Universidade Federal do Rio Grande do Sul

Instituto de Informática

Departamento de Informática Teórica

INF05010 – Otimização Combinatória - Turma B (2025/1) | Professor: Henrique Becker

Relatório Preliminar

Alunos: Antonio Carlos Gonçalves Sarti - 00130172

Artur Chiká Miozzo - 00316118

Marcus Vinicius Gonzales dos Santos - 299907

1. Escolha de representação do problema

A representação é feita modelando o problema como um grafo, fazendo uma variação do problema clássico de coloração de grafos. No problema de coloração, queremos que para dois vértices adjacentes, a mesma cor não pode ser aplicada a ambos, ou seja, nunca uma mesma cor será ligada com ela mesma, o que é a mesma coisa, porém os vértices são os criminosos, as arestas são a aliança entre eles e a penitenciária é a cor, onde dois criminosos aliados não podem ser alocados na mesma penitenciária. A representação de dados é feita da seguinte forma:

- **Criminosos como vértices do grafo**
  - Cada criminoso é representado como um vértice do grafo. Não existe uma estrutura para os criminosos, isso é feito usando índices nas listas, onde na solução final **solution[i]** representa o criminoso **i**.
- **Alianças como arestas**
  - As alianças entre criminosos (comparsas) são representadas como arestas não direcionadas. A estrutura **alliances** é um **set** de tuplas **(c1,c2)** com **c1 < c2**, para evitar duplicatas, onde **c** é o índice de um criminoso.
  - **adj** (lista de adjacência) representa os vizinhos de cada criminoso, onde **adj[i]** é a lista de criminosos que tem aliança com o criminoso **i**.
- **Penitenciária como rótulo**
  - As penitenciárias são representadas em uma lista de penitenciárias. **possible_pens** é a lista de penitenciárias onde **p** é uma penitenciária com valor inteiro de 0 até o número de penitenciárias menos 1.
- **Soluções como listas de inteiros**
  - A alocação dos criminosos nas penitenciárias é representada por uma lista **solution**, onde **solution[i]** é uma penitenciária **p**, isso é, na solução final, o criminoso **i** deve ser alocado a penitenciária **p**.
  - Essa lista é a representação central da solução do problema.

2. Construção da solução inicial

A solução é feita usando uma estratégia gulosa baseada no número de aliados de cada criminoso. A ideia é alocar primeiro os criminosos que exigem mais penitenciárias para minimizar a quantidade delas. Inicialmente, a solução é gerada populando a lista **solution** com o número de criminosos onde para todo criminoso **i**, **solution[i] = -1**, representando que o criminoso **i** não foi alocado a nenhuma penitenciária.

Em seguida, os criminosos são ordenados pelo número de alianças na chamada, isso é, **criminals_ordered** é uma lista ordenada por len(adj[i]) para cada criminoso **i**, e então, para cada criminoso, tenta alocá-lo na primeira penitenciária que for possível. Se não existir nenhuma, cria uma penitenciária nova e segue alocando até que todos criminosos estejam distribuídos respeitando as restrições de aliança.

3. Principais estruturas de dados

As principais estruturas de dados são:

- Lista de inteiros **solution**, que representa o qual a penitenciária cada criminoso está alocado.
- Conjunto de pares **alliances**, onde cada entrada representa a aliança entre dois criminosos.
- Lista de listas **adj**, que representa as alianças de cada criminoso, sendo uma lista de criminosos e para cada uma lista de alianças.
- Lista de inteiros **possible_pens**, que representa todas as penitenciárias, sendo criada a cada iteração dado o número atual de penitenciárias, que é potencialmente incrementado.

