5
Algoritmos Heurísticos

Este capítulo descreve algoritmos heurísticos para a resolução do PAG. Algoritmos heurísticos são uma boa alternativa quando se tem uma instância muito grande do problema ou quando há pouco tempo disponível para a resolução. Na seção 5.1 são descritos métodos já existentes de resolução do PAG. Foram escolhidos os métodos que produzem, atualmente, os melhores resultados. Em seguida, a seção 5.2 descreve um método, proposto por Fischetti e Lodi [35], onde é usado um resolvedor de programação linear inteira genérico como uma ferramenta caixa-preta para explorar eficientemente um subespaço de soluções. Metaeurísticas podem ser especificadas controlando, em um nível mais alto, este subespaço. Nessa seção descreve-se a proposta deste trabalho para a aplicação deste método ao PAG.

5.1
Revisão da Literatura do Problema (Métodos Existentes)

5.1.1
Busca Tabu

As origens da busca tabu são dos anos 70, mas ela foi apresentada na forma atual pela primeira vez por Glover [4]. As idéias básicas também foram esboçadas por Hansen [5]. Esforços adicionais de elaboração aparecem em Glover [8] [9] e de Werra & Hertz [10]. Vários experimentos computacionais mostraram que a busca tabu é uma técnica de aproximação muito boa, que pode competir com quase todas as demais técnicas conhecidas e que, por sua flexibilidade, pode superar muitos procedimentos clássicos. Até agora, não foi apresentada nenhuma explicação formal do seu bom comportamento. Aspectos teóricos da busca tabu foram investigados por [Faigle & Kern [12]; Glover [13]; Fox [15]].

A busca tabu é um método iterativo. O passo geral de um método iterativo consiste em construir, partindo de uma solução corrente i, uma próxima solução

j e checar se se deve parar o processo ou executar um novo passo. Métodos de procura em vizinhança são procedimentos iterativos nos quais uma vizinhança $N(i)$ é definida para cada solução viável i, e a proxima solução j é procurada dentro das soluções em $N(i)$.

O mais antigo método de procura em vizinhança é o método da descida. Seja f uma função que avalia uma solução. O método da descida consiste em 3 passos:

1. Escolher uma solução inicial i.
2. Encontrar a melhor solução $j \in N(i)$, isto implica que $f(j) \leq f(k)$ para qualquer $k \in N(i)$.
3. Se $f(j) \geq f(i)$, então pare. Se não $i = j$ e volta para o passo 2.

O método da descida claramente pára em um ótimo local mas não necessariamente um ótimo global de f. A busca tabu pode ser considerada um método de busca em vizinhança mais elaborado do que o método da descida.

Com o objetivo de melhorar o processo de exploração, é necessário guardar mais informações além do valor corrente da função objetivo, informações relativas ao passado do processo de exploração. Este uso sistemático de memória é uma característica essencial da Busca Tabu. Estas informações serão usadas para guiar o movimento de i para a proxima solução j a ser escolhida em $N(i)$. O uso da regra de memória restringe a escolha da proxima solução a algum subconjunto de $N(i)$, proibindo alguns movimentos que levariam a algumas dessas soluções vizinhas.

A busca tabu não fica presa a ótimos locais. Ela faz isso permitindo movimentos que não melhoram o valor de f, em outras palavras, ela aceita movimentos de i para j onde $f(i) \geq f(j)$. Como movimentos que não melhoram o valor da solução são possíveis, existe o risco de acontecerem ciclos. É neste ponto que o uso da memoria atua, proibindo movimentos que possam levar a soluções recentemente visitadas. Dessa forma pode-se considerar que a estrutura de $N(i)$ dependerá do itinerário e da iteração k. Então, pode-se considerar $N(i,k)$ ao invés de $N(i)$. Segue abaixo uma melhoria do algoritmo da descida que o torna mais proximo da busca tabu. Escrevendo i^* como sendo a melhor solução encontrada e k o contador de iterações.

1. Escolher uma solução inicial i. Fazer $i^* = i$ e $k = 0$.
2. Fazer $k = k + 1$ e gerar um subconjunto V^* de soluções em $N(i,k)$
3. Escolher o melhor $j \in V^*$ em relação a f ou para alguma função modificada f', e fazer $i = j$.

4. Se $f(i) < f(i^*)$, então fazer $i^* = i$.

5. Se uma condição de parada é encontrada então pare. Se não vá para o passo 2.

Observa-se que o procedimento clássico da descida é incluído nessa formulação fazendo-se o procedimento de parada $f(i) \geq f(i^*)$ e i^* é sempre a última solução. Deve ser notado também que pode ser usada uma função modificada f' ao invés de f em algumas circunstâncias a serem descritas mais adiante.

Algumas das condições imediatas de parada:

– $N(i, k+1) = \varnothing$.
– k ser maior do que o número máximo de iterações permitido.
– O número de iterações desde a última melhoria de valor de i^* ser maior do que um número especificado.
– Evidencias de que uma solução ótima foi encontrada.

Apesar dessas regras de parada terem alguma influência sobre o procedimento de busca e sobre os seus resultados, o fator crucial é a definição de $N(i, k)$ a cada iteração k e a escolha de V^*.

A definição de $N(i, k)$ implica que algumas soluções recentemente visitadas são removidas de $N(i)$. Elas são consideradas soluções Tabu, e devem ser evitadas na proxima iteração. Este esquema de memória previne parcialmente a existência de ciclos. Por exemplo, mantendo na iteração k uma lista T (lista tabu) de pelo menos $|T|$ soluções visitadas previne a existência de ciclos de tamanho até $|T|$. Então poderia-se fazer $N(i, k) = N(i) - T$. Entretanto, esta lista T seria de uso impraticável. Então o processo de exploração é descrito em termos de movimentos de uma solução para a proxima. Para cada solução i, define-se $M(i)$ como o conjunto dos movimentos m que pode ser aplicados a i para obter uma nova solução j. Notação $j = i \oplus m$. Então $N(i) = \{j \mid \exists m \in M(i) \text{ com } j = i \oplus m\}$. Em geral são usados movimentos que são reversíveis: para cada m existe um movimento m^{-1} tal que $(i \oplus m) \oplus m^{-1} = i$. Desta maneira ao invés de manter uma lista T das últimas $|T|$ soluções visitadas, mantém-se apenas os últimos $|T|$ movimentos ou os últimos $|T|$ movimentos reversos associados com os movimentos realizados. É claro que este procedimento causa uma perda de informação, mas ele garante que nenhum ciclo de tamanho até $|T|$ ocorrerá.

