Estruturas de Dados e Algoritmos II

Licenciatura em Engenharia Informática

Trabalho Prático 3

Card Exchange

UNIVERSIDADE DE ÉVORA

Grupo: g318
Miguel Pombeiro, 57829 | Miguel Rocha, 58501

Departamento de Informática
Universidade de Évora
Maio 2025

Índice

1 Introdução

O problema proposto em "Card Exchange" descreve uma situação em que se pretende verificar se todos os participantes de um evento de cartas conseguem fazer uma troca com outro participante. Cada participante traz uma carta de qualquer jogo e pode trocar com outro participante, desde que ambos acordem com a troca. Para melhor analisar este problema, é possível modelar as preferências de cartas de cada participante num grafo bipartido e, de seguida, adaptar esse grafo a uma rede de fluxos. Para tal, são necessárias as ligações desde a fonte para o lado esquerdo do grafo e do lado direito para o dreno. Assim, o objetivo final é verificar se o fluxo máximo da rede é igual ao número total de participantes.

2 Algoritmo

2.1 *Input*

Os *inputs* disponíveis para cada teste são:

- O número de participantes do evento (**N**).
- O número de declarações de interesse do evento (**M**).
- Os participantes envolvidos em cada declaração de interesse.

2.2 Descrição da Rede de Fluxos

Para resolver este problema, foi necessário modelar os dados do *input* numa rede de fluxos. Nesta rede, cada participante do evento é representado duas vezes, de acordo com os dois papéis que desempenha: dador da sua carta e recetor da carta de outro participante. Desta forma, é possível construir a rede de fluxos a partir de um grafo bipartido em que os nós do lado esquerdo (c) representam os dadores de cartas e o lado direito (p) os recetores de cartas.

As ligações entre o lado esquerdo e o lado direito representam a relação "gostar", em que o participante do lado esquerdo tem de dar a sua carta ao participante do lado direito para, possivelmente, receber a carta de que gosta. Devem, por isso, ter capacidade 1, significando que o participante do lado esquerdo pode, potencialmente, ficar com a carta do participante do lado direito. Para construir a rede de fluxos é, ainda, necessário representar os nós fonte (*source*) e dreno (*sink*). Assim, para representar que cada participante apenas pode dar uma carta, são feitas ligações entre a fonte e cada um dos nós do lado esquerdo, através de arcos com capacidade 1. Para representar que cada participante apenas pode receber uma carta, são feitas ligações entre cada um dos nós do lado direito e o dreno, através de arcos com capacidade 1.

