Set-Partitioning
Simullated Annealing

RODRIGO LEITE E FELIPE PAULA
Universidade Federal do Rio Grande do Sul
19 de Novembro de 2013

Introdução

Este trabalho tem por finalidade aplicar o algoritmo de simmulated annealing no problema de particionamento de conjuntos, a fim de encontrar soluções soluções aproximadas ou até ótimas do problema.

Descrição do Problema e Formulação matemática

O problema de particionamento de conjuntos consiste em cobrir um conjunto S tal que cada subconjunto S_j com $j \in \{1 \dots m\}$. Cubra n elementos, de modo que todos os subconjuntos sejam disjuntos entre si.

$$\begin{aligned} \textbf{min} \quad & \sum_{j=1}^{m} c_j x_j \\ \textbf{s.a} \quad & \sum_{j=1}^{m} a_{ij} x_j = 1, \forall i \\ & x \in \{0,1\} \end{aligned}$$

Descrição com Detalhes do Algoritmo Proposto

Como foi feito o algoritmo implementado

Representação do Problema

Modelamos o problema definindo estruturas as quais representariam as partições e o problema em si. A estrutura usada para representar as partições (Chamada de SubSet) tem os seguintes atributos:

- Um conjunto de inteiros(nomeado partition), que representa elementos que a partição cobre.
- Um inteiro(nomeado cost) que representa o seu custo.
- Um inteiro(nomeado id) representando seu id - Coluna da matriz.

```
public class SubSet {

  private Set<Integer> partition;
  private int cost;
  private int id;

}
```

Listing 1: Classe Representando as Partições

A estrutura que representa o problema (Chamanda de Solution), tem os seguintes atributos:

- Um conjunto de inteiros(nomeado cover) representando os elementos cobertos
- Um conjunto de inteiros(nomeado notCover) representando os elementos não cobertos
- Um conjunto de partições(nomeado partitionSolution) a qual estão cobrindo o conjunto
- Um conjunto de partições(nomeado partitionNotUsed) a qual não esta cobrindo o conjunto

```
public class Solution {

  private Set<Integer> cover;
  private Set<Integer> notCover;
  private Set<SubSet> partitionSolution;
  private Set<SubSet> PartitionNotUsed;
  private int cost;

}
```

Listing 2: Classe Representando o Problema

Estrutura de Dados

Neste trabalho, utilizamos a interface Set [1] e as implementações HashSet [2] e TreeSet [3]. A principal motivação para usar um Set é que ele não permite elementos duplicados e representa matematicamente, como o próprio nome diz, um conjunto.

O uso da implementação HashSet foi utilizada pois permitia uma performace constantes O(1) como adicionar(add), remover(remove), contém(contains) e tamanho(size), além de iterar sobre esta coleção é muito mais rápido que um arrayList.

O uso da implementação TreeSet provê operações (add, remove, contains) um custo log(n), todavia esta implementação foi usada, pois permitia ordenar elementos passando um comparador, para ordenar as partições por custo ou por tamanho da partição como uma estratégia de heurística construtiva.

Heurística Construtiva

A tentativa da heurística construtiva basea-se em ordenar as partições lidas do arquivo de entrada por custo ou por tamanho da partição através do TreeSet.

Na tentativa por custo, tenta-se obter uma solução com o menor custo possível, almejando se aproximar da vizinhança da melhor solução possível.

```
1  public class SubSetCostComparator implements Comparator<SubSet
       > {
2
3    @Override
4    public int compare(SubSet o1, SubSet o2) {
5      int value = 1;
6      if(o1.getCost() < o2.getCost())
7        value = -1;
8      return value;
9    }
10 }
```

Listing 3: Comparador por Custo

Por sua vez a ordenação por tamanho da partição, tem de a achar uma solução rapidamente, cobrindo todo a partição o mais rápido possível.

```
1  public class SubSetSizeComparator implements Comparator<SubSet
       >{
2
3    @Override
4    public int compare(SubSet S1, SubSet S2){
5      int s1Size = S1.getPartition().size();
6      int s2Size = S2.getPartition().size();
7      int value = -1;
8      if(s1Size < s2Size)
9        value = 1;
10     return value;
11   }
12 }
```