Por eficiência, pode ser conveniente usar várias listas T_r em um tempo determinado. Então alguns constituintes t_r (de i ou de m) darão um status tabu para

indicar se eles estão atualmente permitidos ou não em um movimento. Genericamente o status tabu de um movimento é uma função de seus constituintes, os quais devem mudar a cada iteração. Desta forma pode-se formular uma coleção de condições tabu como:

$$t_r(i,m) \in T_r, \qquad r = 1, \ldots, t.$$

Um movimento m aplicado a uma solução i será um movimento tabu se todas as condições são satisfeitas.

Outra desvantagem da simplificação da lista tabu de soluções para movimentos, é o fato de que pode-se dar um status tabu para uma solução que ainda não foi visitada. Faz-se necessária então uma relaxação do tabu status. Viola-se a regra do status tabu quando uma solução parecer atrativa. Isto será realizado por condições de nível de aspiração.

Um movimento tabu m aplicado a uma solução corrente i pode ser atraente porque ele fornece uma solução melhor do que a melhor encontrada até o momento. É preferível aceitar m apesar de seu status. Isso deve ser feito se ele tem um nível de aspiração $a(i,m)$ melhor do que um valor limitante $A(i,m)$.

Genericamente $A(i,m)$ pode ser visto como um conjunto de valores preferidos para a função $a(i,m)$. Desta forma condições de aspiração podem ser escritas na forma

$$a_r(i,m) \in A_r(i,m), \qquad r = 1, \ldots, a$$

Se pelo menos uma dessas condições é satisfeita pelo movimento tabu m aplicado a i, o movimento m será aceito apesar do seu status.

Há mais uma importante característica da busca tabu a ser descrita. Desde que f em algumas instâncias pode ser trocada por outra função f', podem ser introduzidas algumas intensificações e diversificações na busca.

A memória é usada de diferentes maneiras para guiar o procedimento de busca tabu. Foi mostrado uma memória de curto prazo, cuja utilidade era proibir que alguns movimentos levassem a busca de volta para soluções recentemente visitadas. A memória também pode ser usada num nível mais profundo.

Algumas vezes é vantajoso intensificar a busca em alguma região porque ela pode conter algumas boas soluções. Tal intensificação pode ser realizada dando uma prioridade alta para soluções que tenham características em comum com a solução corrente. Isto pode ser atingido introduzindo um termo extra na função objetivo, que penaliza soluções longe da solução corrente. Intensificação pode ser executada em poucas iterações. Então pode ser útil explorar outra região. Esta diversificação tenderá a extender os esforços de exploração para diferentes regiões, e pode ser feita introduzindo-se um termo na função objetivo. Em algum estágio este termo penaliza soluções que estão próximas da solução corrente. Os termos da intensificação e da diversificação têm pesos que são

modificados através da busca de maneira que o processo alterna de uma tática para outra. Então $f' = f$+intensificação+diversificação.

Busca Tabu

1. Escolher uma solução inicial i. Fazer $i^* = i$ e $k = 0$.
2. Fazer $k = k + 1$ e gerar um subconjunto V^* de soluções em $N(i, k)$ tal que cada uma das condições tabu $t_r(i, m) \in T_r$ é violada $(r = 1, \ldots, t)$ ou no mínimo uma das condições de aspiração $a_r(i, m) \in A_r(i, m)$ para $(r = 1, \ldots, a)$.
3. Escolher um melhor $j = i \oplus m \in V^*$ em relação a f ou f', e fazer $i = j$.
4. Se $f(i) < f(i^*)$, então fazer $i^* = i$
5. Atualizar as condições tabu e de aspiração.
6. Se uma condição de parada for encontrada, então parar. Se não ir para o passo 2.

Como foi descrito até o momento, a busca tabu utiliza uma coleção de princípios de solução inteligente de problemas. Ela usa estruturas flexíveis de memória, um mecanismo de controle associado para ser usado com as estruturas de memoria para definir as restrições Tabu, registros de informações históricas que permitem o uso de estratégias de intensificação e diversificação do processo de busca. A busca Tabu é baseada no uso de memória adaptativa usada para guiar o processo de busca para sair de ótimos locais e obter soluções próximas do ótimo global. Restrições Tabu são usadas para evitar que o processo de busca cicle por soluções já visitadas. Estratégias de intensificação visam explorar características históricas desejáveis enquanto estratégias de diversificação forçam a busca a examinar regiões não visitadas.

A seguir será descrito o algoritmo de busca tabu para o problema do PAG.

Uma solução π para o PAG é representada por um vetor de n elementos, onde n é o numero de tarefas a serem alocadas e o k-ésimo elemento do vetor é o agente ao qual a tarefa k está associada, isto é: $\pi = (\pi_1, \ldots, \pi_n)$, onde $\pi_j \in I; \pi_j = i \Longleftrightarrow x_{ij=1}$.

Dado o conjunto S das soluções possíveis para um problema de otimização, uma vizinhança define para cada solução $\pi \in S$, um conjunto $S_\pi \subset S$ de soluções que em algum sentido estão próximas de π. Neste método são considerados dois tipos simples de vizinhanças clássicas: shift e troca. Dada uma solução π, a vizinhança shift, N_{shift}, é definida sendo o conjunto de soluções que podem ser obtidas realocando um trabalho de um agente para

outro. $N_{shift}(\pi) = \left\{(\pi'_1, \ldots, \pi'_n) \,|\exists j^* \in J \; t.q. \; \pi'_j \neq \pi_{j^*}, \pi'_j = \pi_j \; \forall j \neq j^*\right\}$. A vizinhança de troca, N_{troca} é o conjunto de soluções que podem ser obtidas trocando a alocação de duas tarefas, inicialmente alocadas em diferentes agentes, uma pela outra. $N_{troca}(\pi) = \left\{(\pi'_1, \ldots, \pi'_n) \,|\exists j_1, j_2 \in J, \pi_{j_1} \neq \pi_{j_2}, t.q. \; \pi'_{j_1} = \pi_{j_2}, \pi'_{j_2} = \pi_{j_1}, \pi'_j = \pi_j \; \forall j \neq j_1, j_2\right\}$.

O passo da busca tabu é feito saindo-se de uma solução para outra solução que está contida no conjunto de soluções vizinhas da solução anterior. O seguinte conjunto de regras define o caminhamento neste método.

-Movimentos em $N_{shift} \cup N_{troca}$ que não são Tabu-ativos são considerados admissíveis.

-Para a alocação corrente $\pi = (\pi_1, \ldots, \pi_n)$, tarefas são processadas para diminuir o valor do somatório dos custos das alocações.

