UMA HEURÍSTICA GRASP PARA O PROBLEMA ESTENDIDO DE SEQUENCIAMENTO DE CARROS

Lucas Middeldorf Rizzo
Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627 - Pampulha - Belo Horizonte - MG
CEP 31270-901 - Fone: +55 (31) 3409.5000
rizzo@dcc.ufmg.com

Sebastián Urrutia
Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627 - Pampulha - Belo Horizonte - MG
CEP 31270-901 - Fone: +55 (31) 3409.5000
surrutia@dcc.ufmg.com

RESUMO

Este artigo descreve uma heurística GRASP para o recentemente introduzido Problema Estendido de Sequenciamento de Carros. Uma heurística construtiva é desenvolvida com base em uma heurística para o Problema Clássico de Sequenciamento de Carros. O procedimento de busca local utiliza uma vizinhança simples e de fácil avaliação. Resultados computacionais sobre as instâncias da biblioteca CSPLIB verificam a eficiência do método em comparação com a literatura.

PALAVARAS CHAVE. Heurísticas, Otimização Combinatória, GRASP.

Área principal: Heurísticas

ABSTRACT

This paper describes a GRASP heuristic to the recently introduced Extended Car Sequencing Problem. A constructive heuristic is developed based on an heuristic for the classical Car Sequencing Problem. The local search procedure uses a very simple neighborhood easy to be evaluated. Computational results on instances from the CSPLib´s library verify the efficiency of the method in comparison with the literature.

KEYWORDS. Heuristics, Combinatorial Optimization, GRASP.

Main area: Heuristics

1. Introdução

O problema de sequenciamento de carros (CSP) envolve o agendamento de diferentes tipos de carros ao longo de uma linha de montagem para instalação de algumas opções como ar-condicionado e teto-solar. Cada opção é instalada por uma estação, a qual é capaz de simultaneamente lidar com uma quantidade limitada de carros. O problema clássico CSP [4] considera um conjunto de carros a serem produzidos, cada um deles com um conjunto de opções a serem instaladas, e consiste em encontrar uma sequência de carros que não viole as limitações de nenhuma estação em qualquer parte da sequência. O problema é NP - Completo segundo [3].

Este artigo lida com uma versão estendida do CSP (xCSP) introduzida em [1]. A extensão considera não apenas limitações superiores no número de carros que a estação pode lidar simultaneamente, mas também limitações inferiores. Estas novas limitações visam uma distribuição mais homogênea do trabalho nas estações de instalação de opções.

O artigo de Bautista et al. [1] é, até onde sabemos, o único a lidar com o xCSP. Neste trabalho propomos uma nova simples heurística para o problema baseada na metaheurística GRASP. Focamos em estratégias e estruturas de dados para acelerar procedimento de busca local ao invés de desenvolver uma heurística complexa. As demais partes do artigo são organizadas como se segue: Seção 2 define o problema e introduz nomenclaturas, operações e funções utilizadas. Seção 3 descreve a heurística GRASP proposta. Na seção 4 a heurística construtiva utilizada durante a otimização heurística é detalhada. Seção 5 descreve a busca local. Na seção 6 apresentam-se os resultados obtidos. Por fim na seção 7 realizam-se as conclusões.

2. O Problema Estendido de Sequenciamento de Carros

O xCSP é definido por uma tupla (C,O,p,q,r,s,t) onde C = $\{c_1, c_2, \dots, c_n\}$ é o conjunto de carros a ser produzido e O = $\{o_1, o_2, \dots, o_k\}$ é o conjunto de diferentes opções a serem instaladas. Vetores p e q definem as capacidades de máximo associadas a cada opção o_i, isto é, em cada sequência de $q(o_i)$ carros no máximo $p(o_i)$ deles podem requerer o_i. Analogamente, vetores r e s definem as capacidades de mínimo associadas com cada opção, isto é, em cada sequência de $s(o_i)$ carros no mínimo $r(o_i)$ deles devem requerer o_i. A função $t: C \times O \rightarrow \{0,1\}$ é a função de requerimentos, para cada carro c_i e opção o_j, $t(c_i, o_j)$ é 1 se a opção o_j deve ser instalada no carro c_i e é 0 caso contrário.

