UMA HEURÍSTICA GRASP PARA O PROBLEMA DO CAIXEIRO VIAJANTE PERIÓDICO

Luciana Brugiolo Gonçalves, Simone de Lima Martins, Luiz Satoru Ochi
Instituto de Computação - Universidade Federal Fluminense
Niterói, Rio de Janeiro, Brasil, 24210-240
{lgoncalves, Simone, satoru}@ic.uff.br

Resumo

Neste artigo apresentamos uma heurística GRASP aplicada ao Problema do Caixeiro Viajante Periódico (PCVP). Este problema pode ser visto como uma generalização do clássico problema do caixeiro viajante (PCV) por estender o período de planejamento de um para m dias. No PCVP cada cidade deve ser visitada um determinado número de vezes e para cada dia do horizonte de planejamento devemos resolver um PCV com as cidades associadas a este dia. Para avaliar o algoritmo aqui proposto, foram utilizadas instâncias disponíveis na literatura para o PCVP. Resultados computacionais mostram que o GRASP proposto apresenta soluções similares aos melhores existentes na literatura, mas exigindo tempos computacionais mais reduzidos.

Palavras Chaves: problema do caixeiro viajante periódico, heurísticas, metaheurística GRASP.

Abstract

In this paper we present a GRASP heuristic applied for the period traveling salesman problem. This problem can be seen as a generalization of the classical traveling salesman problem by extending the planning period from a single day to m days. In the PCVP each city must be visited one determined number of times and for each day of the planning horizon we must solve a PCV with the cities associated to this day. To evaluate the proposed algorithm, test instances available in literature for the PCVP had been used. Computational results show that the proposed GRASP presents similar results to the best ones of literature, but demanding less computational times

Keywords: period traveling salesman problem, heuristics, metaheuristic GRASP.

1. Introdução

O Problema do Caixeiro Viajante Periódico (PCVP) é uma generalização do clássico Problema do Caixeiro Viajante (PCV) onde o horizonte de planejamento é de m dias, e para cada dia devemos resolver um PCV com as cidades selecionadas para serem visitadas neste dia.

Considerando N = {1, 2, ..., n} o conjunto de cidades, cada cidade $i \in N$ requer um número de visitas r_i ($1 \leq r_i \leq m$) que devem ser executadas pelo caixeiro durante o horizonte de planejamento de m dias. Além disso, cada cidade i possui um conjunto de possíveis combinações de dias de visita V(i), onde cada combinação é composta por r_i dias em que a cidade i pode ser visitada. Por exemplo, se uma cidade j requer três visitas durante o período de $m = 5$ dias e existindo duas possíveis combinações, V(j) = {(1, 3, 4); (2, 3, 5)}, então o viajante deve visitar a cidade j nos dias 1, 3 e 4 se a combinação (1, 3, 4) for selecionada ou nos dias 2, 3 e 5, caso a segunda combinação seja escolhida. Desta forma o PCVP deve simultaneamente selecionar uma combinação de visitas para cada uma das cidades de modo que a distância total percorrida no horizonte de planejamento seja minimizada.

O PCVP é um problema de otimização NP-difícil que pode estar presente em várias aplicações incluindo problemas de distribuição de alimentos, entrega/coleta postal e coleta de lixo urbano ([1], [2], [3], [4], [8], [9]).

Algoritmos aproximados já foram propostos para solucionar o PCVP, entre os quais destacamos os trabalho de Chao et al.[2] que propôs um procedimento heurístico que primeiramente constrói uma solução inicial e depois usa uma etapa de refinamento baseado na aproximação *record-to-record* de Dueck [5]. Palleta (2002) propõe em [9] uma heurística que combina um algoritmo construtivo com um procedimento de refinamento de uma solução parcial e em Bertazzi et al. (2004) [1] é apresentado uma versão melhorada do algoritmo proposto em [9] que segundo seus autores, apresentam os melhores resultados heurísticos para as instâncias do PCVP por eles analisadas.

A maioria dos trabalhos relacionados ao PCVP foram na verdade desenvolvidos para uma extensão deste problema que é o Problema de Roteamento de Veículos Periódico (PRVP), que apresenta um maior número de restrições que o PCVP. Dentre eles podemos destacar os artigos de Cordeau et al. [4] que propôs um algoritmo de Busca Tabu para o PRVP utilizando o procedimento GENI proposto por Gendreau [7]. Ochi et al., em 2001 [8], apresentaram um algoritmo evolutivo paralelo para o PRVP obtendo os melhores resultados médios para instâncias do PRVP quando comparando as suas soluções com as obtidas por diferentes heurísticas e metaheurísticas seqüenciais da literatura. O que podemos observar, é que embora para o PRVP existam um número razoável de contribuições, incluindo aí, algumas metaheuristicas evolutivas e busca tabu, o mesmo não ocorre para o seu caso particular PCVP. De fato, de nosso conhecimento não existe, nenhuma contribuição específica para o PCVP usando metaheurísticas. O que encontramos são metaheurísticas desenvolvidas para o PRVP [4, 8].