-Para uma dada tarefa j, movimentos de shift e troca são avaliados e o melhor movimento admissível, com relação a função objetivo que é baixar o somatório dos custos, é associado à tarefa.

-O mecanismo de geração de candidatos pára se o melhor movimento adimissível para a tarefa sendo processada melhora o valor da função objetivo. A cada iteração, os movimentos candidatos são regerados.

-De outra maneira, se todos as tarefas foram processadas mas nenhum dos movimentos associados melhoram a função objetivo, o movimento adimissível com o menor incremento é selecionado.

As estratégias de diversificação e intensificação podem ser utilizadas após um número determinado de iterações.

5.1.2 Algoritmos Genéticos

Algoritmos evolutivos que modelam o processo de evolução natural foram propostos pela primeira vez nos anos 60. Uma pesquisa de estratégias de busca baseadas na evolução natural foi feita por Mühlenbein, Gorges-Schleuter & Krämer [6].

Os chamados algoritmos evolutivos são dirigidos principalmente por mutação e seleção. Em termos biológicos estes algoritmos modelam a reprodução assexuada de uma população.

Algoritmo Evolutivo

1. Criar uma população inicial de tamanho λ.
2. Computar a adaptabilidade $F(x_i), i = 1, \ldots, \lambda$.
3. Selecionar os $\mu \leq \lambda$ melhores indivíduos.

4. Criar λ/μ descendentes para cada um dos μ indivíduos por variação pequena.

5. Se não terminou, retornar ao passo 2.

Um algoritmo evolutivo é uma busca randomica que usa seleção e variação. A variação pequena é feita escolhendo-se randomicamente um número de uma distribuição normal com 0. Este número é adicionado ao valor da variável continua. O algoritmo adapta o montante da variação mudando a variancia da distribuição normal.

Algoritmos de busca que modelam a reprodução sexuada são chamados de Algoritmos Genéticos. A reprodução sexuada é caracterizada por combinar duas strings pais em uma string descendente. Esta combinação é chamada de crossover. Os algoritmos genéticos foram introduzidos por Holland [3].

Algoritmo Genético

1. Definir uma representação genética do problema.

2. Criar uma população inicial $P(0) = x_1^0, \ldots, x_N^0$. Fazer $t = 0$.

3. Computar a média de adaptabilidade $f'(t) = \sum_i^N f(x_i)/N$. Associar a cada indivíduo o valor de adaptabilidade normalizada $f(x_i)/f'(t)$.

4. Associar cada x_i uma probabilidade $p(x_i, t)$ proporcional a sua adaptabilidade normalizada. Usando esta distribuição, selecione N vetores de $P(t)$. Isto fornece o conjunto de pais selecionados.

5. Agrupar os pais selecionados em $N/2$ pares. Aplicar crossover com uma certa probabilidade em cada par e outro operador genético como mutação, formando uma nova população $P(t+1)$.

6. Fazer $t = t + 1$ e voltar ao passo 2.

No caso mais simples a representação genética é somente um string de tamanho n, o *cromossomo*. As posições do string são chamadas *locus* do cromossomo. A variável em um locus é chamada *gene*, seu valor um *alelo*. O conjunto de cromossomos é chamado *genótipo*, o qual define um *fenótipo* (o indivíduo) com uma certa adaptabilidade.

O operador genético *mutação* muda o valor de cada posição do string com uma probabilidade m. O operador *crossover* trabalha combinando dois strings. Se dois strings $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$ são dados, então o operador *crossover uniform* combina os dois strings como segue: $z = (z_1, \ldots, z_n)$, $z_i = x_i$ ou $z_i = y_i$.

Normalmente x_i ou y_i são escolhidos com probabilidade igual.

Um algoritmo genético é um algoritmo de busca probabilística inteligente que simula o processo de evolução sobre uma população de soluções aplicando operadores genéticos em cada reprodução. Cada solução na população é avaliada de acordo com alguma medida de adaptação. Soluções com alto grau de adaptabilidade tem a oportunidade de se reproduzir. Novas proles de soluções são geradas e soluções inadequadas na população são trocadas. Este ciclo de avaliação-seleção-reprodução é repetido até que uma solução satisfatória seja encontrada. Em um algoritmo genético uma solução potencial para o problema é representada com um conjunto de parâmetros chamados de genes. Estes parâmetros são unidos para formarem uma string de valores conhecida como um cromossomo. Um bom esquema de representação é importante quando se usa Algoritmos Genéticos devendo ser claramente definidos os significados das operações de crossover, mutação e outros operadores específicos do problema. Para incorporar estas características, nesta abordagem é usada uma representação eficiente na qual a estrutura da solução é uma estrutura ordenada (vetor de n dimensões) de números inteiros. Estes números inteiros identificam o agente ao qual o trabalho representado pelo índice do vetor está associado. Esta representação assegura que a restrição do PAG de que todas as tarefas sejam atendidas por um único agente é respeitada. No entanto, esta representação não garante que as restrições de capacidade de cada agente sejam respeitadas. Os passos envolvidos neste algoritmo genético para o PAG são:

1. Gerar uma população inicial de N soluções construídas randomicamente. Nota-se que as soluções iniciais podem violar a restrição de capacidade.

2. Decodificar a solução para obter o valor de adaptabilidade. A adaptabilidade de uma solução é calculada de acordo com uma função. Normalmente deve ser levado em consideração não somente o custo de uma solução mas também seu grau de violação com relação a restrição de capacidade.

3. Selecionar duas soluções para serem geradoras de outra solução filha. É usado o método de seleção por torneio binário. Neste método duas soluções são escolhidas aleatoriamente e a que tem a melhor função de adaptabilidade é escolhida para ser um dos pais.

4. Gerar uma solução filha aplicando primeiro um operador crossover sobre as soluções pais selecionadas. É usado o simples operador crossover onde um ponto $p \in J$ é selecionado randomicamente e a solução filha consistirá dos primeiros p genes da primeira solução pai e os restantes $(n - p)$ genes são da segunda solução pai. O procedimento de crossover é seguido

por um processo de mutação que envolve trocar elementos em 2 genes selecionados randomicamente, isto é, trocar os agentes associados entre dois trabalhos selecionados randomicamente.

5. Trocar um indivíduo na população pela solução filha gerada. O indivíduo com o mais alto grau de não adaptabilidade deverá sair da população para dar lugar a nova solução filha gerada.

6. Passos 3 a 5 são repetidos até que M filhos não duplicados tenham sido gerados sem melhorar o valor da melhor solução encontrada.

