INF1771 - Inteligência Artificial - (2018.2)
Trabalho 1 – Minimum Latency Problem

Rodrigo Pumar

Bruno Pedrazz

Implementação em Java

- Para solução do problema, foi implementado dois algoritmos:
- Dijkstra's adaptado para contabilizar latência

- Algoritmo Genético com PMX crossover e mutação de troca

Dijkstra's contabilizando latência

- Foi implementado um algoritmo guloso que é uma implementação do algoritmo Dijkstra's com a adaptação de guardar a latência de cada nó visitado, durante sua visita.
- Pela natureza do algoritmo guloso, o percurso guloso foi sempre o mesmo devido a previsibilidade de andar sempre para o nó mais próximo.

Dijkstra’s contabilizando latência

```
// agoritimo guloso
public int[] getTourGuloso(){
    HashSet<Integer> unvisited = new HashSet<Integer>();
    for (int i : cities)
        unvisited.add(new Integer(i));
    unvisited.remove(new Integer(cities[0]));
    int[] tour = new int[cities.length];
    tour[0] = cities[0];
    latencyArray[0]=0f;
    for (int i = 1; i < tour.length; i++) {
        int predecessor = tour[i - 1];
        double minDist = Double.MAX_VALUE;
        int nextCity = -1;
        // vizinho mais proximo
        for (Integer city : unvisited) {
            int currCity = city.intValue();
            double dist = dt.getDistanceBetween(predecessor, currCity);
            if (dist < minDist) {
                minDist = dist;
                nextCity = currCity;
            }
        }
        tour[i] = nextCity;
        latencyArray[nextCity-1]=latencyArray[predecessor-1] + minDist;
        unvisited.remove(new Integer(nextCity));
    }
    return tour;
}
```

Algoritmo Genético com PMX crossover e mutação de troca

- Metaheurística selecionada foi um algoritmo genético.
- Crossover Mapeado Parcialmente (PMX) . Foi usado um vetor de reposicionamento para lidar com as colisões dos genes fora do segmento cortado durante o PMX.

```
for (int i = 0; i < cities.length; i++) {
    if ((i < start) || (i > end)) {
        int n1 = parent1[i]-1;
        int m1 = replacement1[n1];

        int n2 = parent2[i]-1;
        int m2 = replacement2[n2];

        while (m1 != -1) {
            n1 = m1;
            m1 = replacement1[m1];
        }

        while (m2 != -1) {
            n2 = m2;
            m2 = replacement2[m2];
        }

        child1[i] = n1+1;
        child2[i] = n2+1;
    }
}
```

Geração de indivíduos

- A primeira geração contem caminhos aleatórios acrescentado, porem foi acrescentado o caminho da solução gulosa, visto que sabemos que ela é comparativamente melhor que os primeiros indivíduos aleatórios, para que comecemos com pelo menos um pai bom.

```
//Generate a total of [number] Random individuals and
//one gulosoMLP solution adding these to the list of nodes
private void GenerateIndividuals(int number){
    int peep[] = new int[cities.length];
    peep = cities.clone();
    for (int i=0;i<number;i++) {
        int[] poolTourTemp = shuffleArray(peep);
        Integer[] TourTempInteger = toObject(poolTourTemp);
        poolData.add(fillData(TourTempInteger));
    }
    int[] gulosoPath=getTourGuloso();
    Integer[] gulosoInteger = toObject(gulosoPath);
    poolData.add(fillData(gulosoInteger));
}
```

Função heurística

```
// calcula latencia total do percurso
private static double getTourLatency(Integer[] tourTempInteger, double[] latencythis) {
    double totlatency=0;
    for (int i = 0; i < tourTempInteger.length; i++) {
        totlatency +=latencythis[i];
    }
    return totlatency;
}
```

- A função heurística considerada foi a própria latência total do caminho, e usamos o paradigma de que não sabíamos o ótimo/melhor latência alcançável (OPT/BKS), pois normalmente em problemas reais não saberíamos e por isso não usamos esse OPT/BKS na função heurística.

Seleção dos pais

```
sortRemoveDuplicates();
selectIndividuals(capableparents);
int max=ChildData.size()-1;
int min=capableparents;
Set<Integer> unique = new HashSet<Integer>();
for (int j=0;j<10;j++) {
        unique.add(rand.nextInt((max - min) + 1) + min); //so as to not be repeated
}
for (Integer k : unique) {
    poolData.add(ChildData.get(k));// adds not capable parents so that we have
    //more variation even if currently they are bad parents
}
```

- A seleção dos pais que iriam procriar foi implementada ordenando os melhores pais, e sempre pegando os X melhores pais, porem como foi aprendido em aula que é bom ter pais não necessariamente bons, foi adicionado a listagem de pais Y filhos com índice que não estavam entre esses melhores pais, assim adicionando sempre um numero fixo de filhos não ótimos e assim mantendo a variabilidade nos indivíduos em cada geração de maneira bem simples.

Resultados

Instancia	BKS/OPT		Nossa Solução Algoritmo Genético (40 gerações)		Nosso Guloso Dijkstra's	
	Tempo de execução (ms) (paper)	Latência Total	Tempo de execução (ms)	Latência Total	Tempo de execução (ms)	Latência Total
brazil58	550	512361	308	592365.0	3.86	600705.0
dantzig42	170.0	12528	98	12967.0	0.28	13514.0
gr120	9540.0	363454	190	389187.0	1.26	390569.0
gr48	310.0	102378	106	113992.0	0.23	113992.0
pa561	1155320.0	658870	779	779036.0	15.99	779400.0

Resultados

- O nosso algoritmo genético teve melhoras significativas contra o guloso, porem mesmo com o aumento do número de gerações, não conseguimos nos mover mais perto do próximo do BKS/OPT que justificasse o aumento do tempo de execução. Não sabemos se o algoritmo funciona perfeitamente para sair de máximos locais, visto não fizemos estatística da geração de filhos.

Conclusão

- A implementação do algoritmo genético foi bastante iterativa, usando conceitos aprendidos em aula. A implementação deu bastante margem para criatividade e iteração numérica nos valores para maximizar a eficiência do resultado.
- Em problemas reais em que não se sabe qual o melhor possível, acreditamos que essa analise da performance e corretude do algoritmo se tornam bem mais complicados, pois ajudou muito saber qual o máximo ótimo, como parâmetro para guiar e ajustar o código e os parâmetros.