7
Algoritmos

Este capítulo apresentará sete algoritmos para o GAP: cinco deles implementados para este trabalho, enquanto dois são os algoritmos que têm os melhores resultados da literatura para o problema. Dos cinco algoritmos implementados, quatro foram desenvolvidos originalmente para este trabalho e alguns se utilizam de componentes descritos em seções anteriores, como a Vizinhança Elipsoidal (capítulo 5) e RINS (seção 4.3). Este capítulo detalhará os algoritmos desenvolvidos e mencionará também os algoritmos da literatura que estarão presentes nos experimentos do capítulo seguinte.

Os algoritmos estão listados a seguir em ordem crescente de "especificidade": os primeiros são poucos específicos ao GAP e podem facilmente ser implementados e adaptados a outros problemas, equanto os últimos algoritmos são completamente acoplados ao problema em questão e utilizam heurísticas de difícil implementação, que não podem ser facilmente utilizadas em outros problemas. Essa ordenação será repetida na comparação dos resultados: o interesse deste trabalho é analisar a performance de algoritmos "genéricos" (que podem ser facilmente aplicados a qualquer MIP) e comparar estes resultados com os de algoritmos heurísticos altamente especializados.

7.1
CPLEX

O algoritmo CPLEX é a implementação direta da formulação clássica do GAP (descrita na seção 6.1). Somente um parâmetro do resolvedor foi alterado: o limite máximo de tempo, como descrito na seção anterior. Neste momento, cabe mencionar que, apesar de ser utilizado na teoria como uma caixa-preta, basicamente um motor combinatórico que resolve MIPs rapidamente, o CPLEX é um algoritmo extremamente intrincado e que possui um enorme número de parâmetros e configurações. Os manuais distribuídos junto com o software possuem centenas de páginas com parâmetros que podem ser customizados para cada tipo de problema e situação. Neste trabalho, o conjunto de parâmetros padrão praticamente não foi alterado por dois motivos:

– otimizar os parâmetros do resolvedor MIP vai contra a teoria de que os

algoritmos que o utilizam são gerais e podem ser facilmente aplicados a diferentes problemas;

– a ILOG, fabricante do software, recomenda o conjunto de parâmetros padrão para a maioria das situações.

O único parâmetro alterado, para fazer maior proveito do hardware utilizado, foi o uso de múltiplas *threads* pelo resolvedor. Desta maneira, o CPLEX usou *multi-threading* de maneira agressiva, o que gera melhores resultados, mas pode gerar resultados levemente diferentes mesmo para a mesma instância. Este parâmetro também foi alterado para todos os algoritmos que utilizam o CPLEX como resolvedor MIP.

7.1.1 Encontrando várias soluções

A partir da versão 11 o CPLEX oferece a possibilidade de, ao resolver um modelo MIP, retornar não só a melhor solução viável encontrada, mas também um conjunto de soluções viáveis encontradas durante a busca. Essa capacidade de retornar várias soluções é especialmente interessante para algoritmos que mantêm e precisam inicializar um conjunto de soluções. Existem vários parâmetros para controlar quantas soluções são armazenadas, quais os critérios para armazenar (ou descartar) uma solução, etc. Neste trabalho, nos algoritmos que utilizam essa estratégia, todos os parâmetros originais foram mantidos.

7.2 Pós-processamento com vizinhanças elipsoidais (PostProcessing)

Um dos resultados mais importantes do trabalho de Pigatti et al. [30], onde a vizinhança elipsoidal para duas soluções foi proposta, era um excelente resultado obtido da seguinte forma: a partir de duas soluções de boa qualidade e bastante diferentes uma da outra, foi feita uma busca na vizinhança elipsoidal destas soluções, onde uma excelente solução foi encontrada.

Este resultado sugere um algoritmo simples para testar a qualidade das vizinhanças elipsoidais como um mecanismo de pós-processamento: a partir de um conjunto de soluções de boa qualidade, realizar buscas em vizinhanças elipsoidais usando algumas destas soluções como base. O algoritmo 9, chamado de PostProcessing, é uma simples implementação desta estratégia.

O PostProcessing funciona de maneira simples. Na linha 1, calcula-se quanto tempo será dado ao resolvedor MIP. A idéia é que o resolvedor tenha uma fração do tempo total disponível, de maneira que ainda sobre tempo para fazer $repet \cdot (max - min + 1)$ buscas em vizinhanças elipsoidais. Na linha 2,

Algoritmo 9 PostProcessing

Entrada: número máximo e mínimo de soluções-base na vizinhança *min* e *max*, repetições para cada tamanho *repet*