5.1.3 Algoritmo Path Relinking

Nesta seção é abordado o algoritmo PREC (path relinking e ejection chains) descrito em [34]. Este algoritmo é uma extensão da busca local. A busca local começa de uma solução inicial σ e repetidamente troca σ pela melhor solução de sua vizinhança $N(\sigma)$ até que nenhuma melhor solução seja encontrada em $N(\sigma)$. A solução resultante σ é localmente ótima no sentido de que não existe melhor solução em sua vizinhança. Uma vizinhança shift N_{shift} é normalmente usada em métodos de busca local para o PAG, onde $N_{shift}(\sigma) = \{\sigma' | \sigma'$ é obtido de σ mudando-se a alocação de uma tarefa$\}$. O PREC usa uma vizinhança ejection chain, que consiste de uma solução obtida por certa seqüência de movimentos shift. Dado que o tamanho de tal vizinhança pode se tornar exponencial, seu tamanho é cuidadosamente limitado usando-se a informação da relaxação lagrangiana do GAP. A busca local resultante é chamada EC probe. Para detalhes sobre EC probe, ver (Yagiura, Ibaraki, Glover [27]).

Quando a busca visita regiões infactíveis, avalia-se a solução por uma função objetivo penalizada pela infactibilidade:

$pcusto(\sigma) = custo(\sigma) + \sum_{i \in I} \alpha_i p_i(\sigma)$,

onde $p_i(\sigma) = \max\left\{0, \sum_{j \in J, \sigma(j)=i} a_{ij} - b_i\right\}$. Os parâmetros $\alpha_i\ (> 0)$ são adaptavelmente controlados durante a busca usando-se o algoritmo descrito em (Yagiura, Ibaraki, Glover [27]).

As soluções iniciais para EC probes são geradas por path relinking, que é uma metodologia para gerar soluções a partir de duas ou mais soluções. Aqui é gerada uma seqüência $\sigma_0, \sigma_1, \sigma_2, \ldots$ de soluções a partir de suas soluções σ_A e σ_B como segue. Primeiro faz-se $\sigma_0 = \sigma_A$. Então, para cada $k = 1, 2, \ldots,$ é definido σ_k como sendo a solução em $N_{shift}(\sigma_{k-1})$ com o melhor $pcusto$ entre aquelas que a distancia até σ_B é menor do que a distancia até σ_{k-1}, onde a distância entre duas soluções σ e σ' é definida como $|\{j \in J | \sigma(j) \neq \sigma'(j)\}|$. Seja d a distância

entre duas soluções σ_A e σ_B, então $\sigma_d = \sigma_B$. Mantém-se as melhores γ soluções de $\{\sigma_1, \sigma_2, \ldots, \sigma_{d-1}\}$ em um heap, e usa-se estas soluções como soluções iniciais para o EC probes.

Sempre que o path relink é aplicado, as duas soluções σ_A e σ_B são randomicamente escolhidas de um conjunto referencia R ($|R|$ é um parâmetro) de boas soluções. Então, antes de aplicar o path relinking, um shift randomico é aplicado a σ_B com probabilidade $1/2$ para manter a diversidade da busca. Inicialmente R é gerada randomicamente. Uma vez que uma solução factível é encontrada, uma das soluções em R é a solução responsável por esta nova solução factível. Novas soluções são inseridas em R como segue. Sempre que EC probe pára, a solução ótima localmente σ_{lopt} é trocada pela pior solução σ_{pior} em R (exceto a solução responsável pela nova solução factível), sob a condição que σ_{lopt} não seja pior do que σ_{pior} e seja diferente de todas as soluções em R.

5.2 Uma Metaeurística de "*Local Branching*"para o PAG

Fischetti e Lodi[35] recentemente propuseram uma técnica para melhorar o desempenho de resolvedor de programação linear inteira qualquer, no sentido de favorecer a obtenção de boas soluções viáveis logo no início do algoritmo de branch-and-bound. A motivação para isso é evidente, como em numerosos casos o tempo que o resolvedor necessita para achar a solução ótima de um problema e provar a sua otimalidade pode ser excessivo, seria de grande valia dispor de boas soluções o mais cedo possível para que se possa interromper a execução do resolvedor depois de decorrido um tempo razoável. Ou seja, essa técnica visa a melhorar o desempenho de um resolvedor de programação linear como uma heurística.

As potenciais vantagens mencionadas dessa abordagem em relação às metaeurísticas tradicionais são as seguintes:

– Facilidade de implementação. Construir uma boa heurística tradicional para um certo problema costuma ser uma tarefa difícil e exige um estudo mínimo das características desse problema. Por exemplo, implementar uma simples busca local k-opt (troca de até k elementos de uma solução) de forma eficiente normalmente exige algoritmos e estruturas de dados sofisticados. Por outro lado, formular um problema usando programação inteira e passá-lo para um resolvedor costuma ser fácil.

– Flexibilidade e facilidade de manutenção. Quando se aplica otimização a problemas reais complexos, a definição exata do problema costuma ser mudada diversas vezes ao longo de um projeto. Por exemplo, é comum surgirem novas restrições não previstas inicialmente. Adaptar uma heurística tradicional para incorporar uma nova restrição pode ser uma tarefa difícil. Isso é particularmente verdadeiro justamente quando essa heurística foi implementada de forma eficiente. Nesse caso, algoritmos e estruturas de dados criados para tirar vantagem de cada particularidade do problema podem simplesmente não serem adaptáveis para a nova situação. Por outro lado, pequenas alterações em uma formulação em programação inteira costumam ser fáceis.

Fischetti e Lodi propõem o uso, no nível "tático", do resolvedor de programação linear inteira como uma ferramenta caixa-preta para explorar eficientemente subespaços das soluções (vizinhanças) definidas e controladas, em um nível "estratégico", por um algoritmo de branching externo simples. O procedimento lembra em seu espírito as bem conhecidas metaeurísticas, mas a vizinhança é obtida através da introdução, no modelo de programação linear inteira,

de desigualdades inválidas chamadas de cortes de *local branching*. Na nova técnica, a busca de soluções é desenvolvida alternando entre estratégias de alto nível de branching para definir vizinhanças de soluções, e branchings táticos de baixo-nível (definidos pelo resolvedor de programação linear inteira) para explorá-las. O resultado pode então ser visto como uma estratégia de branching de dois-níveis que produz soluções melhoradas a cada estágio da computação.