2.3 Rede de Fluxos Construída

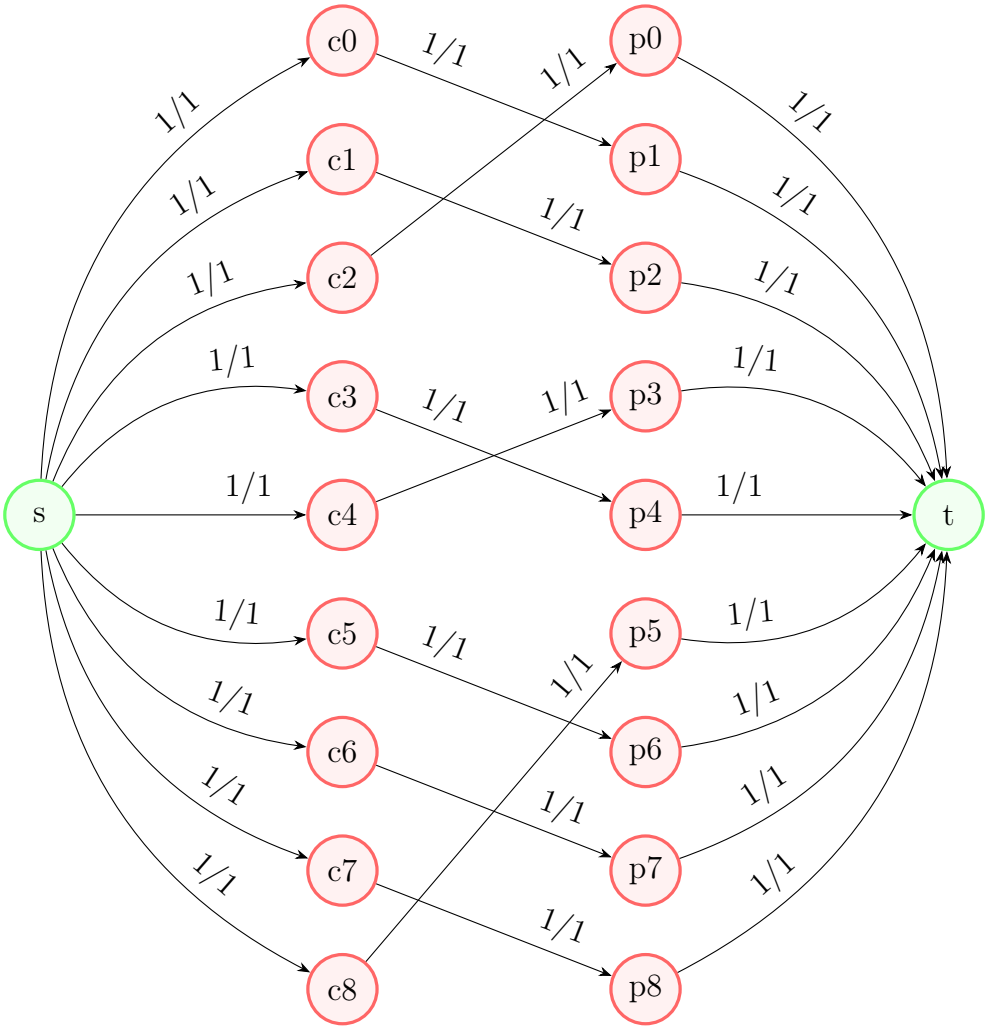

Figura 1: Rede de fluxos construída a partir do "Sample Input 1" do enunciado.

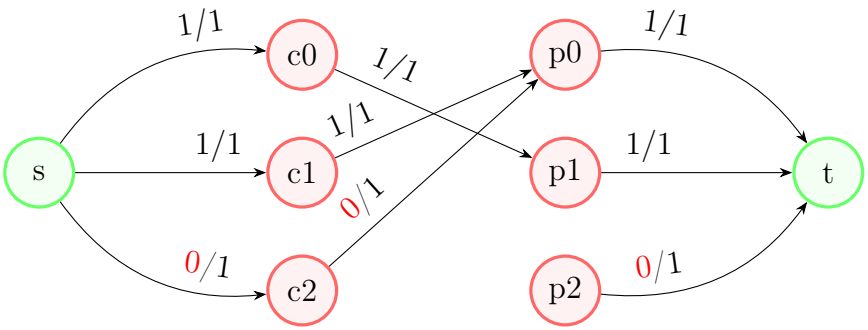

Figura 2: Rede de fluxos construída a partir do "Sample Input 2" do enunciado.

2.4 Descrição do Algoritmo

O primeiro passo para resolver o problema é a modelação das preferências dos participantes do evento numa rede de fluxos, como foi descrito na Subseção 2.2.

Para encontrar o fluxo máximo nesta rede de fluxos, foi utilizado o algoritmo de Edmonds-Karp (Algoritmo 1), que é uma implementação do método de Ford-Fulkerson, com recurso a *Breath-First Search* (BFS).

O algoritmo de Edmonds-Karp começa por criar uma rede residual, sem qualquer fluxo, a partir da rede de fluxos original (Algoritmo 2). Para tal, é inicializada outra rede de fluxos na qual são copiadas todas as arestas da rede inicial, com capacidade igual, e arestas com orientação oposta às originais, com capacidade 0. As arestas comuns à rede original representam a capacidade ainda não utilizada pelo fluxo e as arestas com orientação oposta representam o fluxo que está a ser utilizado (e que pode, eventualmente, ser cancelado).

