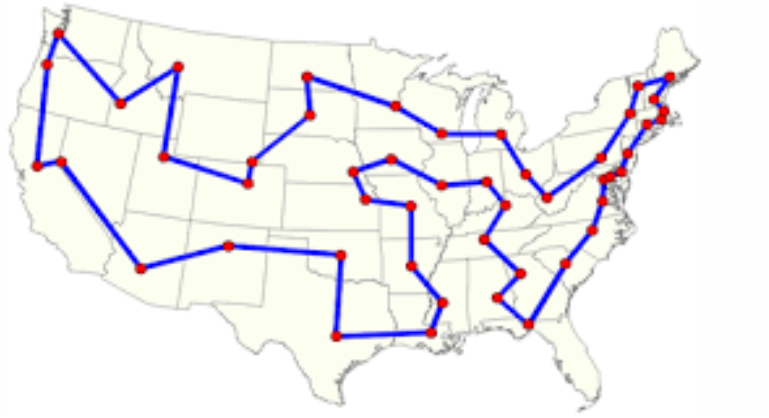
Sistemas Operativos
2020 / 2021

Licenciatura em Engenharia Informática

Projecto de Época de Recurso

Introdução

Dada uma lista de cidades e as respectivas distâncias entre elas, o problema do caixeiro viajante (TSP – *Travelling Salesman Problem*) tem como objectivo determinar o caminho mais curto que um vendedor teria de percorrer de modo a passar por todas as cidades e voltar à cidade de origem.



Sendo n o número de cidades, considere a seguinte matriz de distâncias D com $n = 5$.

$$D = \begin{bmatrix} 0 & 23 & 10 & 4 & 1 \\ 23 & 0 & 9 & 5 & 4 \\ 10 & 9 & 0 & 8 & 2 \\ 4 & 5 & 8 & 0 & 11 \\ 1 & 4 & 2 & 11 & 0 \end{bmatrix}$$

Se considerarmos que as cidades estão numeradas de *1* a *n*, o caminho mais curto de modo a passar por todas as cidades e voltar à cidade de origem é dado por {2, 4, 1, 5, 3}, sendo a distância dada por $d_{2,4} + d_{4,1} + d_{1,5} + d_{5,3} + d_{3,2} = 5 + 4 + 1 + 2 + 9 = 21$.

Para efeitos de comparação, o caminho {1, 2, 3, 4, 5} teria como distância $d_{1,2} + d_{2,3} + d_{3,4} + d_{4,5} + d_{5,1} = 23 + 9 + 8 + 11 + 1 = 52$.

Neste trabalho iremos apenas usar TSP simétricos, i.e., TSP onde as distâncias entre duas cidades são iguais em ambos os sentidos.

Algoritmos de resolução

O algoritmo mais óbvio é o algoritmo de força bruta, em que todos os caminhos (permutações) possíveis são testados até se encontrar o caminho com a distância mínima.

Neste trabalho propomos solucionar o problema do caixeiro viajante utilizando o algoritmo **AJ Evolutivo (AJ-E++)** com a adição de uma heurística.

O algoritmo AJ-E++ pertence à classe dos algoritmos evolutivos, sendo uma versão de um dos seus tipos, o Algoritmo Genético (https://en.wikipedia.org/wiki/Genetic_algorithm). Ao contrário dos algoritmos que trabalham apenas com uma solução, nos algoritmos genéticos trabalha-se sempre com conjuntos de soluções (denominados populações).

O algoritmo funciona da seguinte forma:

1. Inicializa-se aleatoriamente uma população de caminhos.

2. Avalia-se todos os caminhos da população.

3. Escolhe-se os 2 melhores caminhos da população.

4. Realiza a operação de cruzamento PMX (*partially mapped crossover*) aos 2 caminhos escolhidos no ponto anterior. Isto terá como resultado dois novos elementos denominados de descendentes. A descrição desta operação de cruzamento encontra-se em anexo.

5. Aplica-se uma mutação (*exchange mutation*) a cada um dos 2 descendentes. Nesta mutação ocorre a troca de elementos (cidades) entre duas posições aleatórias desse caminho. Esta operação só deve ser executada tendo em conta uma dada probabilidade (por exemplo só será executado 1% das vezes).