Essa nova técnica pode ser contrastada com a técnica implementada na maioria dos resolvedores de programação atuais para realizar uma busca que privilegie a rápida obtenção de boas soluções e que é baseada na fixação de "agressiva"de variáveis. Nesse esquema heurístico, resolve-se um problema linear e analisa-se uma solução fracionária $\overline{x}$. De acordo com regras mais ou menos complexas, algumas variáveis fracionárias são fixadas aos valores inteiros mais próximos de seus valores fracionários. O método é então reaplicado no problema restringido resultante da fixação. Uma nova solução fracionária é encontrada, algumas de suas variáveis são fixadas, e assim por diante. Desta maneira o tamanho do problema é significativamente reduzido após cada fixação, até um ponto em que o resolvedor provavelmente consegue resolvê-los a otimalidade. Existem também técnicas supostamente capazes de concluir que uma variável está "mal fixada"e eliminar essa fixação na tentativa de obter uma melhor solução.

Naturalmente, o ponto crítico em métodos de fixação de variáveis está relacionada a escolha das variáveis a serem fixadas em cada passo. Para problemas difíceis, só se obtém um problema reduzido possível de ser resolvido após vários ciclos de fixação. Isso faz com que escolhas erradas nos níveis de fixação iniciais sejam muito difíceis de serem detectadas. Desta forma torna-se necessário embutir no esquema um mecanismo de backtracking capaz de se recuperar de uma má fixação, o que é uma tarefa muito difícil e computacionalmente cara.

A questão é então como fixar um número relevante de variáveis sem perder a possibilidade de encontrar boas soluções. Para melhor ilustrar este ponto, suponha que seja dada uma solução 0-1 heurística $\overline{x}$ de um programa linear inteiro puramente 0-1 com n variáveis, e deseja-se explorar o subproblema resultante da fixação em 1 de pelo menos 90% das variáveis diferente de zero. Como deveria ser feita a escolha das variáveis a serem fixadas? Colocado nestes termos, a questão tem uma simples resposta: apenas adicionar ao modelo de programação linear inteira uma restrição de fixação *fraca* da forma

$$\sum_{j=1}^{n} \overline{x}_j x_j \geq \left\lceil 0.9 \sum_{j=1}^{n} \overline{x}_j \right\rceil$$

e aplicar o resolvedor ao modelo de programação linear inteira resultante. Desta maneira é evitada uma fixação muito rígida das variáveis. Esta restrição também define uma vizinhança da solução $\overline{x}$ corrente, que é explorada pelo

resolvedor em busca de uma solução melhor. Na restrição exemplificada acima, assume-se que os 10% de folga no lado direito dirigem o resolvedor de maneira eficiente a fixar um numero grande de variáveis, mas com um grau de liberdade razoável o que possibilita ao resolvedor encontrar melhores soluções.

O mecanismo de fixação fraca mencionado acima pode ser visto como o framework descrito a seguir. É considerado um resolvedor de programação linear inteira genérico com variáveis 0-1 da forma:

$$(P) \quad \min c^T x$$
$$Ax \geq b$$
$$x_j \in \{0,1\}$$

Dada uma solução de referencia $\overline{x}$ de (P), seja $\overline{S} = \{j \mid \overline{x}_j = 1\}$. Para um parâmetro inteiro positivo k, define-se como a $k - OPT$ vizinhança $N(\overline{x}, k)$ de $\overline{x}$ o conjunto de soluções factíveis de (P) satisfazendo a restrição de *local branching* adicional:

$$\Delta(x, \overline{x}) = \sum_{j \in \overline{S}} (1 - x_j) + \sum_{j \notin \overline{S}} x_j \leq k$$

onde os dois termos do lado esquerdo contam o número de variáveis trocando seus valores (em relação a $\overline{x}$) de 1 para 0 e de 0 para 1, respectivamente.

A restrição acima pode ser escrita de maneira mais conveniente em sua forma "assimétrica"equivalente:

$$\sum_{j \in \overline{S}} (1 - x_j) \leq k' \, (= k/2)$$

A definição acima é consistente com a vizinhança clássica $k' - OPT$ para o problema do caixeiro viajante (TSP). Aplicando-se a restrição acima sobre a formulação do TSP sobre variáveis de arestas, ela permite que no máximo k' arestas sejam trocadas da rota de referencia $\overline{x}$. No caso do PAG a restrição acima garante que no máximo k' tarefas troquem de máquina na alocação de referencia $\overline{xp}$.

A restrição de *local branching* pode ser usada como um critério de branching com um esquema enumerativo para (P). Dada a solução $\overline{x}$, o espaço solução associado com o nó da ramificação corrente pode ser particionado pela disjunção:

$\Delta(x, \overline{x}) \leq k$ (ramo esquerdo) ou $\Delta(x, \overline{x}) \geq k + 1$ (ramo direito)

O parâmetro tamanho da vizinhança k deve ser escolhido como o maior valor capaz de gerar um subproblema possível de resolver em tempo razoável. A idéia é que a vizinhança $N(\overline{x}, k)$ correspondente ao ramo esquerdo deve ser "suficientemente pequena"para ser otimizada com um curto tempo de computação, mas "grande o bastante"para conter soluções melhores do que $\overline{x}$.

Uma primeira implementação do *local branching* é mostrada na figura 5.1, onde os triângulos marcados com a letra "T"(Tatico) correspondem às sub-

arvores de branching exploradas através de um critério de ramificação qualquer embutido no resolvedor de programação linear inteira. Eles representam a aplicação desse resolvedor de forma exata.

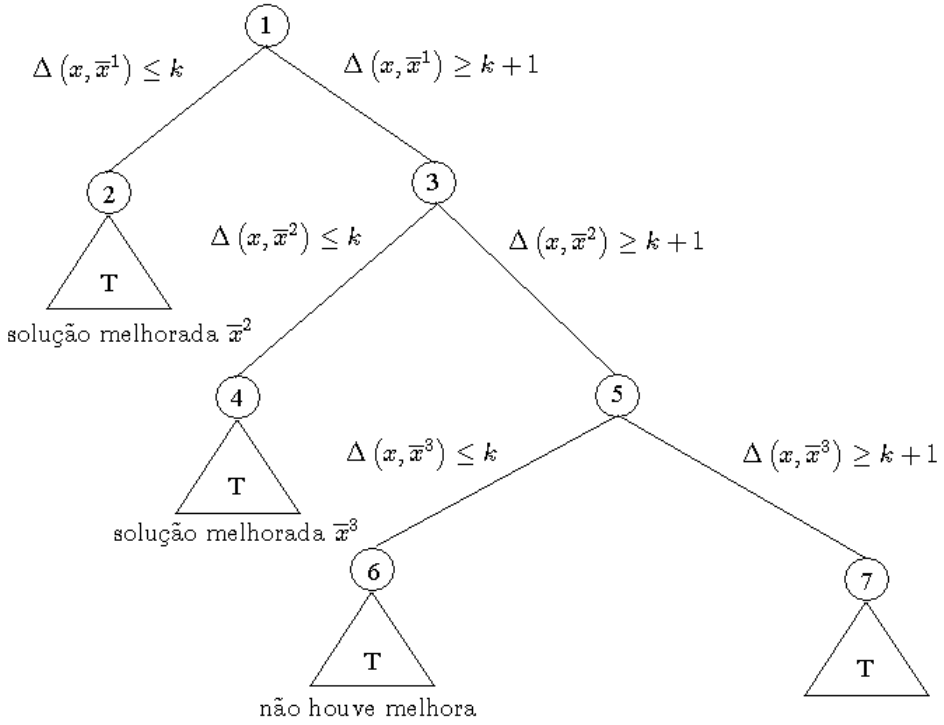

Figura 5.1: Esquema básico do Local Branching.

