

G4: Algoritmos sobre Grafos Pesados: Programação Dinâmica

Nota inicial

Neste módulo consideraremos, sem perda de generalidade, que $V = \{0, \dots, |V| - 1\}$. Esta indexação dos vértices de um grafo por números naturais será útil uma vez que os algoritmos que discutiremos se baseiam na representação de grafos por matrizes de adjacências.

▼ Fecho Transitivo de um Grafo Orientado

A noção de *fecho transitivo* de uma relação binária E é bem conhecida: trata-se da relação obtida adicionando-se à relação os pares necessários por forma a obter uma relação transitiva.

Para calcular o fecho transitivo de uma relação basta iterar o seguinte passo básico:

- Se $(i, k) \in E$ e $(k, j) \in E$ então $E := E \cup (i, j)$

Se a relação em questão for a relação de adjacência de um grafo $G = (V, E)$, o seu fecho transitivo será um grafo $G' = (V, E')$, em que E' é o fecho transitivo de E .

Note-se que:

Se o vértice v é **alcançável** a partir de u em G , então v será **adjacente** a u em G' .

Aparentemente bastará iterar o passo identificado acima, percorrendo sucessivamente todos os pares de vértices i, j e todos os vértices intermédios k , por forma a determinar se j é alcançado a partir de i passando por k .

```
1 for (i=0; i<n; i++)
```

```

2     for (j=0; j<n; j++)
3         R[i][j] = G[i][j];
4
5     for (i=0 ; i<n ; i++)
6         for (j=0 ; j<n ; j++)
7             for (k=0 ; k<n; k++)
8                 if (R[i][k] && R[k][j]) R[i][j] = 1;

```

No entanto esta solução apresenta problemas. Basta observar que se tivermos as seguintes 3 adjacências:

$0 \longrightarrow 1 \longrightarrow 3 \longrightarrow 2$

então claramente o algoritmo falhará na inclusão da aresta $0 \rightarrow 2$, uma vez que nem $0 \rightarrow 3$ nem $1 \rightarrow 2$ terão sido acrescentadas em R quando $i = 0$ e $j = 2$:

- $0 \rightarrow 3$ só será acrescentada quando $i = 0$ e $j = 3$, o que acontecerá na próxima iteração do ciclo interior
- $1 \rightarrow 2$ só será acrescentada quando $i = 1$ e $j = 2$, o que acontecerá ainda mais tarde, só na próxima iteração do ciclo exterior

O problema pode ser resolvido iterando o algoritmo acima até não haver mais pares acrescentados, o que levaria a um algoritmo de tempo $\mathcal{O}(N^4)$. Existe no entanto uma solução mais simples, que consiste em simplesmente trocar a ordem relativa dos ciclos, passando o vértice intermédio a ser iterado no ciclo mais exterior. O algoritmo resultante é conhecido por algoritmo de Warshall, e executa em tempo $\Theta(N^3)$.

```

1 void warshall (GraphM G, GraphM R, int n)
2 {
3     int i, j, k;
4     for (i=0; i<n; i++)
5         for (j=0; j<n; j++)
6             R[i][j] = G[i][j];
7
8     for (k=0 ; k<n; k++)
9         for (i=0 ; i<n ; i++)

```

```

10     for (j=0 ; j<n ; j++)
11         if (R[i][k] && R[k][j]) R[i][j] = 1;
12     }

```

Será talvez mais fácil entender o seu funcionamento observando o seguinte invariante do ciclo exterior.

No início de cada iteração do ciclo mais exterior, $R[i][j] == 1$ sse existe um caminho (não vazio) de i para j contendo além destes apenas vértices pertencentes ao conjunto $\{0, \dots, k - 1\}$.

Para a prova de preservação deste invariante basta observar que a iteração k vai testar se existe caminho de i para j passando adicionalmente por k , e efectuará $R[i][j] = 1$ se for esse o caso (claro que este valor podia já ser 1 antes!).

Quando $k == n$, na terminação do ciclo, teremos calculado o fecho transitivo, como desejado.