Neste artigo é proposta uma heurística GRASP para o PCVP que na sua fase de construção, adapta o algoritmo de construção proposto por Bertazzi et al. [1], tornando-o adaptativo, randômico, e guloso. A motivação para o desenvolvimento do GRASP foi devido ao grande esforço despendido pela heurística proposta em [1] que passamos a denotar por H_BPS. A heurística H_BPS possui três parâmetros e para computar as combinações sugeridas pelos autores são necessárias centenas ou milhares de execuções do mesmo dependendo das dimensões da instância, o que a torna inviável para instâncias de grande porte. Este trabalho apresenta na seção 2, a heurística H_BPS de Bertazzi et al. [1] e o algoritmo GRASP proposto neste artigo é descrito na seção 3. Na seção 4 é apresentada a comparação entre os resultados computacionais obtidos pelos dois algoritmos. Considerações finais são apresentadas na seção 5.

2. Algoritmo heurístico H_BPS

A heurística proposta por Bertazzi et al. (2004) [1], aqui denominada H_BPS descrita na figura 1, é uma combinação de um processo de construção iterativo com um procedimento de refinamento.

H_BPS (p_1, p_2, f):
1. Inicializar m rotas, contendo somente a cidade origem
2. Associar a cidade f à primeira combinação de dias de visita $Vf \in V(f)$
3. Inserir f na rota dos dias da combinação Vf
4. $Z = \{f\}$ conjunto das cidades já processadas
5. Para z = 2 ... número_cidades
6. Executar *City Processing* (Z).
7. If |Z| é múltiplo de p_1, executar Procedimento *Improved*(p_2)
8. Se solução inviável, executar *Feasibility*
9. Executar procedimento $Improved_2(p_2)$, mantendo a viabilidade da rota.

Figura 1. Algoritmo H_BPS[1]

Na linha 1, são inicializadas m rotas, uma para cada dia do período de planejamento, contendo apenas a cidade origem. Na linha 2, a cidade f passada como parâmetro é associada à primeira de suas combinações $Vf \in V(f)$. Em seguida, na linha 3, a cidade é inserida em todas as rotas correspondentes aos dias da combinação associada a ela e, depois disso, inserida no conjunto de cidades já processadas Z, como se pode ver na linha 4. Após a etapa de inicialização (linhas 1-4), realiza-se a etapa de construção através do procedimento *City Processing* acionado na linha 6. Durante esta etapa, uma cidade ainda não processada é selecionada e a ela é associada uma combinação. Para cada dia da combinação é feita a inserção da cidade na rota correspondente a este dia. Sendo assim, a partir do início do processo de construção existem rotas parciais para cada dia, ou seja, elas são construídas em paralelo. Durante esse processo ocorrem interrupções para que um aperfeiçoamento seja executado (*Improved*) almejando diminuir o custo das rotas de cada dia do período, como mostrado na linha 7.

Para finalizar o processo de construção, na linha 8, realiza-se a validação da solução. Para que a solução seja viável as rotas de cada dia do período de planejamento não podem estar vazias, ou seja, cada rota deve ter pelo menos uma cidade a ser visitada pelo caixeiro. Caso a solução construída seja inválida, é executado o procedimento denominado *Feasibility* para viabilizar a solução gerada.

Na linha 9, aplica-se à solução viável construída um outro procedimento de aperfeiçoamento muito simular ao procedimento *Improved (linha 7),* mas que trabalha com algumas restrições para que a viabilidade da solução não seja alterada.

A heurística H_BPS possui três parâmetros (p_1, p_2 e f) que afetam diretamente o funcionamento e o resultado do algoritmo. O primeiro deles, p_1, determina os instantes em que serão feitas as interrupções para a execução do procedimento de aperfeiçoamento (*Improved*) durante o processo de construção, interrupções estas que ocorrem toda vez que o número de cidades processadas for múltiplo de p_1. O parâmetro p_2 afeta diretamente as funções de aperfeiçoamento onde são feitas tentativas de melhorar a solução corrente através da troca das combinações das cidades já processadas. Retiram-se p_2 cidades da solução corrente, e então é feita uma tentativa de melhorar o resultado através da seleção de outro conjunto de combinações que determinam em que rotas as cidades serão re-inseridas. Como já visto, o último parâmetro, f, determina qual é a primeira cidade a ser processada. Este parâmetro é importante por afetar diretamente a ordem de seleção das cidades na etapa de construção. Para que seja possível descrever os procedimentos *City Processing*, *Improved* e *Feasibility* utilizados pela heurística H_BPS, é necessário apresentar as regras de inserção e remoção utilizadas por eles.

Regra para inserção:

A regra de inserção é utilizada para colocar uma cidade *i* em uma determinada rota, que é um processo necessário aos procedimentos de construção e aperfeiçoamento do H_BPS.

Se a rota onde a cidade *i* vai ser inserida estiver vazia, então ela deve ser ligada diretamente com a cidade origem, ou seja, é criado um laço conectando a cidade origem e a cidade *i*. Mas, se a rota já possui alguma cidade, então é necessário verificar em que posição a cidade causa menor acréscimo ao custo da rota. Assim para cada par de arcos, ($cidade_x$, $sucessor_x$) e ($cidade_y$, $sucessor_y$), os métodos mostrados a seguir são aplicados.