Na figura 5.1, tem-se uma solução inicial $\overline{x}^1$ no nó raiz 1. O ramo esquerdo nó 2 corresponde a otimização com a $k-OPT$ vizinhança $N(\overline{x}^1, k)$, que é feita através de um esquema de ramificação tático que converge para uma solução ótima $\overline{x}^2$. Esta solução se torna então a nova solução de referência. O esquema é então aplicado novamente no lado direito do ramo nó 3, onde a exploração de $N(\overline{x}^2, k) \setminus N(\overline{x}^1, k)$ no nó 4 produz a nova solução de referência $\overline{x}^3$. O nó 5 é então endereçado, ele corresponde ao problema inicial (P) com mais duas restrições adicionais $\Delta(x, \overline{x}^1) \geq k+1$ e $\Delta(x, \overline{x}^2) \geq k+1$. Neste exemplo, o ramo esquerdo nó 6 produz um subproblema que não contem nenhuma solução melhor do que a solução de referência. Nesta situação a adição da restrição de ramificação $\Delta(x, \overline{x}^3) \geq k+1$ dirige-se ao ramo direito nó 7, que é explorado pela ramificação tática. Deve ser notado que a solução fracionária LP do nó 1 não é necessariamente cortada nos dois nós filhos 2 e 3, como é o caso quando se aplica o branching padrão nas variáveis. O mesmo é válido para os nós 3 e 5. A filosofia do *local branching* é bem diferente da filosofia de branching padrão. Não deseja-se forçar o valor das variáveis fracionárias, mas sim instruir o método de solução a explorar primeiro algumas regiões promissoras do espaço

de soluções. A vantagem do esquema de *local branching* é que ele costuma levar a uma alteração mais rápida e mais freqüente da solução de referência até que se chegue a um ponto onde o *local branching* não possa mais ser aplicado (nó 7, no exemplo).

Em alguns casos, a solução exata do nó do ramo esquerdo pode consumir muito tempo para um determinado valor do parâmetro k. Do ponto de vista heurístico, é razoável que se imponha um tempo limite para a computação do ramo esquerdo. No caso deste limite ser excedido, podem ocorrer dois casos:

1. Se a solução referência for melhorada, deve-se voltar ao nó pai e criar um novo ramo esquerdo associado a nova solução referencia, sem modificar o valor do parâmetro k. Esta situação é ilustrada na figura 5.2, onde o nó 3 atualmente tem 3 filhos: o nó 4, para o qual o limite de tempo foi atingido com uma solução melhorada $\overline{x}^3$, e os nós regulares esquerdo 4' e direito 5. Deve ser notado que, no exemplo, a vizinhança associada ao nó 4 não foi explorada completamente, desta forma seria matematicamente incorreto adicionar a condição do ramo direito $\Delta(x, \overline{x}^2) \geq k + 1$ nos nós 4' e 5.

2. Se o limite de tempo é atingido sem que se encontre uma solução melhorada, deve ser reduzido o tamanho da vizinhança. Isto é feito dividindo-se o valor de k por um fator $\alpha > 1$. Esta situação é ilustrada na figura 5.3, onde o nó 3 tem 3 filhos: nó 4, onde o limite de tempo foi atingido sem que se encontrasse uma solução melhorada, nó 4', para o qual a redução do tamanho da vizinhança permitiu encontrar uma solução provadamente ótima $\overline{x}^3$, e o nó 5.

Outra melhoria no método pode ser obtida adicionando-se mecanismos de diversificação bem conhecidos, idênticos aos das metaeurísticas de busca local. A diversificação é aplicada sempre que é provado que o nó esquerdo corrente não contem uma solução melhorada. Este caso acontece no nó 6 na figura 5.1. Com o objetivo de manter um controle estratégico sobre a enumeração nesta situação, são usados dois mecanismos de diversificação diferentes. Primeiro é aplicado uma diversificação "fraca"consistindo em aumentar o tamanho da vizinhança corrente de um fator $\beta > 1$. A diversificação produz então um nó de ramo esquerdo que é processado pelo branching tático com um certo limite de tempo. No caso de não ser encontrada uma solução melhorada mesmo na vizinhança aumentada, é aplicado o passo da diversificação "forte", tal como em [19]. Neste caso procura-se por uma solução (tipicamente pior do que a solução referencia) que não está "muito longe"de $\overline{x}^2$, por exemplo, uma solução factível x tal que

$$\Delta(x, \overline{x}^2) \leq \lfloor \beta^2 k \rfloor$$

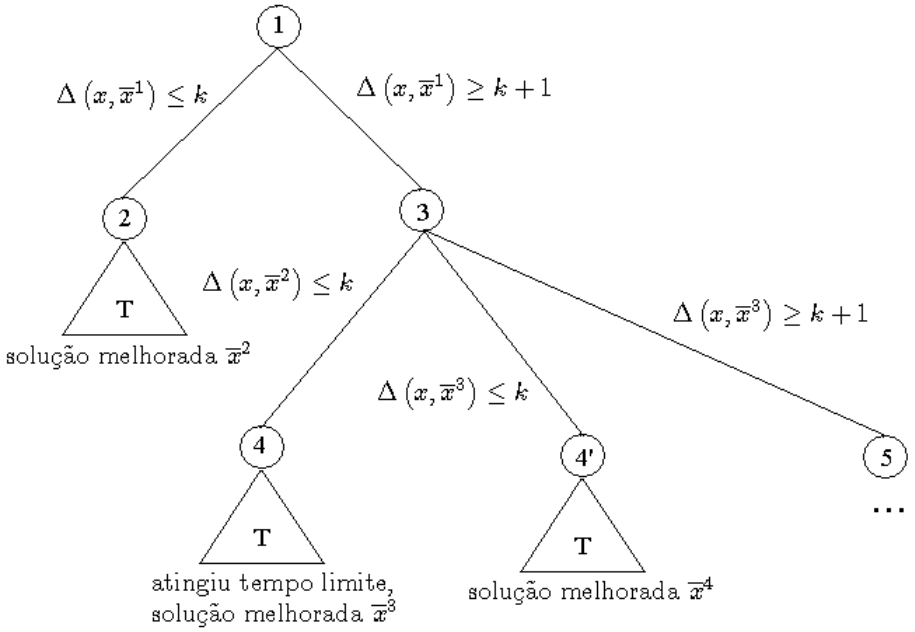

Figura 5.2: Trabalhando com um limite de tempo: caso 1.

