstruir o caminho
    int increment;

    // Enquanto existir um caminho aumentante na rede residual
    while ((increment = r.findPath(p)) > 0) {
        // Aumenta o fluxo ao longo do caminho encontrado
        incrementFlow(p, increment, r);

        // Atualiza o valor total do fluxo
        flowValue += increment;
    }

    return flowValue; // Retorna o fluxo máximo encontrado
}
```

1.3.2 Construção da Rede Residual

```
    private FlowNetwork buildResidualNetwork() {
    FlowNetwork r = new FlowNetwork(nodes, source, sink);

    for (int u = 0; u < nodes; u++) {
        for (Edge e : adjacents[u]) {
            int v = e.destination();
            int cap = e.capacity() - e.flow();

            // Aresta direta com capacidade residual (fluxo que ainda pode passar)
            if (cap > 0) r.addEdge(u, v, cap);
```

```
            // Aresta reversa com a quantidade de fluxo que pode ser revertido
            if (e.flow() > 0) r.addEdge(v, u, e.flow());
        }
    }

    return r;
}
```

1.3.3 Caminho Aumentante com *BFS* (*Breadth-First Search*)

```
    private int findPath(int[] p) {
    int[] cf = new int[nodes]; // Capacidade residual até cada nó
    Queue<Integer> q = new LinkedList<>();

    // Inicializa predecessores como "nenhum"
    for (int u = 0; u < nodes; ++u) p[u] = NONE;

    cf[source] = INFINITY; // Capacidade infinita na origem
    q.add(source); // Começa a BFS a partir da origem

    while (!q.isEmpty()) {
        int u = q.remove();

        if (u == sink) break; // Chegou ao destino

        for (Edge e : adjacents[u]) {
            int v = e.destination();

            // Se há capacidade e o nó ainda não foi visitado
            if (e.capacity() > 0 && cf[v] == 0) {
                cf[v] = Math.min(cf[u], e.capacity()); // Atualiza capacidade mínima
                p[v] = u; // Guarda o predecessor
                q.add(v); // Continua a busca
            }
        }
    }

    return cf[sink]; // Capacidade do caminho até o sink
```

```
}
```

1.3.4 Incremento do Fluxo

```
    private void incrementFlow(int[] p, int increment, FlowNetwork r) {
    int v = sink;
    int u = p[v];

    while (u != NONE) {
        boolean uv = false;

        // Tenta encontrar a aresta direta u → v
        for (Edge e : adjacents[u]) {
            if (e.destination() == v) {
                e.flow(e.flow() + increment); // Aumenta o fluxo
                r.updateResidualCapacity(u, v, e.capacity(), e.flow());
                uv = true;
                break;
            }
        }

        // Caso contrário, ajusta a aresta reversa v → u
        if (!uv) {
            for (Edge e : adjacents[v]) {
                if (e.destination() == u) {
                    e.flow(e.flow() - increment); // Diminui o fluxo (reverte)
                    r.updateResidualCapacity(v, u, e.capacity(), e.flow());
                    break;
                }
            }
        }

        v = u;
        u = p[v]; // Move-se para o próximo nó no caminho
    }
}
```

2 Análise de Complexidade

2.1 Complexidade Temporal

A solução utiliza o algoritmo de Edmonds-Karp para calcular o fluxo máximo. Este algoritmo usa **Busca em Largura (*BFS*)** para encontrar caminhos aumentantes.

- Cada chamada ao *BFS* tem complexidade $O(E)$, onde E é o número de arestas na rede.
- O número máximo de aumentos de fluxo é $O(V \cdot E)$, onde V é o número de vértices.
- Logo, a complexidade total do algoritmo é:

$$O(V \cdot E^2)$$

No contexto do problema:

- $V = 2N + 2$ (nós duplicados + *source* + *sink*)
- $E = O(N + M)$, considerando as arestas do *source*, do *sink* e as M declarações de interesse

Portanto, no pior caso, a complexidade temporal é:

$$O(N \cdot (N + M)^2)$$

2.2 Complexidade Espacial

A estrutura principal usada é uma lista de adjacências que armazena as arestas da rede de fluxo.

- A lista de adjacência tem $O(V)$ listas internas.
- Cada lista armazena até $O(E)$ arestas no total.

Além disso, são usados:

- Vetores auxiliares de tamanho $O(V)$ (como o vetor de capacidade residual).
- Uma cópia da rede residual é criada em cada iteração do algoritmo.

Logo, a complexidade espacial total é:

$$O(V + E) = O(N + M)$$

3 Referências

Referências

[1] Slides das aulas teóricas.