Método 1: Remover os arcos ($cidade_x$, $sucessor_x$) e ($cidade_y$, $sucessor_y$), e inserir os arcos ($cidade_x$, $cidade_i$), ($cidade_i$, $cidade_y$) e ($sucessor_x$, $sucessor_y$).

Método 2: Remover os arcos ($cidade_x$, $sucessor_x$) e ($cidade_y$, $sucessor_y$), e insere os arcos ($sucessor_x$, $cidade_i$), ($cidade_i$, $sucessor_y$) e ($cidade_x$, $cidade_y$).

Depois de aplicar cada método é necessário inverter o caminho entre as cidades $cidades_y$ e $sucessor_x$. São examinados todos os pares de arcos e a rota corrente será aquela que obtiver o menor custo. O custo de inserção de uma cidade em uma rota é igual ao somatório dos custos dos arcos inseridos menos o somatório dos custos dos arcos removidos.

Regra de remoção:

Para se remover uma cidade *r* da rota de um determinado dia do período é aplicada o seguinte procedimento. Primeiramente o arco incidente na cidade *r* é removido. Então, cada par de arcos remanescentes é apagado e a árvore resultante é re-conectada de todas as possíveis formas para formar novas rotas. Depois de examinar todas as possibilidades, a menor rota entre todas as obtidas torna-se a rota corrente. Este método é baseado na heurística *3-optimal*.

Procedimento *City Processing*

Este procedimento, cujo pseudo-código é apresentado na figura 2, é aplicado para selecionar a próxima cidade que será processada, associar a ela uma combinação e inseri-la nas rotas dos dias da combinação escolhida.

City Processing (Z)
1. Calcular o custo de seleção de cada cidade ainda não processada, $cs(i) = r_i \times \min_{j \in Z} d_{ij}$
2. Selecionar a cidade *s* com custo de seleção máximo
3. Calcular o custo de inserir a cidade *s* em cada dia do período de planejamento
4. Calcular o custo de inserção de cada combinação de V(*s*)
5. Associar à cidade *s* a combinação $Vs \in V(s)$ com custo de inserção mínimo
6. Inserir *s* em todos os dias da combinação *Vs*
7. Colocar *s* no conjunto das cidades já processadas, $Z = Z \cup \{s\}$.

Figura 2. Procedimento City Processing utilizado por do H_BPS

Nas linhas 1 e 2, seleciona-se uma cidade *s* ainda não processada que apresente o maior custo de seleção, sendo o custo de seleção de *s* igual ao produto $r_s \times \min_{j \in Z} d_{sj,}$onde d_{sj} é à distância entre as cidades *s* e *j*. Após esta etapa é necessário encontrar a combinação *Vs* com menor custo de inserção definido pelo somatório dos custos de se inserir a cidade em cada dia da combinação e associá-la a *s*. Na linha 3 calcula-se o custo de inserir a cidade em cada dia do período de planejamento e, na linha 4, é determinado o custo de inserção de cada combinação de *s*. Para finalizar o procedimento, associa-se

a combinação de custo mínimo à cidade *s*, insere-se *s* nas rotas da combinação selecionada e adiciona-se *s* ao conjunto Z das cidade já processadas, como mostrado nas linhas 5 a 7.

Procedimento *Improved*

O procedimento *Improved* que está descrito na figura 3 é executado quando ocorre uma condição de parada durante a etapa de construção do algoritmo.

```
Procedimento Improved (p2)
 1.      - Para cada u = 1, 2, ...., ⌈|Z|/p2⌉:
 2.              Para cada  j = 1 + (u -1)p2, up2:
 3.                      Se j ≤ |Z|, então selecionar a j-ésima cidade do conjunto Z;
 4.                          senão selecionar a mod(j, |Z|)-ésima cidade.
 5.                      Sendo s a cidade selecionada,
 6.                      Remover a cidade s de todas as rotas em que está inserida utilizando a regra de
 7.                      remoção[2].
 8.              Para cada  j = 1 + (u -1)p2, up2:
 9.                      Se j ≤ |Z|, então selecionar a j-ésima cidade do conjunto Z;
10.                          senão selecionar a mod(j, |Z|)-ésima cidade.
11.                      Sendo s a cidade selecionada,
12.                      Selecionar uma nova combinação Vs de visitas para s.
13.                      Utilizar a regra de inserção[2] para colocar a cidade s nas rotas dos dias de Vs.
14.             Se o custo das novas rotas é menor que a das antigas, então atualizar a solução corrente.
```

Figura 3. Procedimento Improved utilizado para aperfeiçoamento da solução no H_BPS

Na tentativa de melhorar a solução corrente, a cada iteração p_2 cidades já processadas são removidas das rotas em que estão inseridas (linhas 2 a 7). Depois disso, é selecionada uma nova combinação para cada cidade que foi removida e, em seguida, elas são inseridas nas rotas correspondentes às novas combinações associadas, como mostrado nas linhas 8 a 13.

Na linha 14 é verificado se as alterações realizadas resultaram em redução no custo da solução, caso afirmativo o novo conjunto de combinações é adotado como solução corrente e as rotas são atualizadas (figura 3).