De seguida, o algoritmo entra na sua fase cíclica, que é repetida enquanto for possível encontrar, na rede residual, um caminho simples que permita aumentar o fluxo que passa da fonte para o dreno, incrementando, deste modo, o fluxo total na rede original. Assim, de forma a conseguir guardar este caminho, é necessário um vetor onde serão guardados os nós predecessores de cada nó, para poder ser utilizado posteriormente.

Como já foi referido, o algoritmo de Edmonds-Karp, utiliza o algoritmo de BFS para encontrar estes caminhos que permitem aumentar o fluxo (Algoritmo 3). O algoritmo de BFS é um algoritmo de pesquisa em largura em grafos que permite explorar todos os vértices nível a nível, ou seja, explora todos os vértices que se encontram a uma mesma distância de um vértice inicial, antes de explorar vértices que se encontram a distâncias superiores. Desta forma, o algoritmo de pesquisa em largura vai sempre encontrar o caminho mais curto da fonte para o dreno, pelo que os caminhos encontrados vão ser cada vez maiores, podendo, potencialmente, incluir todos os vértices da rede.

Para conseguir encontrar a encontrar os caminhos que permitem aumentar o fluxo da rede, é necessário fazer algumas alterações nas estruturas de dados utilizadas no algoritmo base de BFS:

- Utilizar um vetor de predecessores, para manter um registo do caminho encontrado.
- Adaptar o vetor que permite marcar os vértices já visitados, para registar o fluxo que é possível aumentar até eles.

Esta pesquisa em largura vai percorrer os vários caminhos possíveis da fonte até ao dreno da rede residual, calculando as suas capacidades residuais mínimas. Para tal, apenas podem ser utilizadas as arestas que ainda têm uma capacidade positiva. Esta adaptação do BFS deve terminar quando é explorado o vértice que corresponde ao dreno, o que

indica que chegou ao fim da rede residual com o vetor de predecessores preenchido com o caminho, bem como o valor de fluxo que é possível aumentar na rede.

Caso tenha sido encontrado um caminho que permita aumentar o fluxo na rede, é necessário atualizar o fluxo da rede de fluxos original, bem como a capacidade das arestas da rede residual (Algoritmo 4). Para isso, é utilizado o vetor de predecessores obtido anteriormente, percorrendo o caminho do fim para o início, ou seja, do dreno para a fonte. Estas atualizações são feitas conforme as arestas deste caminho estão ou não na rede de fluxos original: caso estejam, o fluxo da rede de fluxos é incrementado do valor de fluxo a aumentar, e caso não estejam (o caminho usou uma aresta de cancelamento), o fluxo da rede de fluxos é decrementado. São, ainda, atualizadas as capacidades das arestas da rede residual de modo a refletir estas alterações.

No final, quando já não for possível encontrar mais caminhos que permitam aumentar o fluxo na rede, o algoritmo de Edmonds-Karp deve ter conseguido maximizar o fluxo que é possível ter na rede.

Deste modo, tendo modelado todas a restrições do problema numa rede de fluxos, conclui-se que se o fluxo máximo encontrado for igual ao número de participantes no evento, então existe uma forma viável de todos os participantes conseguirem trocar a sua carta por uma desejada.

2.5 Pseudocódigo

Para a resolução do problema proposto, foi necessário implementar vários algoritmos:

- O algoritmo principal que resolve o problema, Edmonds-Karp (Algoritmo 1). É responsável por criar uma rede residual a partir de uma rede de fluxos, e tem como objetivo encontrar o fluxo máximo dessa rede de fluxos.
- O algoritmo responsável por criar uma rede residual a partir de uma rede de fluxos, RESIDUAL-NET (Algoritmo 2) e faz parte do algoritmo de Edmonds-Karp.
- O algoritmo de pesquisa em largura, BFS-FIND-PATH (Algoritmo 3). Faz parte do algoritmo de Edmonds-Karp, tem como objetivo encontrar um caminho válido no grafo da rede residual e retornar um possível aumento para o fluxo dessa rede, nesse caminho.
- O algoritmo de aumento do fluxo numa rede residual, INCREMENT-FLOW (Algoritmo 4), que também faz parte do algoritmo de Edmonds-Karp. Tem como objetivo incrementar o fluxo de um determinado caminho, em ambas as redes. As arestas do caminho na rede residual são depois atualizadas com recurso a outro algoritmo que foi omitido.