6. Calcula-se a distância total dos 2 caminhos obtidos após a mutação.

7. Junta-se estes 2 caminhos à população e retiram-se à população obtida os 2 piores caminhos.

8. O algoritmo volta ao ponto 3 enquanto não houver uma condição de término dada pelo tempo ou número de iterações máxima.

No fim, o algoritmo deverá ser capaz de retornar a distância do caminho mais curto encontrado durante a sua execução, que corresponde ao elemento da população final com o menor caminho. Note que a melhor solução encontrada pelo algoritmo pode ou não ser a melhor solução em termos globais.

Implementação concorrencial do algoritmo AJ-E++ (com a adição de uma heurística)

Dado que o algoritmo AJ-E++ tem uma forte componente aleatória, um dos grandes factores que pode influenciar a solução é o número de iterações realizadas pelo algoritmo (ou de forma indirecta, o tempo que se dá ao algoritmo para tentar encontrar a melhor solução).

Desta forma, propomos a implementação paralela e concorrencial do algoritmo nas suas versões *Base* e *Avançada*.

Em cada uma destas versões será executada uma heurística após a execução do Algoritmo AJ-E++ . A heurística consiste basicamente em aplicar o método de força bruta (geração de todas as permutações do caminho) à melhor solução obtida. Como o método de força bruta é muito pesado em termos computacionais, a heurística aplica-se a todos os subconjuntos do caminho obtido cujo tamanho *t* (número parametrizável) seja menor ou igual a 8.

Versão Base

1. Criar *m threads* (número parametrizável) em que cada *thread* corre o algoritmo AJ-E++

2. Após um tempo de execução, as *threads* são interrompidas e cada uma delas atualiza a memória central, com o seu melhor caminho. Dado que duas ou mais *threads* podem aceder simultaneamente à memória central e corrompê-la, a atualização desta deve ser feita de forma concorrencial.

3. Seja o **caminho base** definido como o caminho da melhor solução obtida no ponto 2, e seja *m* o número de threads disponíveis.

    a) Gera-se todas os subconjuntos possíveis do caminho base cujo tamanho seja *t*. Os pontos seguintes são executados em *threads*, uma para cada subconjunto de forma concorrencial, havendo no máximo *m* threads a serem executadas concorrencialmente.

    b) Para cada subconjunto, geram-se todas as permutações desse subconjunto.

    c) Para cada permutação do ponto anterior, substitui no caminho base a permutação na posição correta.

    d) Avalia o caminho obtido no ponto anterior, e se a sua distância for inferior à melhor solução já obtida, deve atualizar a melhor solução.

4. Informa-se o utilizador da melhor solução encontrada no ponto 2 e a melhor solução obtida após a execução da heurística (ponto 3). Esta informação deve ser feita logo que **todas** as *threads* atualizem o seu melhor caminho na memória central no ponto 2 e no ponto 3. Para garantir que esta actualização seja feita de forma adequada, deve ser feita a sincronização na atualização e leitura dos resultados.

Versão Avançada

A versão avançada é semelhante à versão base com as seguintes alterações:

1. De acordo com um parâmetro de entrada que representa uma percentagem do tempo total, em cada múltiplo dessa percentagem de tempo procede-se à seguinte operação:

    a) Junta-se as populações de todas os processos numa única população.

    b) Seja a variável *popSize* o tamanho de cada população em cada processo. Ordena-se essa população e escolhe-se uma nova população com os *popSize* melhores caminhos.

c) Actualiza-se todos os processos com a nova população obtida no ponto 1.b)

2. A operação anterior não deve ser efectuada no final do tempo de execução do algoritmo (último múltiplo da percentagem de tempo total).

Desenvolvimento

A aplicação deverá ser feita na linguagem de programação Java, em Windows (ou no seu sistema operativo preferido), usando as técnicas de programação paralela e concorrencial utilizadas nas aulas laboratoriais, nomeadamente *threads*, semáforos, métodos sincronizados, etc.