Na implementação do *Local Branching*, isto é atingido aplicando-se ramificações táticas ao problema corrente aumentado da restrição acima, mas sem impor qualquer limite ao valor da solução ótima. A exploração é abortada assim que se encontra a primeira solução factível. Esta solução $\overline{x}^N$(tipicamente pior do que a melhor solução corrente) é então usada como a nova solução referência, e o método é reaplicado com o objetivo de iterativamente melhorar $\overline{x}^N$.

O método consiste em um laço principal que é executado até que o número máximo de iterações seja atingido. A cada iteração um problema de programação linear inteira deve ser resolvido pelo resolvedor.

Quatro diferentes casos podem acontecer depois de cada chamada ao resolvedor:

1. **Solução Ótima Encontrada**: o modelo de programação linear inteira corrente foi resolvido a otimalidade, desta maneira a ultima restrição de *local branching* deve ser revertida, a solução de referencia $\overline{x}$ é atualizada para a nova encontrada e o processo é iterado novamente.

2. **Provou Inviabilidade**: foi provado que o modelo de programação linear inteira corrente não tem um valor de solução factível que respeite um valor limite. Assim sendo, a ultima restrição de *local branching* é revertida. O processo é iterado.

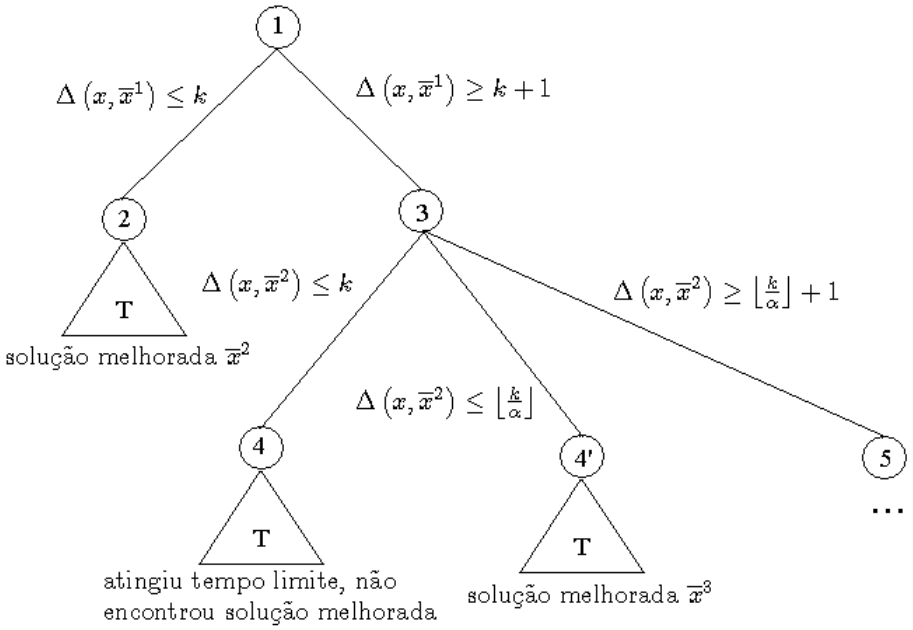

Figura 5.3: Trabalhando com um limite de tempo: caso 2.

3. **Solução Viável Encontrada**: Uma solução de custo melhor do que a ultima solução de referencia foi encontrada, mas o resolvedor de programação linear inteira não foi capaz de provar sua otimalidade. Nesta caso não se pode reverter a ultima restrição de *local branching*. Com o objetivo de continuar cortando a solução de referencia corrente $\overline{x}$, a ultima restrição de *local branching* $\Delta(x,\overline{x}) \leq rhs$ é trocada pela restrição tabu $\Delta(x,\overline{x}) \geq rhs$. A solução de referencia é atualizada e o processo iterado.

4. **Não Encontrou Solução Viável**: Nenhuma solução melhor do que um limite foi encontrada, mas não existe nenhuma garantia de que não exista. Neste caso a ultima restrição de *local branching* deve ser eliminada, e uma nova restrição de *local branching* que defina uma vizinhança menor deve ser colocada.

No algoritmo 4 uma função chamada MIP de três parâmetros é a responsavel por resolver o modelo de programação linear inteira. O primeiro parâmetro é a nova solução encontrada, o segundo é o valor de cutoff e o terceiro é um booleano indicando se se deve parar na primeira solução encontrada ou não. A função MIP tem como retorno o valor da solução encontrada.

Com o objetivo de melhor ilustrar o método de *local branching* para o PAG, será descrito um exemplo. Dada a seguinte instância do PAG com 2 agentes e 6 tarefas:

Capacidade Agente 1: 10.

for *cont = 0; cont < numIteracoes; cont++* **do**
  $cutoff$ =MIP(x,∞,$true$);
  $\overline{x} = x$;
  adiciona ao Modelo a restrição de *local branching* relativa a $\overline{x}$;
  for *;;* **do**
    val = MIP(x,$cutoff$,$false$);
    if *Solução Ótima Encontrada* **then**
      Reverter ultima restrição de *local branching*;
      $\overline{x} = x$;
      Adicionar restrição de *local branching* relativa a $\overline{x}$;
      $cutoff = val$;
    else
      if *Provou Inviabilidade* **then**
        Reverter ultima restrição de *local branching*;
        break;
      else
        if *Solução Viável Encontrada* **then**
          Trocar a ultima restrição de *local branching* $\Delta(x, \overline{x}) \leq rhs$ por $\Delta(x, \overline{x}) \geq rhs$;
          $\overline{x} = x$;
          $cutoff = val$;
        else
          if *Não Encontrou Solução Viável* **then**
            Reduzir folga da ultima restrição de *local branching*;

Algoritmo 4: Algoritmo de *Local Branching*

Custos de Alocar as tarefas ao Agente 1: 1, 0, 0, 1, 0, 0.

Consumo das tarefas no Agente 1: 2, 5, 2, 3, 5, 4.