Procedimento *Feasibility*

Após o processamento de todas as cidades, se a solução obtida contém alguma rota vazia, executa-se o procedimento *Feasibility* para viabilizar a solução. Para isso, retira-se uma cidade que pertença a rotas com pelo menos uma outra cidade e seleciona-se para a cidade removida uma combinação que contemple o maior número de rotas vazias. A cidade cuja troca causar menor aumento no custo total da solução será selecionada para ter sua combinação modificada, sendo removida das rotas da combinação antiga e inserida nas rotas da nova combinação. Desta forma a cada etapa o número de rotas vazias diminui, sendo necessário executar este procedimento enquanto houver rota vazia. Assim é finalizada a etapa de construção da heurística H_BPS garantindo que a solução gerada é viável.

Procedimento *Improved$_2$*

O último procedimento executado pelo algoritmo H_BPS [1], é o procedimento *Improved$_2$* muito semelhante ao *Improved*. A diferença é que no *Improved$_2$* existe uma restrição que garante que as alterações não afetem a viabilidade da solução. Apenas as cidades que não sejam únicas nas rotas que pertençam podem ser retiradas quando ocorre uma tentativa de troca de combinação. Assim, não é possível remover cidades que estejam sozinhas em uma determinada rota.

Como apresentado em Bertazzi et al.[1], os resultados desta heurística são os melhores conhecidos para um grupo de instâncias do PCVP, mas analisando as instâncias nas quais os testes foram feitos, é possível notar que o custo computacional para alcançá-los pode ser considerável. Mesmo que cada execução da heurística demore em média alguns segundos, para se testar as possíveis combinações de parâmetros sugeridas pelos autores da heurística (p_1 = 1, 2, 3, ..., 20; p_2 = 1, 2, 3, ..., 20 e f = 1, ..., n onde n é o número de cidades da instância), o tempo computacional aumenta consideravelmente. O número médio de cidades nas instâncias testadas é 119, logo, há em média 20 x 20 x 119 = 47.600 possibilidades de combinações de parâmetros para cada instância desta magnitude.

Estes dois fatos, o primeiro a de que a heurística H_BPS obtém as melhores soluções aproximadas para o PCVP, e a segunda a de que o H_BPS para alcançar estas soluções, necessita de centenas e milhares de execuções para cada instância onerando significativamente o seu tempo computacional final, incentivaram o desenvolvimento de uma metaheurística GRASP (denominado G_GMO) que pudesse alcançar resultados similares aos do H_BPS mas exigindo tempos computacionais bem menores.

3. ALGORITMO G_GMO

A heurística GRASP é basicamente um processo iterativo do tipo *multi-start*, onde cada iteração consiste de duas fases: a construção de uma solução inicial de forma gulosa, aleatória e adaptativa e um procedimento de busca local que tem o objetivo de aprimorar a solução gerada na primeira fase. Cada iteração GRASP trabalha de forma independente e o resultado final será então a melhor solução dentre todas as iterações [6, 10, 11].

A fase de construção do GRASP proposto (*aqui denominado G_GMO (**G**RASP proposto por **G**onçalves, **M**artins e **O**chi))* é um processo iterativo onde, em cada iteração desta etapa, cada cidade que ainda não pertence ao conjunto das cidades processadas Z é avaliada por uma função gulosa que estima o custo de se inserir a cidade na solução parcial. As cidades são ordenadas em ordem decrescente segundo os valores estimados e colocadas em uma lista chamada Lista de Candidatos (LC). A próxima cidade a fazer parte da solução será escolhida aleatoriamente da sub lista LRC composta pelos [α x (|N|-|Z|)] elementos melhor avaliados de LC, sendo que N representa o conjunto de todas as cidades envolvidas no processo. A fase de construção termina quando uma solução válida é gerada.

A solução obtida não é necessariamente um ótimo local, então geralmente um procedimento de busca local é executado almejando um aprimoramento de cada solução construída. Este processo de procura sucessiva por uma solução melhor na região é executado até que nenhuma solução melhor seja encontrada.

Fase de Construção do G_GMO

Na etapa de construção do G_GMO, as cidades são processadas utilizando essencialmente o procedimento *Processing City* já descrito anteriormente. Contudo, foi criado um processo diferente para selecionar as cidades. O procedimento construtivo pode ser visto na figura 4.

Como podemos ver nas linhas 1 a 3, a inicialização é a mesma do algoritmo H_BPS apresentado na figura 1, mas como o GRASP é do tipo *multi-start* este processo foi agregado ao procedimento de construção.

Na linha 4 é calculado o valor da função gulosa para avaliação de cada cidade $i \in$ N-Z ,ainda não inserida na solução. Esta função consiste do custo de inserção $ci(i) = r_i \times \min_{j \in Z} d_{ij}$.

```
Procedimento Construtivo(α, f, p1, p2, p)
 1.    Associar a cidade f a primeira combinação de dias de visita  Vf ∈ V(f)
 2.    Inserir f na rota dos dias da combinação Vf
 3.    Z = {f} conjunto das cidades já processadas
 4.    Calcular o custo de seleção de cada cidade ainda não processada
 5.    Para z = 2 .. n,
 6.        Ordenar as cidades na ordem não crescente do custo de seleção
 7.        Construir a LRC com ( (|N| - |Z|) x [α/100] )  cidades de maior custo de seleção
 8.        Escolher aleatoriamente uma cidade s da LRC
 9.        Calcular o custo de inserção de cada combinação de V(s)
10.        Associar à cidade s a combinação Vs com custo de inserção mínimo
12.        Inserir s em todos os dias da combinação
13.        Colocar s no conjunto das cidades já processadas, Z = Z ∪ {s}
14.        Atualizar o custo de seleção de cada cidade ainda não processada
15.        If |Z| é múltiplo de (numero_cidades x [p1\100]) e iteração < p, executar Procedimento
16.        Improved(p2)
17.        Se solução inviável, executar Feasibility
18.    Finalizar
```