Entradas

A entrada de informação é feita usando ficheiros de texto, um para cada problema. Cada ficheiro de texto está separado por linhas, em que na primeira linha é dado o número de cidades e nas restantes linhas é dada a matriz *D* de distâncias.

```
5
 0 23 10  4  1
23  0  9  5  4
10  9  0  8  2
 4  5  8  0 11
 1  4  2 11  0
```

O programa deverá ser capaz de ser invocado pela linha de comandos passando como argumentos o nome do ficheiro de texto com o problema, o número de processos a serem criados, o tempo máximo de execução do algoritmo (em segundos), o tamanho da população, a probabilidade de mutação, e o tamanho de cada subconjunto a usar na execução da heurística. Por exemplo, o comando **tsp prob13.txt 10 60 80 0.01 8** deverá executar o ficheiro de teste "prob13.txt" usando 10 *threads* em paralelo durante 60 segundos, com uma população de 80 caminhos por *thread*, uma probabilidade de 1% do operador de mutação se realizar, e os subconjuntos devem ter tamanho 8.

Resultados

De modo a se validar a qualidade do algoritmo, deverá ser construída uma tabela com as seguintes colunas:

1. Número do teste (de 1 a 10).
2. Nome do ficheiro de teste
3. Tempo total de execução.
4. Número de *threads* usado (parâmetro *m* na descrição dos algoritmos).
5. Tamanho da população.
6. Probabilidade de mutação.

7. Melhor caminho encontrado e sua distância.
8. Número de iterações necessárias para chegar ao melhor caminho encontrado.
9. Tempo que demorou até o programa atingir o melhor caminho encontrado.
10. Melhor caminhado após a aplicação da heurística e sua distância

Cada teste deverá ser repetido 10 vezes para os mesmos parâmetros de entrada, e deverá ser possível obter valores médios de tempo e número de iterações, assim como o número de vezes em que se encontrou o caminho óptimo.

Os ficheiros de teste a utilizar serão disponibilizados no moodle da disciplina, assim como um exemplo de um ficheiro com resultados e respectivas estatísticas.

Entrega e avaliação

Os trabalhos deverão ser realizados individualmente ou em grupos de 2 alunos da mesma turma de laboratório, e deverão ser originais. Trabalhos plagiados ou cujo código tenha sido partilhado com outros serão atribuídos nota **zero**.

Todos os ficheiros deverão ser colocados num **ficheiro zip** (com o número de todos os elementos do grupo) e submetido via *moodle* **até às 23:55 do dia 28/Fevereiro/2021**. Deverá também ser colocado no zip um **relatório em pdf** com a identificação dos alunos, as tabelas de resultados e a descrições das soluções que considerarem relevantes. Este documento deverá ser mantido curto e directo (2-3 páginas).

Irá considerar-se a seguinte grelha de avaliação:

Algoritmo AJ-E++	4.0 val.
Heurística	2.0 val.
Algoritmo concorrencial	
Versão Base	3.0 val.
Versão Avançada	2.0 val.
Outra versão original	2.0 val.
Utilização de memória central	0.5 val.
Utilização de semáforos e de mecanismos de sincronização	1.5 val.
Relatório com a tabela de testes	2.0 val.
Qualidade da solução e código	3.0 val.

Dada a situação de pandemia e estado de emergência em vigor, não haverá discussões de projectos.

Cruzamento PMX

Proposto por Goldberg e Lingle (1985), no cruzamento PMX pretende-se que 2 permutações pais se cruzem e se obtenham duas novas permutações filhos. O algoritmo tem o seguinte funcionamento:

1. Escolher aleatoriamente dois pontos de cruzamento.
2. Trocar estes dois segmentos, entre os pontos de cruzamento, nos filhos que se geram.
3. O resto das cadeias dos filhos obtém-se fazendo mapeamento entre os dois pais:
   a) Se um valor não está contido no segmento trocado, permanece igual.
   b) Se está contido no segmento trocado, então substitui-se pelo valor que o dito segmento tinha no outro pai.