Capacidade Agente 2: 13.

Custos de Alocar as tarefas ao Agente 2: 0, 3, 1, 0, 1, 2.

Consumo das tarefas no Agente 2: 6, 3, 1, 5, 4, 2.

Esta instância tem o seguinte modelo de programação Inteira:

$$\begin{aligned} \min \quad & 1x_{11} + 0x_{12} + 0x_{13} + 1x_{14} + 0x_{15} + 0x_{16} + \\ & 0x_{21} + 3x_{22} + 1x_{23} + 0x_{24} + 1x_{25} + 1x_{26} \end{aligned}$$

$s.a.$

$$1x_{11} + 1x_{21} = 1$$
$$1x_{12} + 1x_{22} = 1$$
$$1x_{13} + 1x_{23} = 1$$
$$1x_{14} + 1x_{24} = 1$$
$$1x_{15} + 1x_{25} = 1$$

$$1x_{16} + 1x_{26} = 1$$
$$2x_{11} + 5x_{12} + 2x_{13} + 3x_{14} + 5x_{15} + 4x_{16} \leq 10$$
$$6x_{21} + 3x_{22} + 1x_{23} + 5x_{24} + 4x_{25} + 2x_{26} \leq 13$$

Supondo que a solução inicial $\overline{x}^1$ de custo 7 seja a seguinte alocação para as tarefas: 1, 2, 2, 2, 1, 2. Esta solução geraria a restrição de *local branching* que segue:

$$x_{11} + x_{15} + x_{22} + x_{23} + x_{24} + x_{26} \geq 6 - 2$$

Após ser adicionada ao modelo, a restrição acima assegura que no máximo duas tarefas poderão trocar de agente, relativamente a solução $\overline{x}^1$. Então o resolvedor de programação linear inteira é chamado e retorna a solução otimizada $\overline{x}^2 = 1, 1, 2, 2, 2, 2$ com custo 5. Nota-se que de fato apenas 2 tarefas mudaram de agente: a tarefa 2 mudou para o agente 1 e a tarefa 5 mudou para o agente 2. Como o resolvedor de programação linear inteira conseguiu provar a otimalidade do modelo, a restrição de *local branching* deve ser revertida para:

$$x_{11} + x_{15} + x_{22} + x_{23} + x_{24} + x_{26} \leq 6 - 2$$

Uma nova restrição de *local branching* relativa a $\overline{x}^2$ deve ser adicionada ao modelo:

$$x_{11} + x_{12} + x_{23} + x_{24} + x_{25} + x_{16} \geq 6 - 2$$

O resolvedor de programação linear inteira é chamado novamente e retorna a solução otimizada $\overline{x}^3 = 1, 1, 1, 2, 2, 2$, que tem custo 4. Como o resolvedor MIP novamente resolveu a otimalidade a ultima restrição de *local branching* deve ser revertida para:

$$x_{11} + x_{12} + x_{23} + x_{24} + x_{25} + x_{16} \leq 6 - 2$$

E a nova restrição de ramificação relativa a solução $\overline{x}^3$ adicionada:

$$x_{11} + x_{12} + x_{13} + x_{24} + x_{25} + x_{26} \geq 6 - 2$$

O resolvedor de programação linear inteira é então chamado novamente. Como não é possível encontrar uma solução com valor menor do que 4 trocando-se apenas a alocação de 2 tarefas, relativamente a solução $\overline{x}^3$, o resolvedor de programação linear inteira vai responder que o modelo é inviável. Desta maneira a ultima restrição de *local branching* corrente também é revertida:

$$x_{11} + x_{12} + x_{13} + x_{24} + x_{25} + x_{26} \leq 6 - 2$$

Feito isso o modelo tem o seu limite superior ajustado para $+\infty$ e uma nova descida a partir de uma possível solução diversificada é iniciada.

5.2.1 Experiência Computacional

A metaeurística para o PAG baseada no algoritmo de *local branching*, implementado neste trabalho, encontrou soluções com valores próximos aos

das melhores soluções conhecidas e em alguns chegando a encontrar os valores ótimos.

Aplicar o metodo de *local branching* na formulação clássica do PAG foi uma alternativa interessante porque demandou pouco esforço de implementação e pequeno tempo de computação para atingir boas soluções.

A tabela 5.1 apresenta os resultados do algoritmo para as intâncias do PAG classes C, D e E disponíveis na OR-Library[1]. Valores assinalados com o † são sabidos serem ótimos.

A exemplo de Fischetti e Lodi [35], neste trabalho foi usado como resolvedor de programação linear inteira o solver MIP ILOG CPLEX 7.1. A máquina utilizada nestas rodadas foi um PC Pentium IV 2Ghz com 512 MB de memória RAM.

tipo	m	n	LB	melhor conhecida	mínimo	média	máximo	solução encontrada	t. melhor solução	limite de nós
C	5	100	1930	†1931	*1931	1935.0	1937	1/5	0.26	5000
C	5	200	3455	†3456	*3456	3457.8	3460	1/5	12.08	5000
C	10	100	1400	†1402	*1402	1403.0	1404	1/5	2.90	5000
C	10	200	2804	†2806	*2806	2807.0	2808	2/5	135.12	5000
C	20	100	1242	†1243	*1243	1244.4	1246	1/5	2.84	5000
C	20	200	2391	†2391	*2391	2392.2	2393	4/5	154.70	5000
D	5	100	6350	†6353	6357	6360.0	6365	1/5	78.12	10000
D	5	200	12741	12743	12745	12751.2	12755	2/5	457.30	10000
D	10	100	6342	6349	6362	6367.4	6372	2/5	385.81	10000
D	10	200	12426	12433	12440	12450.6	12462	3/5	1993.41	10000
D	20	100	6177	6196	6233	6242.8	6247	5/5	2955.18	10000
D	20	200	12230	12244	12278	12285.8	12291	5/5	12160.20	10000
E	5	100	12673	†12681	*12681	12689.0	12701	1/5	3.43	10000
E	5	200	24927	†24930	*24930	24932.6	24934	1/5	5.40	10000
E	10	100	11568	†11577	*11577	11589.4	11618	6/20	231.85	1000
E	10	200	23302	†23307	*23307	23319.0	23347	2/20	88.19	1000
E	20	100	8431	†8436	8451	8464.4	8535	15/20	1671.21	1000
E	20	200	22377	†22379	22380	22381.2	22382	3/5	1490.81	10000

Tabela 5.1: Resultados do *Local Branching* para o PAG.

[1]URL da OR-Library: http://mscmga.ms.ic.ac.uk/jeb/orlib/gapinfo.html