No entanto, também foi usado um algoritmo, cujo objetivo é apenas comparar o resultado do Edmonds-Karp (Algoritmo 1) e verificar se o fluxo máximo retornado é igual ao número de participantes do evento, imprimindo na consola o resultado pedido ("YES" ou "NO"). Devido à sua baixa complexidade, está omitido nesta secção do relatório.

Algoritmo 1 EDMONDS-KARP(G, s, t)

Require: G, um grafo que representa uma rede de fluxos; *nNodes*, número de vértices de G; *s*, vértice que representa a fonte; *t*, vértice que representa o dreno
Ensure: O fluxo inicial de todos os vértices de G é 0

```
1:  flowValue ← 0
2:  let p[0...nNodes]
3:  Gf ← RESIDUAL-NET(G, s, t)
4:
5:  while (cf ← BFS-FIND-PATH(Gf, p, s, t)) > 0 do
6:      INCREMENT-FLOW(G, Gf, p, cf, t)
7:      flowValue = flowValue + cf
8:  end while
9:
10: return flowValue
```

Algoritmo 2 Construção uma rede residual a partir de uma rede de fluxos, RESIDUAL-NET(G, s, t)

Require: G, um grafo com lista de adjacências que representa uma rede de fluxos; *nNodes*, número de vértices de G; *s*, vértice que representa a fonte; *t*, vértice que representa o dreno

Ensure: Ainda não há fluxos a passar na rede de fluxos G

```
1: let Gf be a new flow network
2:
3: for i ← 0 to nNodes do
4:     for each Edge e in G.adj[i] do
5:         NEW-EDGE (Gf, i, e.destination, e.capacity)      ▷ Arco com direção igual
6:         NEW-EDGE (Gf, e.destination, i, 0)               ▷ Arco com direção oposta
7:     end for
8: end for
9:
10: return Gf
```

Algoritmo 3 BFS-FIND-PATH(Gf, p, s, t) - Percurso em largura

Require: Gf, uma rede residual com a sua lista de adjacências; p, um *array* com os predecessores de cada vértice; *nNodes*, número de vértices de G; *s*, vértice que representa a fonte; *t*, vértice que representa o dreno

```
1: let cf[0...nNodes] be a new array
2: for  i ← 0 to nNodes do
3:     p[i] ← NIL
4: end for
5: cf[s] ← +∞
6:
7: let Q be a new EMPTY queue
8: ENQUEUE(Q, s)
9:
10: while Q ! = EMPTY do
11:     u ← DEQUEUE(Q)
12:     if u == t then
13:         Break
14:     end if
15:     for each Edge e in Gf.adj[u] do
16:         v ← e.destination
17:         if e.capacity > 0 and cf[v] == 0 then
18:             cf[v] ← min(cf[u], e.capacity)
19:             p[v] ← u
20:             ENQUEUE(Q, v)
21:         end if
22:     end for
23: end while
24: return cf[t] and p
```

Algoritmo 4 Incrementar o fluxo de uma rede residual, INCREMENT-FLOW(G, Gf, p, i, t)