Saída: *x_best*, melhor solução encontrada na busca

1: TL = tempo para o resolvedor (tempo total menos o tempo das buscas nas vizinhanças elipsoidais)
2: S = resolver MIP(com limite de tempo TL, retornando o conjunto das melhores soluções encontradas durante a busca)
3: *x_best* = melhor solução de S
4: calcular a distância entre todas as soluções de S
5: **for** cada tamanho de vizinhança v em $[min, max]$ **do**
6: Sub = sub-conjuntos de S de tamanho v
7: Ordenar Sub, comparando cada sub-conjunto pela soma da distância entre seus elementos
8: Sub = os *repet* sub-conjuntos de maior distância
9: **for** cada sub-conjunto T em Sub **do**
10: x = EllipsoidalSearch(T)
11: **if** x é melhor que *x_best* **then**
12: *x_best* = x
13: **end if**
14: **end for**
15: **end for**
16: **return** *x_best*

o resolvedor MIP é executado, com a instrução de retornar todas as soluções viáveis encontradas durante sua busca - essas soluções são armazenadas no conjunto S. Na linha 3, a variável *x_best* é inicializada com a melhor solução de S. Na linha 4 são calculadas as distâncias entre todas as soluções de S. O laço que se inicia na linha 5 itera sobre todos os tamanhos do conjunto-base, de *min* até *max*. Nas linhas 6 a 7 são escolhidos os sub-conjuntos de S que tenham tamanho v e que possuam as soluções mais distantes umas das outras. Finalmente, nas linha 9 a 14 o método EllipsoidalSearch (descrito no algoritmo 8) é chamado, realizando uma busca na vizinhança elipsoidal formada pelas soluções em T. A melhor solução encontrada é retornada na linha 16.

A estratégia do algoritmo PostProcessing é dar ao resolvedor MIP bastante tempo para encontrar várias soluções de alta qualidade e, então, tentar encontrar soluções melhores nas vizinhanças elipsoidais dessas soluções. Neste trabalho, os parâmetros utilizados foram: $min = 2$, $max = 5$, $repet = 4$. Os valores de *min* e *max* foram propositalmente pequenos já que, caso contrário, os passos das linhas 6 a 8 poderiam ser muito custosos computacionalmente. Os parâmetros utilizados em EllipsoidalSearch foram os mesmo utilizados no algoritmo PathRelinkMIP, detalhados na seção 7.4 a seguir.

7.3 Variable Neighborhood Seach Branching (VNSBra)

O algoritmo VNSBra foi implementado como descrito na seção 4.4. Os parâmetros utilizados foram iguais aos sugeridos em [21], exceto o tempo limite local (o tempo máximo permitido para cada busca local), que naquele trabalho variava entre 300s e 2400s, foi fixado em 300s nestes experimentos.

7.4 Path Relink com vizinhança elipsoidal (PathRelinkMIP)

Existem vários motivos que fazem a implementação da metaeurística *Path Relink* (PR) uma abordagem óbvia neste trabalho: a semelhança entre o conceito de vizinhanças elipsoidais e o passo de *relink* da metaeurística *Path Relink* (seção 5.2), os bons resultados apresentados pela metaeurística em um grande conjunto de problemas (seção 2.2.2) e, finalmente, os ótimos resultados apresentados para o GAP pelo algoritmo PREC de Yagiura et al. [39] (seção 6.3.5).

A partir do algoritmo geral de PR (algoritmo 2) e se baseando em algumas idéias utilizadas no algoritmo PREC, propõe-se o algoritmo 10, chamado de PathRelinkMIP. A idéia deste algoritmo é manter sempre um conjunto fixo de diferentes soluções de alta qualidade (conjunto R) - devido ao seu tamanho máximo fixo, esse conjunto possui um método especial de adicionar soluções, descrito no algoritmo 11.

O conjunto R é inicializado nas linhas 2 a 8. Inicialmente, o resolvedor MIP é utilizado (com um limite de tempo fixo) e todas as soluções encontradas durante a busca são adicionadas a R. Se o número de soluções encontradas é menor que o tamanho máximo do conjunto (fix), outras soluções são geradas a partir das soluções encontradas através dos algoritmos Mutate (algoritmo 12) e KOPT (algoritmo 6).

O laço principal do algoritmo é executado enquanto ainda houver tempo restante. Na linha 10, é inicializado um conjunto de soluções V. Esse conjunto de soluções será populado nas linhas 12 e 13 com o resultado de buscas na vizinhança elipsoidal, até ele possuir no mínimo max_relink soluções. Na linha 12, são escolhidas as soluções-base da vizinhança (o número de soluções-base é aleatório e definido pelos parâmetros min e max). Na linha 13, as soluções da busca na vizinhança elipsoidal são adicionadas ao conjunto V - esse é o passo de *relink*. Quando o conjunto V possuir no mínimo max_relink soluções, cada solução de V é otimizada pela busca local KOPT e o resultado é possivelmente adicionado a R. O resultado do algoritmo é a melhor solução encontrada, que estará presente em R.