Memoization e Programação Dinâmica

Números de Fibonacci:

Recordemos a definição da sequência de Fibonacci:

$$\text{fib}(1) = 1$$

$$\text{fib}(2) = 1$$

$$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$$

É bem sabido que a implementação recursiva “**top-down**” desta definição é proibitiva, por efectuar muitos cálculos redundantes. Basta expandir $\text{fib}(n - 1)$ na definição acima para observar este fenómeno:

$$\text{fib}(n) = 2 * \text{fib}(n - 2) + \text{fib}(n - 3)$$

Por forma a partilhar estes resultados intermédios, basta efectuar o cálculo “**bottom-up**” armazenando (**memoizing**) os resultados num vector para poderem ser utilizados quando necessário. Isto permite o cálculo em tempo $\Theta(n)$:

```

1  int Fibonacci (int n) {
2      int k, fib[n];
3
4      fib[1] = 1;
5      fib[2] = 1;
6      for (k=3 ; k<=n; k++)
7          fib[k] = fib[k-1] + fib[k-2];
8      return fib[n];
9  }

```

Quando são necessários, os valores de `fib[k-1]` e `fib[k-2]`, em vez de recalculados recursivamente, são simplesmente consultados no vector de resultados.

A estratégia algorítmica conhecida por *programação dinâmica* consiste simplesmente em otimizar a estratégia de cálculo *bottom-up* com memorização de resultados intermédios, observando que, em cada passo de cálculo, são necessários apenas os dois últimos resultados calculados, o que dispensa o armazenamento, neste caso, de toda a série de Fibonacci, bastando guardar em duas variáveis os dois últimos números da sequência.

O resultado é um algoritmo que utiliza apenas espaço em $\Theta(1)$:

```

1  int Fibonacci (int n) {
2      int k, a=1, b=1, t;
3      for (k=3 ; k<=n; k++) {
4          t = a+b;
5          b = a;
6          a = t;
7      }
8      return a;
9  }

```

Distância entre vértices de um grafo:

Consideremos o problema do cálculo da distância entre vértices num grafo pesado.

Naturalmente, uma vez que a distância é o peso do caminho mais curto, podemos aplicar o algoritmo de Dijkstra para identificar o caminho mais curto e considerar então o seu peso. Mas tentemos abordar o problema de outra forma.

Seja $d_k(i, j)$ o peso do caminho mais curto de i para j passando apenas pelos vértices de $S_k = \{0, \dots, k - 1\}$. Esta noção tem uma definição simples, recursiva em k :

$$d_0(i, j) = w_{i,j} \quad [\text{peso da aresta } (i, j)]$$
$$d_{k+1}(i, j) = \min (d_k(i, j) , d_k(i, k) + d_k(k, j))$$

É imediato que a distância entre os vértices é $\delta(i, j) = d_{|V|}(i, j)$

O que é interessante é que este cálculo recursivo apresenta um padrão (**top-down**) muito ineficiente, mas que, tal como no caso do cálculo da sequência de Fibonacci, tem muito potencial para armazenamento de resultados intermédios, se se optar por uma estratégia de cálculo **bottom-up**, calculando e armazenando, por esta ordem, d_1, d_2, \dots, d_n .

Utilizaremos um vector de matrizes (i.e., uma matriz tri-dimensional) tal que $D[k][i][j] = d_k(i, j)$.

É imediato escrever o algoritmo com *memoization* com base na definição anterior:

```
1 void memoDistances (GraphM G, GraphM D[MAX], int n)
2 {
3     int i, j, k;
4     for (i=0; i<n; i++)
5         for (j=0; j<n; j++)
6             D[0][i][j] = G[i][j];
7
8     for (k=0 ; k<n; k++)
9         for (i=0 ; i<n ; i++)
```

```

10     for (j=0 ; j<n ; j++)
11         D[k+1][i][j] = min(D[k][i][j], D[k][i][k]+D[k][k][
j]);
12     }

```

Mais uma vez, o passo decisivo para se obter um algoritmo baseado em programação dinâmica é a observação de que é possível dispensar o armazenamento de matrizes intermédias, usando uma única matriz D que armazena, no início da iteração k , o equivalente à matriz $D[k]$ no algoritmo anterior.

```

1 void dynDistances (GraphM G, GraphM D, int n)
2 {
3     int i, j, k;
4     for (i=0; i<n; i++)
5         for (j=0; j<n; j++)
6             D[i][j] = G[i][j];
7
8     for (k=0 ; k<n; k++)
9         for (i=0 ; i<n ; i++)
10            for (j=0 ; j<n ; j++)
11                D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
12 }
13

```

Observe-se que $d_{k+1}(i, k) = d_k(i, k)$ e $d_{k+1}(k, j) = d_k(k, j)$, e por isso o conteúdo da linha k e a coluna k não são alteradas. É isto que torna desnecessário armazenar a matriz anterior ($k-1$).

