G4: Algoritmos sobre Grafos Pesados: Programação Dinâmica

Nota inicial
Neste módulo consideraremos, sem perda de generalidade, que . 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 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 e então

Se a relação em questão for a relação de adjacência de um grafo , o seu fecho transitivo será um grafo , em que é o fecho transitivo de .
Note-se que:

> *Se o vértice é **alcançável** a partir de em , então será **adjacente** a em .*

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

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

```
        for (j=0; j<n; j++)
            R[i][j] = G[i][j];

    for (i=0 ; i<n ; i++)
        for (j=0 ; j<n ; j++)
            for (k=0 ; k<n; k++)
                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:

então claramente o algoritmo falhará na inclusão da aresta , uma vez que nem nem terão sido acrescentadas em R quando e :

- só será acrescentada quando e , o que acontecerá na próxima iteração do ciclo interior
- só será acrescentada quando e , 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 . 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 .

```
void warshall (GraphM G, GraphM R, int n)
{
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            R[i][j] = G[i][j];

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

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

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 para contendo além destes apenas vértices pertencentes ao conjunto .*

Para a prova de preservação deste invariante basta observar que a iteração vai testar se existe caminho de para *passando adicionalmente por* , 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:

É bem sabido que a implementação recursiva "**top-down**" desta definição é proibitiva, por efectuar muitos cálculos redundantes. Basta expandir na definição acima para observar este fenómeno:

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 :

```c
int Fibonacci (int n) {
    int k, fib[n+1];

    fib[1] = 1;
    fib[2] = 1;
    for (k=3 ; k<=n; k++)
        fib[k] = fib[k-1] + fib[k-2];
    return fib[n];
}
```

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 optimizar 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 :

```c
int Fibonacci (int n) {
    int k, a=1, b=1, t;
    for (k=3 ; k<=n; k++) {
        t = a+b;
        b = a;
        a = t;
    }
    return a;
}
```

Distância entre vértices de um grafo:

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

> *Naturalmente, o problema de cálculo das distâncias num grafo sem pesos é um caso particular deste! Basta considerar pesos = 1 no grafo inicial.*

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 o peso do caminho mais curto de para passando apenas pelos vértices de . Esta noção tem uma definição simples, recursiva em :

[peso da aresta (i, j)]

É imediato que a distância entre os vértices é

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, .

Utilizaremos um vector de matrizes (i.e., uma matriz tri-dimensional) tal que .

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

```
void memoDistances (GraphM G, GraphM D[MAX], int n)
{
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            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 que armazena, no início da iteração , 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 e , 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.

```
void FloydWarshall (GraphM G, GraphM D, int P[MAX][MAX], int
n)
{
    int i, j, k;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
            D[i][j] = G[i][j];
            P[i][j] = -1;
        }

    for (k=0 ; k<n; k++)
        for (i=0 ; i<n ; i++)
            for (j=0 ; j<n ; j++) {
                if (D[i][k] + D[k][j] < D[i][j]) {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = k;
                }
            }
}

```

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 assimptóticos semelhante ao do algoritmo de Dijkstra (repetido a partir de todos os vértices do grafo).

Código para teste dos algoritmos:

```
int main(int argc, const char * argv[]) {

    GraphM gm1 = {
        { 0, 0, 1, 0 },
        { 1, 0, 0, 0 },
        { 0, 0, 0, 1 },
        { 0, 1, 0, 0 }
    };
    GraphM gmr1;
    int n1 = 4;

    GraphM gm2 = {
        { NE,  2, NE, NE, NE,  9,  5, NE, NE},
        {  2, NE,  4, NE, NE, NE,  6, NE, NE},
        { NE,  4, NE,  2, NE, NE, NE,  5, NE},
        { NE, NE,  2, NE,  1, NE, NE,  1, NE},
        { NE, NE, NE,  1, NE,  6, NE, NE,  3},
```

```
        {  9, NE, NE, NE,  6, NE, NE, NE,  1},
        {  5,  6, NE, NE, NE, NE, NE,  5,  2},
        { NE, NE,  5,  1, NE, NE,  5, NE,  4},
        { NE, NE, NE, NE,  3,  1,  2,  4, NE}
    };
    GraphM gmr2[MAX];
    int paths[MAX][MAX];
    int n2 = 9;

    warshall (gm1, gmr1, n1);
    printGraphM(gmr1, n1);

    memoDistances(gm2, gmr2, n2);
    printGraphM(gmr2[n2], n2); printf("\n");

    // dynDistances(gm2, gmr1, n2);

    FloydWarshall(gm2, gmr1, paths, n2);
    printGraphM(gmr1, n2); printf("\n");
    printGraphM(paths, n2);
}

```