Require: G, um grafo com lista de adjacências que representa uma rede de fluxos; Gf, uma rede residual com a sua lista de adjacências; p, um *array* com os predecessores de cada vértice; i, o valor do fluxo a incrementar; t, vértice que representa o dreno

```
1: v ← t
2: u ← p[v]
3:
4: while u ! = NIL do
5:     uv ← false
6:     for each Edge e in G.adj[u] do
7:         if e.destination == v then
8:             e.flow ← e.flow + i
9:             Gf.UpdateResidualCapacity(u, v, e.capacity, e.flow)
10:            uv ← true
11:            Break
12:        end if
13:    end for
14:
15:    if !uv then
16:        for each Edge e in G.adj[v] do
17:            if e.destination == u then
18:                e.flow ← e.flow − i
19:                Gf.UpdateResidualCapacity(v, u, e.capacity, e.flow)
20:                Break
21:            end if
22:        end for
23:    end if
24:    v ← u
25:    u ← p[v]
26: end while
27: return distance
```

3 Complexidades

Para o cálculo das complexidades, de modo a facilitar a leitura em alguns casos, foram utilizadas variáveis que não estão nos parâmetros do enunciado.
Desta forma, em algoritmos que usam a rede de fluxos:

- Variável V, corresponde ao número de vértices da rede de fluxos. Esta variável, nos parâmetros do enunciado, é igual a $2N + 2$. Este valor deve-se à rede de fluxos ser feita a partir de um grafo bipartido N para N, e da adição dos nós da fonte e do dreno, totalizando $2N + 2$.

- Variável E, corresponde ao número de arestas da rede de fluxos. Esta variável, nos parâmetros do enunciado, é igual a $2N + M$. Este valor resulta da rede de fluxos conter as ligações dos vértices, que representam participantes, à fonte e ao dreno, bem como as ligações correspondentes às declarações de interesse.

Em algoritmos que usam a rede residual:

- Variável V, corresponde ao número de vértices da rede residual. Esta variável, nos parâmetros do enunciado, é igual a $2N + 2$. Como a rede residual tem o mesmo número de vértices da rede de fluxos, este valor mantém-se.

- Variável E, corresponde ao número de arestas da rede residual (arcos diretos e inversos). Esta variável, nos parâmetros do enunciado, é igual a $2(2N + M) = 4N + 2M$. Este valor é devido ao facto de a rede residual conter, também, arestas com a direção oposta à rede de fluxos original.

3.1 Temporal

Complexidade da criação de uma rede residual (Algoritmo 2):

- Linha 1: A criação de uma rede de fluxos tem complexidade $\Theta(V)$, sendo V o número de vértices da rede de fluxos G.

- O ciclo das linhas **3 - 8** é executado V vezes (variável *nNodes*).

  - O ciclo das linhas **4 - 7** é executado E vezes, sendo E o número de arestas da rede de fluxos.

    - O custo de adicionar uma nova aresta à rede residual é constante.

Logo, a complexidade final deste algoritmo é $\Theta(V) + \Theta(V + E) = \Theta(V + E)$, que é equivalente a $\Theta((2N + 2) + (2N + M) = \Theta(N + M)$, usando os parâmetros do enunciado.

Complexidade do BFS-FIND-PATH (Algoritmo 3):
Considera-se que as operações de criação de uma *queue* e de um vetor vazios, bem como ENQUEUE e DEQUEUE, têm custo $\Theta(1)$. Todas as outras para além das mencionadas a seguir terão, também, custo constante.

- O ciclo das linhas **2 - 4** tem custo $\Theta(V)$, sendo V o número de vértices da rede residual Gf (variável *nNodes*).
- O ciclo das linhas **10 - 23** tem custo $\mathcal{O}(V + E)$, no seu pior caso. Onde E, é o número de arestas da rede residual Gf.
  - $\mathcal{O}(V)$ na linha **11**.
  - O ciclo das linhas **15 - 22** é $\mathcal{O}(E)$ (análise agregada dos dois ciclos).

Assim a complexidade deste algoritmo é $\mathcal{O}(V+E)+\Theta(V) = \mathcal{O}(V+E)$, que é equivalente a $\mathcal{O}((2N + 2) + (4N + 2M))) = \mathcal{O}(N + M)$, usando os parâmetros do enunciado.