4. Vizinhança e a estratégia de escolha de vizinhos (quando aplicável)

- A vizinhança é definida por soluções que diferem da atual pelo reposicionamento de criminosos para outras penitenciárias. O programa seleciona 5 vizinhos (default), onde ele pega 5 criminosos aleatórios, e tenta colocá-los na primeira penitenciária que der, em uma ordem aleatória, testando todas elas. Esses 5 vizinhos então são testados, e o primeiro que melhorar a solução é escolhido para a próxima iteração. Caso uma melhora seja encontrada, a busca reinicia. O processo para quando não há mais vizinhos que proporcionem melhorias, indicando um ótimo local.

5. Processo de combinação e factibilização (quando aplicável)

- a factibilidade de um candidato, ou resposta é bem simples: se o candidato contém criminosos na mesma aliança na mesma penitenciária, a solução é infactível. no caso do nosso problema, na pior das hipóteses, cada criminoso estará na sua própria penitenciária, mas o teste de factibilidade nos ajuda manter as respostas do começo ao fim factíveis.

6. Parâmetros do método (valores usados nos experimentos)

- os parâmetros dos métodos são, o número de iterações (foi usado 1000), a seed (foi usado 1) e a força da perturbação, ou seja, a quantidades de vizinhos criados para testar (foi usado 5)

7. Critério de parada (NÃO pode ser diretamente tempo)

- o programa para quando o número de iterações chegar no máximo.

8. Comparações preliminares e execuções mínimas:

- durante os testes exploratórios, observamos os seguintes comportamentos ao variar os parâmetros:

- **Número de iterações (max_ils_iterations)**: Usamos o valor de **1000**, pois valores muito baixos (como 100 ou 300) para alguns casos não permitiam que o algoritmo escapasse de ótimos locais iniciais. Já valores muito altos (acima de 3000), mostraram ganhos pouco expressivos com custo computacional elevado.
- **Força da perturbação (perturbation_strength)**: Testamos os valores 3, 5, 7 e 10. Notamos que:
  - Valores muito baixos (como 3) resultaram em baixa diversificação;
  - Valores muito altos (como 10) dificultavam a recuperação de soluções boas;
  - **Optamos por 5 (default)**, pois equilibrava bem exploração e intensificação.
- **Semente aleatória (random_seed)**: Usamos valor fixo 1 para garantir reprodutibilidade. A influência da semente foi limitada, com pequenas variações entre execuções e em alguns casos pouca melhora no valor objetivo.

Tabela 1 - parâmetros e resultados						
Instância	**Iterações**	**Semente**	**Perturbação**	***Target***	***Bound***	**Valor Objetivo**
01.txt	1000	1	5	3	3	**3**
02.txt	1000	1	5	4	4	**4**
03.txt	1000	1	5	5	5	**6**
04.txt	1000	1	5	10	9	**10**
05.txt	1000	1	5	13	8	**12**
06.txt	1000	1	5	14	6	**10**
07.txt	1000	1	5	18	13	**19**
08.txt	1000	1	5	20	9	**17**
09.txt	1000	1	5	11	7	**13**
10.txt	1000	1	5	39	13	**36**

9. Outras observações:

- Ao mesmo tempo dos testes acima, foram executados testes mais radicais com o mesmo sucesso. Aqui seguem suas conclusões:

- A semente oferecida resulta em resultados diferentes, mas pouco melhores. Os números de iterações e de vizinhos resultam em quase pouco ganho para muito esforço. Logo os problemas estão na implementação do código, mais especificamente, na representação da vizinhança. Aprende-se na cadeira de AI que para melhor resultado, deve-se combinar uma junção de “breath” e “depth” quando cria-se a vizinhança de uma heurística, ou seja, uma árvore “n-aria”. Nossa vizinhança atual cria apenas *n* vizinhos e a melhor melhoria é 1, resultando numa escolha de vizinho muito “cega”. Acreditamos que nosso algoritmo pode ser melhorado com essa implementação básica na criação de candidatos para a próxima iteração do algoritmo, e logo após um ajuste nos parâmetros para fazer com que ele rode ainda em um tempo razoável.