Figura 4. Procedimento de Construção do G_GMO

Para seleção da próxima cidade a ser inserida na solução, é necessário ordenar as cidades que ainda não foram processadas segundo o custo da função gulosa e selecionar uma fração α delas para fazerem parte da LRC, como descrito nas linhas 6 e 7. Na linha 8, temos a seleção aleatória da próxima cidade que será processada.

Após sortear uma das cidades da LRC, o procedimento continua da mesma forma que na heurística H_BPS. Calcula-se o custo de inserção de cada combinação, associa-se à cidade aquela combinação de menor custo de inserção, insere-a nas rotas de cada dia da combinação e a coloca no conjunto Z (linhas 9 a 13). Após o processamento de uma cidade é preciso atualizar o custo de seleção das demais cidades candidatas como vemos na linha 14 do algoritmo.

As interrupções para execução do procedimento *Improved*, um dos procedimentos mais dispendiosos do algoritmo, ocorre nas p primeiras iterações do GRASP, e sempre que p_1 x |N|/100 cidades tiverem sido processadas, como mostrado na linha 15.

A única alteração do procedimento *Improved* ocorre na forma de tratamento do parâmetro p_2. Na heurística H_BPS, p_2 é um parâmetro fixo independente do número de cidades da instância, enquanto que no GRASP, p_2 corresponde a uma porcentagem do número de cidades já processadas até o momento. Deste modo, sempre um número razoável de cidades é retirado independente do instante em que ocorreu a parada para a execução do procedimento. Os demais passos do procedimento são idênticos ao algoritmo heurístico.

Na linha 17, após a inserção de todas as cidades nas rotas, continua sendo necessário tratar a viabilidade. Caso exista alguma rota vazia no período, é necessário executar o procedimento que torna viável a solução gerada (Procedimento *Feasibility*).

Fase de Busca Local do G_GMO

Na segunda etapa do G_GMO é utilizado o procedimento $Improved_2$ da mesma forma que aplicado na heurística H_BPS, sempre tendo o cuidado de não tornar a solução inválida. Da mesma

forma que no procedimento *Improved*, no procedimento $Improved_2$ as cidades são retiradas em grupos de p_2 x |N|/100.

O algoritmo apresentado na figura 5 resume o funcionamento do GRASP proposto. O parâmetro α não tem um valor fixo durante toda a execução do algoritmo, como é feito na versão básica do GRASP. Existe um conjunto com alguns valores pré-definidos que são assumidos durante o processamento. As iterações são divididas de forma que o algoritmo execute um mesmo número de vezes com cada valor do conjunto. Os valores escolhidos para preencher o vetor de α foram escolhidos depois de um estudo que determinou o nível de aleatoriedade desejado para o algoritmo.

Na linha 1, executa-se o controle do número máximo de iterações a serem realizadas *Max_it*. Antes de executar o procedimento de construção é necessário escolher o valor de α que será utilizado durante a iteração (linha 2). O procedimento de construção é executado na linha 3 e, com uma solução válida construída, é executado o procedimento de busca local na linha 4.

Como já foi mencionado, pelo fato das iterações do GRASP serem independentes, a solução final do algoritmo é a melhor solução entre as soluções encontradas em cada iteração, cuja verificação é feita nas linhas 5 e 6.

```
GRASP G_GMP ( p1, p2, f, Max_it)
1.      Para iteração = 1 .. Max_it,
2.          Selecionar α do vetor_de_alfas;
3.          Executar procedimento Construtivo (α, f, p1, p2, p)
4.          Improved2 (p2)
5.          Se custo (solução_atual ) < custo (melhor_solução), então
6.                  Melhor_solução ← solução_atual;
```

Figura 5. Algoritmo G_GMO

Deve-se ressaltar que a alteração na forma de tratamento dos parâmetros p_1 e p_2 foi feita para uma melhor adaptação ao tamanho de cada instância, visto que na heurística H_BPS um mesmo valor fixo é utilizado para instâncias de tamanho bem variado. Por exemplo, uma instância com 50 cidades e outra com 200 reagem de forma diferente se suas cidades forem retiradas em grupos de 15. Para a primeira instância este valor pode ser grande demais, já para a segunda ele pode ser um valor razoável. Mas, para as mesmas instâncias, se as cidades são retiradas em grupos de 15% do número de cidades espera-se um aproveitamento semelhante em ambas às instâncias.

Depois de apresentar o GRASP G_GMO e a heurística H_BPS, na próxima seção são comparados os custos das soluções e o tempo de CPU para cada um dos algoritmos.