Complexidade do incremento de fluxo (Algoritmo 4):
Todas as operações não mencionadas têm custo temporal constante.

- O ciclo das linhas **4 - 26** tem custo $\mathcal{O}(V + E)$, sendo V o número de vértices e E o número de arestas da rede de fluxos.
  - $\mathcal{O}(E + V)$ do ciclo das linhas **6 - 13** (análise agregada dos dois ciclos).
  - $\mathcal{O}(E + V)$ do ciclo das linhas **16 - 22** (análise agregada dos dois ciclos).
  - $\mathcal{O}(V)$ das linhas **5, 24, 25**.

Assim, no pior caso, a complexidade final deste algoritmo é $\mathcal{O}(V+E)$, que é equivalente a $\mathcal{O}((2N + 2) + (2N + M)) = O(N + M)$, usando os parâmetros do enunciado.

Complexidade do Edmonds-Karp (Algoritmo 1):
Todas as operações não mencionadas têm custo temporal constante.

- Linha **3**: A construção de uma rede residual tem complexidade $\Theta(V + E)$ sendo V o número de vértices e E o número de arestas da rede de fluxos G.
- O ciclo das linhas **5 - 8** é executado, no máximo, $\mathcal{O}(VE)$ vezes, sendo V o número de vértices e E o número de arestas da rede residual Gf.
  - Linha **5**: O percurso em largura (BFS) na rede residual tem custo $\mathcal{O}(V + E)$.
  - Linha **6**: O custo do incremento do fluxo em ambas as redes é $\mathcal{O}(V + E)$.

Assim, a complexidade temporal do algoritmo de Edmonds-Karp é:

$$\Theta(V+E) + \mathcal{O}(VE(V+E)) = \mathcal{O}(VE^2)$$

Substituindo pelos parâmetros do problema obtemos $\mathcal{O}((2N+2)(4N+2M)(4N+2M)$ $= \mathcal{O}(N(N+M)(N+M))$. Assim, esta será a complexidade deste algoritmo, e também do problema.

3.2 Espacial

Para fazer a análise da complexidade espacial, é necessário ter em conta todo o espaço de memória, que varia com o *input*, ao longo de todo o programa.

- Quando é criada uma rede de fluxos, é necessário a criação de um *array* de listas, com *2N + 2* posições, para a lista de adjacências. Aqui estão representadas as arestas da rede de fluxos, que no total são, $2N+M$. Portanto, $\Theta((2N+2)+(2N+M)$ $= \Theta(N+M)$.

- Quando é criada uma rede residual, é necessário a criação de um *array* de listas, com *2N + 2* posições, para a lista de adjacências. Aqui estão representadas os arcos com as duas orientações, totalizando $2(2N+M)$. Portanto, $\Theta((2N+2)+(4N+2M)$ $= \Theta(N+M)$.

- No algoritmo de pesquisa em largura, é utilizado um vetor de dimensão $2N+2$ e uma *queue* (*LinkedList*) que terá, no máximo, $(2N+2)-1$ elementos. Portanto, no pior caso, $\mathcal{O}((2N+2)+(2N+1) = \mathcal{O}(N)$.

- No algoritmo de Edmonds-Karp, para além da rede residual, é utilizado um vetor de dimensão *2N+2*, para guardar um caminho de predecessores. Portanto, $\Theta(N)$.

As restantes variáveis utilizadas são números inteiros e não variam com *input*, tendo complexidade espacial constante. Assim sendo, a complexidade espacial deste programa é $\Theta(N+M)$.

4 Comentários e fontes consultadas

Este programa cumpriu os requisitos propostos, de modo a verificar se todos os participantes de um evento de cartas, conseguem realizar pelo menos uma troca para obter uma carta que gostam.

A única fonte consultada foi os slides desta cadeira, que proporcionou o pseudocódigo bem como a implementação em *Java* do algoritmo de *Edmonds-Karp*.