Abaixo listamos algumas notações utilizadas ao longo deste artigo:

• uma sequência $\pi = <c_{\pi_1}, c_{\pi_2}, \dots, c_{\pi_k}>$ é uma sucessão de carros.
•$|\pi|$ é o número de carros na sequência π.
• a concatenação de duas sequências π_1e π_2 chamada $\pi_1.\pi_2$ é a sequência de carros π_1 seguida pela sequência de carros π_2.
• a sequência π_k é um fator de outra sequência π chama $\pi_k \subseteq \pi$ se existirem duas (possivelmente vazias) sequências π_1e π_2 tal que $\pi = \pi_1.\pi_k.\pi_2$
• o número de carros requerendo a opção o_i em uma sequência π é denotada t(π, o_i) e definida por t(π, o_i) = $\sum_{c_k \in \pi} t(c_k, o_i)$.
• a posição inicial de uma subsequência na sequência original é denotada por π_{k_0}. Se por exemplo π_{k_0} = 3 sabemos que a subsequência π_k começa na terceira posição de π.
• dada uma sequência π, o índice de utilização dinâmico de uma opção [3] é definida por dynUtilRate $(o_i, \pi) = \frac{[t(C,o_i) - t(\pi,o_i)].q(o_i)}{p(o_i).[|C| - |\pi|]}$.

Podemos então definir o custo de máximo de uma sequência dado por:

$$max\_cost(\pi) = \sum_{o_i \in O} \sum_{f(\pi,i,k)} max\_violations(\pi_k, o_i)$$

onde

• $f(\pi, i, k)$ é o conjunto das sequências $\pi_k \subseteq \pi$ tal que $\begin{cases} |\pi_k| = q(o_i) & se\ \pi_{k_0} > 0 \\ p(o_i) + 1 \leq |\pi_k| \leq q(o_i) & se\ \pi_{k_0} = 0 \end{cases}$

• $max\_violations(\pi_k, o_i) = \begin{cases} 0 & se\ t(\pi_k, o_i) \leq p(o_i) \\ t(\pi_k, o_i) - p(o_i) & se\ t(\pi_k, o_i) > p(o_i) \end{cases}$

ou seja, *max_violation* conta o número de carros com determinada opção que excedem a limitação superior $p(o_i)$ para cada subsequência $q(o_i)$. A função *max_cost* soma as violações (violações de máximo) sobre cada opção e cada subsequência de tamanho $q(o_i)$ da sequência original.

Analogamente podemos definir o custo de mínimo de uma sequências por:

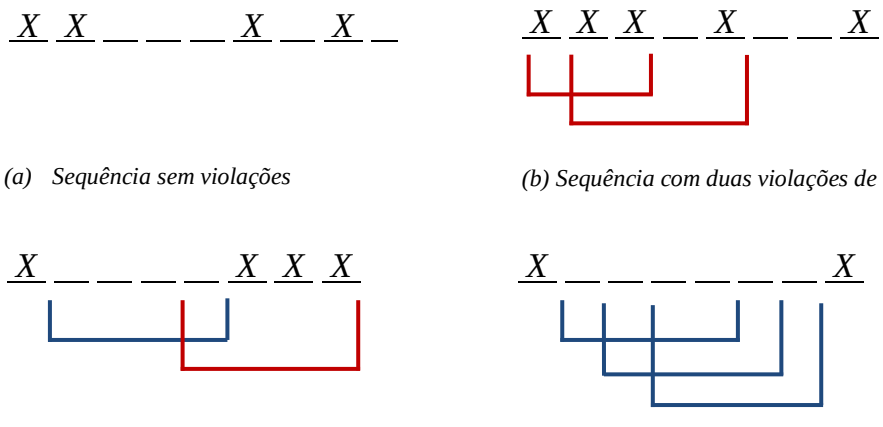
$$min\_cost(\pi) = \sum_{o_i \in O} \sum_{\substack{\pi_k \subseteq \pi \text{ tal que} \\ |\pi_k| = s(o_i)}} min\_violations(\pi_k, o_i)$$

onde

• $min\_violations(\pi_k, o_i) = \begin{cases} r(o_i) - t(\pi_k, o_i) & se\ t(\pi_k, o_i) < r(o_i) \\ 0 & se\ t(\pi_k, o_i) \geq r(o_i) \end{cases}$

em outras palavras, $min\_violations$ conta o número de carros com certa opção que estão faltando para alcançar o limite inferior de cada subsequência de tamanho $s(o_i)$. A função min_cost soma as violações (violações de mínimo) para cada opção e cada subsequência de tamanho $s(o_i)$ da sequência original.