4. Resultados Computacionais

Para avaliar os dois algoritmos foram utilizadas trinta e três instancias da literatura. As instâncias de 1 a 10 foram introduzidas por Christofides e Beasley [3], as de 11 a 23 por Chao et al.[2] e as demais foram propostas por Cordeau et al. [4]. Neste trabalho além dos resultados obtidos pela implementação original do H_BPS, foi feita uma nova implementação deste algoritmo (H_BPS*) neste trabalho, com o intuito de se obter mais informações do H_BSP.

Para implementação da heurística H_BPS* e da metaheurística G_GMO foi utilizada a linguagem de programação C, compilada com GCC versão 3.2.2, executados em máquinas Pentium® 4 2.6GHz com 512Mbytes de RAM. Nas seções seguintes são apresentados os custos das soluções e o tempo de CPU para todas as instâncias.

4.1 Avaliação da qualidade das Soluções

Na tabela 1 são apresentadas as soluções encontradas por cada um dos algoritmos. Na coluna 1 temos a identificação da instância, na coluna 2 e 3 são fornecidos respectivamente o número de cidades e o horizonte do período de planejamento.

Na coluna 4 são descritos resultados do H_BPS implementado por Bertazzi et al.[1] que executou a heurística H_PBS utilizando todas as possíveis combinações dos parâmetros p_1, p_2 e f, p_1 = 1, 2, 3, ..., 20; p_2 = 1, 2, 3, ..., 20 e f = 1, ..., n onde n é o número de cidades da instância. Na coluna 5, apresenta-se o melhor valor obtido pelo H_BPS*, onde o H_BPS foi executado para somente quatro das combinações destes parâmetros, demandando por conseguinte menor tempo computacional. Esta versão aqui denominada por H_BPS* é a nossa implementação do H_BPS. O objetivo inicial do H_BPS* é avaliar o tempo de CPU deste em uma execução usando valores fixos para os três parâmetros de entrada bem como avaliar o seu desempenho no caso de se executar apenas parte das combinações dos parâmetros existentes. Na coluna 6 (%) mostramos as diferenças entre os resultados do H_BPS e H_BPS* (estes valores indicam o quanto pior foi o desempenho do H_BPS* em relação ao H_BPS). Finalmente nas demais colunas apresentamos os resultados do G_GMO. Como o GRASP possui um componente aleatório, o G_GMO foi executado três vezes com a seguinte configuração dos parâmetros: número máximo de iterações = 250, p = 20, p_1 = 10 e p_2 = 3 e $f = \max_{i \in I} d_{0i}$. O vetor com os valores de α foi preenchido com 2, 5, 10, 15 e 20. Na coluna 7, é apresentada a média dos resultados obtidos em todas as execuções, e na coluna 8 temos a diferença entre esse resultado e o apresentado na coluna 4 (H_BPS). Já na coluna 9 podemos ver a melhor entre todas as soluções encontradas pelo GRASP, e na linha 10 temos a diferença dessa solução em relação ao valor apresentado na coluna 4.

Id	Cidades	Período	H_BPS	H_BPS*		G_GMO			
						Média		*Best*	
				Custo	%	Custo	%	Custo	%
1	50	2	**432,10**	436,07	0,92	432,14	0,01	**432,10**	0,00
2	50	5	**1105,81**	1118,52	1,15	1106,52	0,06	1106,24	0,04
3	50	5	**466,71**	469,22	0,54	**466,71**	0,00	**466,71**	0,00
4	75	2	**549,05**	555,44	1,16	551,22	0,39	550,18	0,21
5	75	5	**1382,33**	1383,34	0,07	1385,85	0,25	1383,34	0,07
6	75	10	**643,50**	646,93	0,53	643,56	0,01	**643,50**	0,00
7	100	2	**643,80**	658,93	2,35	643,80	0,00	**643,80**	0,00
8	100	5	**1613,80**	1627,65	0,86	1619,07	0,33	1616,42	0,16
9	100	8	**722,69**	731,86	1,27	723,61	0,13	723,03	0,05
10	100	5	**1237,77**	1251,58	1,12	1244,15	0,52	1238,34	0,05
11	65	4	**490,97**	**490,97**	0,00	**490,97**	0,00	**490,97**	0,00
12	87	4	**664,10**	**664,10**	0,00	**664,10**	0,00	**664,10**	0,00
13	109	4	**830,80**	**830,80**	0,00	**830,80**	0,00	**830,80**	0,00
14	131	4	**994,60**	**994,60**	0,00	**994,60**	0,00	**994,60**	0,00
15	153	4	**1157,07**	**1157,07**	0,00	**1157,07**	0,00	**1157,07**	0,00
16	48	4	**660,12**	**660,12**	0,00	**660,12**	0,00	**660,12**	0,00
17	66	4	**776,43**	**776,43**	0,00	**776,43**	0,00	**776,43**	0,00
18	84	4	**873,73**	876,44	0,31	**873,73**	0,00	**873,73**	0,00
19	102	4	**958,51**	**958,51**	0,00	**958,51**	0,00	**958,51**	0,00
20	120	4	**1033,58**	**1033,58**	0,00	**1033,58**	0,00	**1033,58**	0,00
21	77	4	**1375,07**	**1375,07**	0,00	**1375,07**	0,00	**1375,07**	0,00
22	154	4	**4312,31**	4339,88	0,64	**4312,31**	0,00	**4312,31**	0,00
23	231	4	**8308,51**	8442,01	1,61	8340,06	0,38	**8308,51**	0,00
24	48	4	**2064,84**	**2064,84**	0,00	**2064,84**	0,00	**2064,84**	0,00
25	96	4	**3207,44**	3241,07	1,05	3218,90	0,36	3218,35	0,34
26	144	4	**4030,54**	4113,24	2,05	4079,57	1,22	4065,07	0,86
27	192	4	**4558,94**	4594,40	0,78	4585,90	0,59	4580,23	0,47
28	240	4	**4628,89**	4683,21	1,17	4676,53	1,03	4658,67	0,64
29	288	4	**5534,94**	5618,72	1,51	5584,76	0,90	5574,32	0,71
30	72	6	**4435,39**	4471,04	0,80	4454,93	0,44	4451,96	0,37
31	144	6	**5376,11**	5414,42	0,71	5396,83	0,39	5392,44	0,30
32	216	6	**7282,39**	7345,70	0,87	7326,97	0,61	7324,80	0,58
33	288	6	**8280,07**	8389,93	1,33	8339,18	0,71	8331,03	0,62
Média Final					0,69		0,25		0,17