Algoritmo 10 PathRelinkMIP

Entrada: tempo de busca local t_local, tempo total t_total, tamanho do conjunto de soluções fix, tamanho do conjunto de busca b, máximo de soluções a cada *relink* max_relink, mínimo e máximo da vizinhança elipsoidal min, max

Saída: x_best, melhor solução encontrada na busca

1: $R =$ FixedSetSolutionSet(tamanho máximo fix)
2: $S =$ resolver MIP (limite de tempo $5 \cdot t\_local$, guardar todas soluções encontradas)
3: **while** $|S| < fix$ **do**
4: $S = S \cup$ KOPT(Mutate(solução qualquer de S, $rand(30, 60)$))
5: **end while**
6: **for** cada solução $s \in S$ **do**
7: R.AddSolution(s)
8: **end for**
9: **while** t_total não for atingido **do**
10: $V = \{\}$
11: **while** $|V| < max\_relink$ **do**
12: $E =$ escolher aleatoriamente de min a max soluções de R
13: $V = V \cup$ EllipsoidalSearch(E, guardar max_relink melhores soluções, limite de tempo t_local)
14: **end while**
15: **for** cada solução $s \in V$ **do**
16: R.AddSolution(KOPT(s))
17: **end for**
18: **end while**
19: **return** melhor solução em R

Algoritmo 11 FixedSizeSolutionSet.AddSolution

Entrada: tamanho máximo do conjunto max, solução a ser adicionada s, soluções no conjunto S

Saída: **true**, se a solução foi inserida, **false**, caso contrário

1: **if** $s \in S$ **then**
2: **return false**
3: **else**
4: **if** $|S| < max$ **then**
5: $S = S \cup s$
6: **return true**
7: **else if** s é melhor que a pior solução de S **then**
8: retirar a pior solução de S
9: $S = S \cup s$
10: **return true**
11: **else**
12: **return false**
13: **end if**
14: **end if**

Algoritmo 12 Mutate

Entrada: solução inicial s, número de mutações num
Saída: solução modificada x

```
1: cont = 0
2: x = s
3: while cont < num do
4:    t1, t2 = escolher duas tarefas aleatórias
5:    if é possível trocar t1 com t2 em x then
6:       x = trocar tarefas t1, t2 em x
7:       cont = cont + 1
8:    end if
9: end while
```

O algoritmo EllipsoidalSearch, descrito na seção 5.1, é utilizado como método de busca local nos algoritmos PostProcessing e PathRelinkMIP. Os parâmetros utilizados nos dois foram os mesmos: $opt = 16$ e $max\_failures = 2$. A busca local KOPT também usa o parâmetro $opt = 16$.

7.5 Stabilized branch-and-price com RINS e guided dives (BPRINS)

A formulação com número exponencial de colunas para o GAP é, como descrito na seção 6.2, comprovadamente melhor que a formulação clássica: os *bounds* encontrados são melhores e é possível realizar *branching* nas mesmas variáveis da formulação clássica. O sub-problema da geração de colunas é a mochila 0-1, problema para o qual excelentes algoritmos exatos (tanto *branch and bound* quanto de programação dinâmica) existem.

A implementação ideal de um *branch and price* para o GAP seria implementar a formulação no CPLEX e deixá-lo tomar conta do processo de *branch and bound* - afinal, o resolvedor é conhecido por utilizar diversas técnicas/heurísticas para encontrar excelentes *bounds* e escolher em qual variável realizar o *branching*. Infelizmente, a versão utilizada do resolvedor não permite a implementação de algoritmos *branch and price*. Dessa maneira, para implementar o *branch and price* para o GAP, além de implementar toda a parte de geração de colunas (inclusive a solução do sub-problema), ainda foi preciso implementar toda a lógica do *branch and bound*.

O *branch and price* (BPRINS) implementado é fortemente baseado no algoritmo descrito no trabalho de Pigatti et al. [30]: é utilizada uma técnica para estabilizar a geração de colunas, o *branching* é feito nas variáveis x_{ij} e o algoritmo de programação dinâmica proposto por Pisinger [31] é utilizado para resolver o sub-problema da mochila.

Além das técnicas descritas em [30], o BP possui dois novos componentes:

o uso da heurística RINS (descrita na seção 4.3) e de *Guided Dives* (descritos na seção 4.3.1). Cada vez que a busca atinge um nível de profundidade *depth* par (i.e. um número par de variáveis foram fixadas), a heurística RINS é utilizada com limite de tempo igual a $min(100, 1000/1.3^{depth})$ segundos.

7.6 TSEC e PREC

Os algoritmos TSEC e PREC, descritos respectivamente nas seções 6.3.4 e 6.3.5, não foram implementados neste trabalho. Entretanto, sendo os algoritmos que possuem os melhores resultados da literatura para o GAP, a comparação com os outros algoritmos propostos aqui é de grande valia. Os resultados reportados no capítulo 8 foram retirados de [39].