Para ilustrar o problema suponha a seguinte situação: considere um subconjunto de carros que requerem a instalação de apenas uma opção. Chamamos este subconjunto de c_A e a opção de o_a. Suponha que $p(o_a) = 2$ e $q(o_a) = 4$, ou seja, em cada subsequência de quatro carros no máximo dois deles devem possuir a opção instalada. Suponha também, $r(o_a) = 1$ e $s(o_a) = 4$, então em toda subsequência de quatro carros pelo menos um deve possuir a opção instalada. A Figura 1 ilustra algumas possíveis situações:



(a) Sequência sem violações

(b) Sequência com duas violações de máximo.

(c) Sequência com uma violação de máximo e uma de mínimo.

(d) Sequência com três violações de mínimo.

Figura 1: *Exemplo de violações para um conjunto de carros em uma sequência de tamanho oito. Carros do conjunto c_A são representados com um X os demais com um espaço vazio.*

O xCSP consiste em achar uma sequência de carros a serem produzidos que minimizem a soma ponderada de funções max_cost e min_cost. Observe que em comparação com o clássico CSP a formulação do xCSP introduz as seguintes mudanças: (i) inclui o conceito de limitações inferiores, (ii) considera um problema de otimização ao invés de um de viabilidade, e (iii) considera subsequências de tamanho menor que $q(o_i)$ no início da sequência.

3. Abordagem Heurística

Para resolver o problema heuristicamente implementamos um algoritmo baseado na metaheurística GRASP [2], que funciona da seguinte forma:

1. Solução Final ← ∅
2. **para** k=1 **até** MAXIT **faça**
   a) Solução ← Algoritmo Guloso Randomizado
   b) Solução ← Busca Local (Solução)
   c) **se** Solução melhor que Solução Final
   Solução Final ← Solução
   fim-se
   fim-para
3. **retorne** Solução Final.

Em cada iteração uma solução inicial é construída por um algoritmo guloso randomizado para ser refinado por um procedimento de busca local. Este processo é repetido MAXIT vezes, atualizando a solução toda vez que encontramos uma solução melhor. Esta solução final é retornada quando o algoritmo termina.

As próximas duas seções detalham os dois principais procedimentos do algoritmo: a heurística gulosa randomizada e o procedimento de busca local.

4. Algoritmo Guloso Randomizado

O algoritmo randomizado proposto é baseado na heurística proposta em [3]. Esta constrói uma sequência de carros adicionando, em cada iteração, um dado carro ainda não sequenciado na última posição da sequência parcial que esta sendo construida. A função gulosa que decide qual carro adicionar em cada passo considera o número de violações de máximo que a inclusão do carro gera na sequência em construção. Esta encontra o carro c_j que minimiza:

$$newViolations(\pi, c_j) = \sum_{o_i \in O} t(c_j, o_i) * max\_violations(last\_cars(\pi.c_j, q(o_i)), o_i)$$

onde $last\_cars(\pi.c_j, q(o_i))$ é a sequência composta pelos últimos $q(o_i)$ carros de $\pi.c_j$ se $|\pi.c_j| \geq q(o_i)$ ou a sequência $\pi.c_j$ caso contrário. Empates neste critério são não apenas possíveis, mas frequentes. Para este caso apresentamos um critério de desempate:

1. Escolha o carro que minimize $newViolations\_min(\pi, c_j)$ (1).

2. Se ainda houver mais de um carro diferente empatado no critério guloso, crie a lista de candidatos restritos escolhendo dois dos carros empatados e escolha o carro que possui o maior índice de utilização dinâmica, definido por:
   $DHU(c_j) = \sum_{o_i \in O} t(c_j, o_i) * weight(o_i)$
   onde $weight(o_i) = 2^k$ se o_i é a opção com o k menor dynUtilRate para a sequência atual.

(1) $newViolations\_min\ (\pi, c_j)$
$$= \sum_{o_i \in O} t(c_j, o_i) * min\_violations(last\_cars(\pi.c_j, s(o_i)), o_i)$$

5. Busca Local

A vizinhança swap é definida por um operador que consiste na troca de posição de dois carros de uma dada sequência π, isto é:

$$Swap(j, i, \pi) = \pi_1.c_i.\pi_2.c_j.\pi_3$$

onde a forma de π era inicialmente $\pi_1.c_j.\pi_2.c_i.\pi_3$.

O movimento pode modificar o valor da função objetivo apenas se $t(c_j, o_i) \neq t(c_i, o_i)$ para algum $i \in O$.