Tabela 1. Comparação entre os custos das soluções.

Na tabela 1, destacado em negrito temos os melhores resultados. A diferença de 0,69% (última linha) que podemos verificar entre a média final dos resultados H_BPS e H_BPS* se deve ao fato de que o número de combinações de parâmetros testados pela nossa implementação (H_BPS*) ser muito menor do que os da implementação da literatura (H_BPS).

Podemos notar também que as soluções conseguidas pelo GRASP G_GMO, quando não conseguem empatar com o da literatura, ficam muito pouco distante dela. A diferença média de 0,17% torna-se pequena quando se considera o esforço computacional elevado da heurística H_BPS [1]. A seguir são apresentados os tempos de CPU das implementações.

4.2. Avaliação dos tempos de execução

Na tabela 2 são apresentados os tempos de CPU em segundos da heurística H_BPS, H_BPS* e do GRASP G_GMO. Nas colunas 1 a 3 temos respectivamente o identificador, o número de cidades e o período de planejamento das instâncias. Na coluna 4 ($tempo_0$), temos o tempo médio utilizado para uma única execução da heurística H_BPS, tempo este conseguido por nossa implementação da referida heurística. Os autores da H_BPS [1] não apresentaram os tempos computacionais, tornando impossível a comparação de tempos com a implementação deles. Na tabela 2, as colunas 5 e 6 mostram respectivamente os tempos totais do H_BPS* e H_BPS em cada instância da literatura. Os tempos do H_BPS foram obtidas da seguinte forma: de posse do tempo exigido em uma execução do H_BPS* (para uma combinação dos parâmetros – coluna 4), multiplicamos este tempo pelo número de combinações existentes destes parâmetros, pois os autores de H_BPS [1] executaram o algoritmo para todas possíveis combinaçãoes. Finalmente a coluna 7, apresenta os tempos de CPU do G_GMO. Na coluna t-G_GMO temos a *média do tempo* gasto pelo GRASP para uma execução com 250 iterações.

Analisando os resultados apresentados na tabela 2, podemos concluir que a heurística H_BPS pode se tornar inviável para o tratamento de instâncias muito grandes.

Se considerarmos que é necessário computar cada possibilidade de combinação de parâmetros para obter os resultados do H_BPS de Bertazzi et al. [1], teremos um tempo de execução estimado igual a 20 x 20 x (número_cidades) x ($tempo_0$) apresentado na coluna 6 ($tempo_2$). Supondo que o número de combinações pudesse ser reduzido através da fixação do parâmetro f como a cidade mais longe da origem ($f = \max_{i \in I} d_{0i}$.), cuja utilização mostrou uma tendência a prover melhores resultados, ainda assim o tempo de CPU seria muito grande, mesmo o número de combinações (20 x 20) sendo reduzido drasticamente, como apresentado na coluna 5 (t-H_BPS).

Por exemplo, para a instância 1, que possui 50 cidades, o número de combinações para $p_1 = 1..20$, $p_2 = 1..20$ e $f = 1..50$ seria 20.000 e, como uma execução dura em média 0,41 segundos, então o tempo estimado para executar todas as possibilidades seria 8.200 segundos. Fixando $f = \max_{i \in I} d_{0i}$, o número de combinações cairia para 400 possibilidades, mas o tempo de CPU continuaria extremamente alto se comparado com o GRASP, (164 para H_BSP* e 30,19 para o GRASP).