Sugestão: usar dois *arrays* auxiliares que guardem os valores que cada elemento dentro do segmento correspondente nas permutações pais. Fazendo para cada posição *i*:

A1[P2[i]] = P1[i] e A2[P1[i]] = P2[i]

como está no exemplo abaixo.

Exemplo:

```
     P1 = 9 8 4 | 5 6  7 | 1  2 3 10
     P2 = 8 7 1 | 2 3 10 | 9  5 4  6

Após a troca dos segmentos, os filhos são:
     F1 = X X X | 2 3 10 | X  X X X
     F2 = X X X | 5 6  7 | X  X X X

e os arrays auxiliares são
     A1 = X X 5   6 X  X   X  X X X 7
     A2 = X X X   X X  2   3 10 X X X

Para completar F1 e F2, copia-se primeiro os valores que não estão no segmento trocado:
     F1 = 9 8 4 | 2 3 10 | 1  X X X
     F2 = 8 X 1 | 5 6  7 | 9  X 4 X

Depois mapeia-se os valores restantes:
     F1 = 9  8 4 | 2 3 10 | 1 5 6 7
     F2 = 8 10 1 | 5 6  7 | 9 2 4 3
```

```
public class TestPMXCrossover {

    public static void main(String[] args) {
        int n = 10;
        int parent1[] = {9,8,4,5,6,7,1,2,3,10};
        int parent2[] = {8,7,1,2,3,10,9,5,4,6};
        int offSpring1 [] =  new int[n];
        int offSpring2 [] =  new int[n];

        Random rand = new Random();
        pmxCrossover(parent1, parent2,offSpring1,offSpring2,n,rand);

        for (int i=0; i< n; i++)
            System.out.printf("%2d ",offSpring1[i]);
        System.out.println();

        for (int i=0; i< n; i++)
            System.out.printf("%2d ",offSpring2[i]);

    }

    static void pmxCrossover(int parent1[],int parent2[],
                int offSpring1[],int offSpring2[],int n,Random rand) {
        int replacement1[] = new int[n+1];
        int replacement2[] = new int[n+1];
        int i, n1, m1, n2, m2;
        int swap;

        for (i=0; i< n; i++)
            System.out.printf("%2d ",parent1[i]);
        System.out.println();

        for (i=0; i< n; i++)
            System.out.printf("%2d ",parent2[i]);
        System.out.println();

        int cuttingPoint1 = rand.nextInt(n);
        int cuttingPoint2 = rand.nextInt(n);

        //int cuttingPoint1 = 3;
        //int cuttingPoint2 = 5;

        while (cuttingPoint1 == cuttingPoint2) {
            cuttingPoint2 = rand.nextInt(n);
        }
        if (cuttingPoint1 > cuttingPoint2) {
            swap = cuttingPoint1;
            cuttingPoint1 = cuttingPoint2;
            cuttingPoint2 = swap;
        }

        System.out.printf("cp1 = %d cp2 = %d\n",cuttingPoint1,cuttingPoint2);

        for (i=0; i < n+1; i++) {
            replacement1[i] = -1;
            replacement2[i] = -1;
        }
```

```
        for (i=cuttingPoint1; i <= cuttingPoint2; i++) {
            offSpring1[i] = parent2[i];
            offSpring2[i] = parent1[i];
            replacement1[parent2[i]] = parent1[i];
            replacement2[parent1[i]] = parent2[i];
        }

        for (i=0; i< n+1; i++)
            System.out.printf("%2d ",replacement1[i]);
        System.out.println();

        for (i=0; i< n+1; i++)
            System.out.printf("%2d ",replacement2[i]);
        System.out.println();

        // fill in remaining slots with replacements
        for (i = 0; i < n; i++) {
            if ((i < cuttingPoint1) || (i > cuttingPoint2)) {
                n1 = parent1[i];
                m1 = replacement1[n1];

                n2 = parent2[i];
                m2 = replacement2[n2];
                while (m1 != -1) {
                    n1 = m1;
                    m1 = replacement1[m1];
                }
                while (m2 != -1) {
                    n2 = m2;
                    m2 = replacement2[m2];
                }
                offSpring1[i] = n1;
                offSpring2[i] = n2;
            }
        }
    }
}
```