Outras vizinhanças geradas por outras operações são usualmente empregadas em procedimentos de busca local para o CSP na literatura. Entre eles podemos citar inserção e inversão:

$Insertion(c_i, \pi, p) = \pi_1.\pi_2.c_i.\pi_3$ onde π era da forma $\pi_1.c_i.\pi_2.\pi_3$ e p a nova posição de c_i em π

$Inversion(\pi_1.\pi_2.\pi_3) = \pi_1.reverses(\pi_2).\pi_3$ onde $reverses(\pi.c_j) = c_j.reverses(\pi)$

Outras vizinhanças podem ser encontradas na literatura. Neste artigo, apenas a vizinhança swap é utilizada. A vizinhança é conectada (todas as soluções viáveis são alcançadas por qualquer outra solução por um número finito de movimentos swap) e é grande suficiente (cada solução possui $O(n^2)$ vizinhos) para permitir a implementação de um eficiente procedimento de busca local.

Seja a função $Delta$ definida a seguir*:* dados um custo de máximo e um custo de mínimo da sequência π, $Delta$ retorna a variação dos custos de máximo e de mínimo se os carros c_j e c_i trocarem de posição. Esta variação é definida por:

$$Delta(c_j, c_i) = M * (max\_cost(\pi_1.c_i.\pi_2.c_j.\pi_3) - max\_cost(\pi)) + (min\_cust(\pi_1.c_i.\pi_2.c_j.\pi_3) - min\_cost(\pi)).$$

onde M é uma grande constante como definido em [1].

Embora definido desta forma, o cálculo de $Delta$ seria extremamente ineficiente se executasse explicitamente as funções computacionais max_cost e min$_cost$. Avaliar o custo de uma sequência π inteira é um processo muito custoso. Sua complexidade computacional é $O(|O| * |C| * q_{max})$ onde $q_{max} = max\{q(o_i)\ \forall\ i\}$. Para cada uma das $|O|$ opções existem $O(|C|)$ subsequências a serem avaliadas, contando em cada uma delas o número de carros que requerem a instalação da opção dada.

Note que a diferença entre duas subsequências consecutivas é no máximo de dois carros. Depois de avaliar a função para a primeira subsequência, cada próxima subsequência pode ser avaliada em O(1). Logo, a complexidade computacional cai para $O(|O| * |C|)$. Além disto, observando que apenas $O(q_{max})$ sequências podem ser afetadas pelo movimento swap, a complexidade computacional pode ser reduzida para $O(|O| * q_{max})$. Portanto podemos definir uma função max_cost_temp, com complexidade $O(|O| * q_{max})$, que dado um custo de máximo da sequência π e dois carros c_j e c_i a serem trocados retorna o novo custo considerando apenas as subsequências onde custo pode variar.

Analogamente podemos definir min_cost_temp com complexidade O(min_cost_temp) = O($|O| * s_{max}$).

Então $Delta$ é definido da nova forma:

$$Delta(c_j, c_i) = M * (max\_cost\_temp(K_1, \pi, c_j, c_i) - K_1) + (min\_cost\_temp(K_2, \pi, c_j, c_i) - K_2)$$

onde K_1 e K_2 são o custo de máximo e custo de mínimo respectivamente da sequência na qual a operação Swap esta sendo avaliada.

Por fim, propomos um melhor procedimento de busca local como se segue:

1. **faça**:
2. **para** i=1...|C|-1 **faça**:
3. **para** j=i...|C| **faça**:
4. $Delta(c_j, c_i)$
5. fim-para
6. fim-para
7. Encontre c_j e c_i tal que $Delta(c_j, c_i)$ é o menor $\forall\, i, j$.
8. $Swap(j, i, \pi)$
9. **While** ($Delta < 0$).

6. Resultados

Todas as instâncias utilizadas neste artigo foram providas por CSPlib [6] . Um total 74 foram utilizadas nos experimentos, quatro delas com 100 carros enquanto as demais 70 instâncias possuem 200 carros. Cada instância considera cinco opções para serem ou não instaladas em cada carro. *Tabela 1* apresenta as características de cada instância.

Para adaptar as instâncias para o xCSP construímos limitação de mínimo por cada limitação de máximo como feito em [1]:

$$s(o_i) = q(o_i) \text{ e } r(o_i) = p(o_i) - 1$$

Para propósitos de comparação todos os testes são rodados por um máximo de 1000 segundos para cada instância, como em [1]. (o algoritmo termina se uma solução de custo 0 é encontrada para ambas violações de máximo e de mínimo).