All-Pairs Shortest Paths: algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall é uma variante deste que calcula, além das

distâncias, os próprios *caminhos mais curtos entre todos os pares de vértices do grafo*.

Os caminhos são armazenados de forma compacta, numa matriz bi-dimensional de vértices intermédios.

```
1 void FloydWarshall (GraphM G, GraphM D, int P[MAX][MAX], int n
2 )
3 {
4     int i, j, k;
5
6     for (i=0; i<n; i++)
7         for (j=0; j<n; j++) {
8             D[i][j] = G[i][j];
9             P[i][j] = -1;
10        }
11
12    for (k=0 ; k<n; k++)
13        for (i=0 ; i<n ; i++)
14            for (j=0 ; j<n ; j++) {
15                if (D[i][k] + D[k][j] < D[i][j]) {
16                    D[i][j] = D[i][k] + D[k][j];
17                    P[i][j] = k;
18                }
19            }
20 }
```

Para compreender a interpretação da informação de vértices intermédios calculados na matriz P, considere-se o seguinte exemplo (cfr. grafo usado como exemplo em [+G3. Algoritmos sobre Grafos Pesados: Estratégia Greedy](#)):

Para saber qual é o caminho mais curto entre os vértices 0 e 5, consulta-se $P[0][5] == 8$. Isto significa que o caminho passa pelo vértice 8. Consulta-se então

recursivamente:

- $P[0][8] == 6$. Então consulta-se ainda recursivamente:
 - $P[0][6] == -1$
 - $P[6][8] == -1$
- $P[8][5] == -1$

Os valores -1 são casos de paragem, significando que o caminho mais curto corresponde nestes casos a arestas directas. Sendo assim, o caminho mais curto entre 0 e 5 é (0, 6, 8, 5).

O tempo de execução deste algoritmo é em termos assintóticos semelhante ao do algoritmo de Dijkstra (repetido a partir de todos os vértices do grafo).

Código para teste dos algoritmos:

```
1  int main(int argc, const char * argv[]) {
2
3      GraphM gm1 = {
4          { 0, 0, 1, 0 },
5          { 1, 0, 0, 0 },
6          { 0, 0, 0, 1 },
7          { 0, 1, 0, 0 }
8      };
9      GraphM gmr1;
10     int n1 = 4;
11
12     GraphM gm2 = {
13         { NE, 2, NE, NE, NE, 9, 5, NE, NE},
14         { 2, NE, 4, NE, NE, NE, 6, NE, NE},
15         { NE, 4, NE, 2, NE, NE, NE, 5, NE},
16         { NE, NE, 2, NE, 1, NE, NE, 1, NE},
17         { NE, NE, NE, 1, NE, 6, NE, NE, 3},
18         { 9, NE, NE, NE, 6, NE, NE, NE, 1},
19         { 5, 6, NE, NE, NE, NE, NE, 5, 2},
20         { NE, NE, 5, 1, NE, NE, 5, NE, 4},
```

```
21     { NE, NE, NE, NE, 3, 1, 2, 4, NE}
22 };
23 GraphM gmr2[MAX];
24 int paths[MAX][MAX];
25 int n2 = 9;
26
27 warshall (gm1, gmr1, n1);
28 printGraphM(gmr1, n1);
29
30 memoDistances(gm2, gmr2, n2);
31 printGraphM(gmr2[n2], n2); printf("\n");
32
33 // dynDistances(gm2, gmr1, n2);
34
35 FloydWarshall(gm2, gmr1, paths, n2);
36 printGraphM(gmr1, n2); printf("\n");
37 printGraphM(paths, n2);
38 }
39
```