Instância			Tempos H_BSP			t-G_GMO
Id	Cidades	Período	$Tempo_0$	t-H_BPS*	t-H_BPS	
1	50	2	0,41	164,00	8.200,00	30,19
2	50	5	0,29	116,00	5.800,00	12,84
3	50	5	0,54	216,00	10.800,00	63,72
4	75	2	1,36	544,00	40.800,00	97,72
5	75	5	0,85	340,00	25.500,00	35,75
6	75	10	2,45	980,00	73.500,00	385,02
7	100	2	2,63	1.052,00	105.200,00	238,07
8	100	5	1,56	624,00	62.400,00	82,43
9	100	8	4,90	1.960,00	196.000,00	791,67
10	100	5	2,52	1.008,00	100.800,00	195,58
11	65	4	0,25	100,00	6.500,00	11,32
12	87	4	0,56	224,00	19.488,00	26,55
13	109	4	1,10	440,00	47.960,00	50,98
14	131	4	1,94	776,00	101.656,00	92,17
15	153	4	3,07	1.228,00	187.884,00	144,88
16	48	4	0,29	116,00	5.568,00	24,89
17	66	4	0,68	272,00	17.952,00	63,44
18	84	4	1,37	548,00	46.032,00	126,73
19	102	4	2,59	1.036,00	105.672,00	226,98
20	120	4	3,84	1.536,00	184.320,00	371,88
21	77	4	0,49	196,00	15.092,00	20,73
22	154	4	3,38	1.352,00	208.208,00	164,30
23	231	4	10,14	4.056,00	936.936,00	567,88
24	48	4	0,29	116,00	5.568,00	12,28
25	96	4	2,38	952,00	91.392,00	94,16
26	144	4	6,86	2.744,00	395.136,00	318,85
27	192	4	16,87	6.748,00	1.295.616,00	751,24
28	240	4	26,14	10.456,00	2.509.440,00	1446,92
29	288	4	36,27	14.508,00	4.178.304,00	2443,80
30	72	6	1,48	592,00	42.624,00	61,21
31	144	6	10,37	4.148,00	597.312,00	466,88
32	216	6	18,40	7.360,00	1.589.760,00	1588,82
33	288	6	38,35	15.340,00	4.417.920,00	3670,89
Média			6,20	2480,24	534404,24	444,87

Tabela 2. Comparação tempo de CPU estimado.

Podemos notar pela tabela 2 que o esforço computacional da H_BPS é muito maior em tempos computacionais exigidos se comparado com o do GRASP. O G_GMO despende um tempo muito menor para obter resultados muito próximos aos obtidos por H_BPS de Bertazzi et al.[1].

Após a análise dos resultados, pode-se verificar uma outra característica importante e positiva do nosso G_GMO, a de que a qualidade da solução do G_GMO não está ligada diretamente ao tamanho das instâncias (número de cidades). Sendo assim, ela pode ser aplicada a problemas de dimensões elevadas sem perda da qualidade da sua solução final.

5. Considerações Finais

Neste artigo propomos um algoritmo GRASP para a solução aproximada do problema do caixeiro viajante periódico. Como foi possível mostrar, o G_GMO apresenta soluções similares aos do H_BPS mas o esforço computacional da metaheurística G_GMO é muito menor que o da heurística proposta por Bertazzi et al. em 2004 que apresenta os melhores resultados aproximados da literatura.

Como sugestões de trabalhos futuros, podemos agregar módulos de religamento e/ou mineração de dados para fazer uma busca local mais eficiente, ou mesmo, desenvolver módulos de construção e busca local alternativos. Uma outra possibilidade é o desenvolvimento de outras metaheurísticas tais como Busca Tabu, Algoritmos Genéticos, *Simulated Annealing* dentre outras para o PCVP.

Agradecimentos

Ao CNPq e CAPES que financiaram parcialmente este trabalho

6. Referências

[1] Bertazzi L., Paletta G., Speranza M. An improved heuristic for the period traveling salesman problem. Computers & Operational Research 31:1215-1222, 2004.

[2] Chao I-M., Golden BL., Wasil EA. A new heuristic for the period traveling salesman problem. Computers and Operations Research 22:553-565, 1995.

[3] Christofides N., Beasley JE. The period routing problem. Networks 1984; 14: 237-256.

[4] Cordeau J., Gendreau M., Laporte G. A Tabu Search Heuristic for Period and Multi-Depot Vehicle Routing Problems. Networks 30:105-119, 1997.

[5] Dueck G. New optimization heuristic, the gread deluge algorithm and the record-to-record travel. Scientific Center Technical Report, IBM Alemã 1990.

[6] Festa, P. , Resende, M. G. C. An annotated bibliography of GRASP, European Journal of Operational Research, 2004

[7] Gendreau M., Hertz A., Laporte G. New insertion and post optimization procedures for the traveling salesman problem. Operations Research 40: 1086-1094, 1992.

[8] Ochi, L. S., Vianna, D. S., Drummond, L. M. A., An asynchronous parallel metaheuristic for the Period Vehicle Routing Problem. In Future Generations on Computer Systems. ELSEVIER, v. 17, 379-386, 2001.

[9] Paletta G. The period traveling salesman problem: a new heuristic algorithm. Computers & Operational Research 29:1345-1352, 2002.

[10] Resende, M. G. C., and Ribeiro, C. C., Greedy randomized adaptive search procedures, in Handbook of Metaheuristics, F. Glover and G. Kochenberger, eds., Kluwer Academic Publishers, 219-249, 2003

[11] Silva G., Ochi L.S., Martins S. L. Experimental Comparison of Greedy Randomized Adaptive Search Procedures for the Maximum Diversity Problem. Experimental and Efficient Algorithms - WEA 2004; LNCS 3059, 498-512, 2004.