O algoritmo foi implementado em C e compilado usando o GCC. Os testes foram rodados em um Pentium IV 3.0 GHz com o sistema operacional Linux. Tanto o computador utilizado quanto as instâncias testadas são idênticas as utilizadas em [1].

Conjunto	Número de instâncias	Instâncias resolvíveis	% de uso de cada opção	Total de carros
Clássicas	9	4	Variável	100
p60	10	10	60%	200
p65	10	10	65%	200
p70	10	10	70%	200
p75	10	10	75%	200
p80	10	10	80%	200
p85	10	10	85%	200
p90	10	10	90%	200

Tabela 1 *– Características de cada instância. Instâncias resolvíveis referem-se apenas a violações de máximo.*

Apenas instâncias clássicas que são sabidamente resolvíveis, isto é, aquelas para as quais uma solução com zero violação de máximo é conhecida, são utilizadas neste artigo. Estas soluções foram encontradas em [4] e [5].

A Tabela 2 apresenta os resultados computacionais. A segunda e terceira coluna apresentam o número de violações de máximo respectivamente para o algoritmo descrito neste artigo e para o apresentado em [1]. As próximas duas colunas apresentam o número de violações de mínimo. Nas quatro colunas os valores apresentados são os valores médios obtidos na execução das heurísticas para cada conjunto de instâncias. As duas últimas colunas mostram o número de instâncias resolvidas para cada conjunto. O tempo médio para cada iteração GRASP foi de 0,3 segundos para o algoritmo descrito neste trabalho, enquanto em [1] um tempo médio de 1 segundo iteração GRASP é reportado.

Conjunto	UOA	UOA [1]	LUA	LUA [1]	Instâncias Resolvidas	Instâncias Resolvidas [1]
Clássico	2,5	7,75	0	0	1	0
p60	0	0	11	14,3	10	10
p65	0	0	1,3	2,8	10	10
p70	0	0	0,1	0,1	10	10
p75	0	0	0,1	0	10	10
p80	0	0	0	0,1	10	10
p85	0	0,3	2,1	3,2	10	8
p90	0,3	4,2	0,8	2	7	2

Tabela 2 *– Custos médios encontrados para cada conjunto de instâncias. UOA representa o custo de máximo e LUA os custos de mínimo.*

Tendo em vista a Tabela 2 podemos enfatizar que todos os custos encontrados no mesmo tempo de execução foram menores ou iguais, com uma única exceção para o custo de mínimo no conjunto de instâncias p75.

7. Conclusões

Neste artigo introduzimos um novo algoritmo heurístico GRASP para Problema Estendido de Sequenciamento de Carros. A heurística construtiva é uma adaptação da heurística para o clássico Problema de Sequenciamento de Carros. O procedimento de busca local é bem simples, utilizando apenas a vizinhança swap.

Uma implementação cuidadosa do procedimento de busca local em termos da avaliação de custos das soluções vizinhas nos permitiu decrescer o tempo de execução de cada iteração GRASP em comparação com [1]. Em consequência nosso algoritmo encontrou resultados muito melhores no mesmo tempo de execução com o mesmo ambiente computacional.

Trabalhos futuros incluem o uso de outras vizinhanças no procedimento de busca local e o desenvolvimento de novas heurísticas baseadas em outras metaheurísticas.

Referências

[1] **Bautista, J., Pereira, J., Adenso-Díaz, B.** (2008). A GRASP approach for the extended car sequencing problem, Journal of Scheduling, v.11 n.1, p.3-16, Fevereiro

[2] **Feo, T. A., Resende M. G. C.** (1989) A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters, 8, 67–71.

[3] **Gent, I.** (1998). Two results on car-sequencing problems (Technical report APES-02-1998).

[4] **Gottlieb, J., Puchta, M., Solnon, C**. (2003). A study of greedy, local search and ant colony optimization approaches for car sequencing problems. In G. R. Raidl et al. (Eds.), Lecture notes in computer science: Vol. 2611. Applications of evolutionary computing (pp. 246–257). Berlin: Springer.

[5] **Regin, J. C., Puget, J. F.** (1997). A filtering algorithm for global sequencing constraints. In Lecture notes in computer science: Vol. 1330. Principles and practice of constraint programming, CP-97 (pp. 32–46). Berlin: Springer.

[6] http://www.csplib.org/