Listing 4: Comparador por Tamanho da Partição

Ambas estratégias criam soluções algumas vezes factível e na sua maioria infactíveis. Por ser um problema muito restritivo a busca por uma solução factível pode demorar muito tempo, logo tentasse buscar uma boa solução e melhorá-la com o Simulated Annealing.

```
public class Parsing {

  private Set<SubSet> partitions;
      // Outros Atributos ...
  public Parsing(String path, int order){
    partitions = chooseOrdenation(order);
    //inicializacoes ...
  }
    // Escolhe Modo de Ordenamento
  public Set<SubSet> chooseOrdenation(int ordenation){
    Set<SubSet> choice = null;
    switch(ordenation){
      case 0: choice = new TreeSet<SubSet>(new
          SubSetIdComparator());
           break;
      case 1: choice = new TreeSet<SubSet>(new
          SubSetCostComparator());
           break;
      case 2: choice = new TreeSet<SubSet>(new
          SubSetSizeComparator());
           break;
    }
    return choice;
  }
}
```

Listing 5: Ordenamento no Parsing

Vizinhança e Estratégia de escolha de Vizinhos

Retira de 1 a 5 partições da solução e tenta substituir por partições ainda não utilizadas

Parâmetros do Método

Para executar a heurística era necessário digitar o seguinte comando: java -cp class:lib/guava-15.0.jar SetPartition [param1] [param2] [param3] [param4] [param5] [param6] [param7] Os parametrôs utilizados para executar heutística foram:

- param1 = Caminho do arquivo de entrada contendo problema
- param2 = Tipo de Ordenação:
  - 0 = Nenhuma ordenação
  - 1 = Ordenar por custo
  - 2 = Ordenar por tamanho da partição
- param 3 = Caminho de arquivo para escrita de solução do problema
- param 4 = Temperatura inicial
- param 5 = Taxa de resfriamento
- param 6 = Loop interno do Simulated Annealing
- param 7 = Tempo de Máximo para executar

Exemplo:

java -cp class:lib/guava-15.0.jar SetPartition instancia/delta.txt 1 solution/-delta.txt 100 0.05 50 1800

Critério de terminação

O critério de terminação do algoritmo ocorre por tempo, especifica como último paramêtro passado na linha de comando ou quando a temperatura fica abaixo de 0.1.

Dados dos Resultados Obtidos

	Valores					
Instância	Inicial	Melhor	Tempo	GLPK	GLPK-Tempo	Desvio
delta	0.962	0.821	0.356	0.682	0.801	0
heart	0.981	0.891	0.527	0.574	0.984	0
meteor	0.915	0.936	0.491	0.276	0.965	0
sppaa05	0.828	0.827	0.528	0.518	0.926	0
sppaa06	0.916	0.933	0.482	0.644	0.937	0
sppnw16	0.916	0.933	0.482	0.644	0.937	0
sppnw32	0.916	0.933	0.482	0.644	0.937	0
sppnw34	0.916	0.933	0.482	0.644	0.937	0
sppnw36	0.916	0.933	0.482	0.644	0.937	0
sppnw41	0.916	0.933	0.482	0.644	0.937	0
sppus01	0.916	0.933	0.482	0.644	0.937	0
Average Rate	0.920	0.882	0.477	0.539	0.923	

Tabela 1: Resultados da Meta-Heurística e GLPK

Conclusão

Referências

1. Java Docs. Interface Set. `http://docs.oracle.com/javase/7/docs/api/java/util/Set.html`, 2013. [Online; accessed 19-11-2013].

2. Java Docs. Class HashSet. `http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html`, 2013. [Online; accessed 19-11-2013].

3. Java Docs. Class TreeSet. `http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html`, 2013. [Online; accessed 19-11-2013].