INSTITUTO DE COMPUTAÇÃO - UNICAMP
MC658 - Análise de Algoritmos III

Docente: Prof. Cid Carvalho de Souza
PED: Natanael Ramos

3º Trabalho Prático

Grupo 03:

José Ribeiro - RA : 176665
Felipe Lopes De Mello - RA : 171119

IC/UNICAMP
2019

1 Modelação do Problema

1.1 Variáveis Adotadas

Duas variáveis foram escolhidas para a modelagem do problema. A primeira delas, chamada de *makespan*, define o tempo de término do último trabalho (*job*) a ser executado. Como o objetivo do problema é encontrar um escalonamento viável onde o último *job* deve ser concluído o mais cedo possível, esta será a variável utilizada na função objetivo do resolvedor do Minizinc. Como os tempos são dados em números inteiros, o domínio dessa variável são os inteiros no intervalo denotado de *indexTime* que vai de 0 à soma dos produtos do números de *jobs* por suas respectivas durações. Ou seja, no pior caso, todos os *jobs* serão executados sequencialmente.

A segunda variável é um vetor de tamanho igual ao número de *jobs* onde cada posição assume um valor dentro do intervalo *indexTime*. Esta variável é chamada de *jobStartTime* e representa os tempos de início de cada *job*.

1.2 Restrições

Foram impostos 5 conjuntos de restrições ao modelo projetado com o objetivo de encontrar um escalonamento viável.

O primeiro conjunto de restrições impõe que se um *job* 'A' antecede outro *job* 'B' em uma mesma ordem, então 'A' deve terminar antes do início de 'B' no escalonamento final. Essa restrição é construída a partir de um laço que itera sobre o conjunto de *jobs*. Note que há restrições redundantes dentro deste conjunto para ordens com mais de 2 *jobs* com o objetivo de otimização. Este fato pode ser verificado com o exemplo de uma ordem com 3 *jobs* 'A', 'B' e 'C'. Esse conjunto relacionará os *jobs* 'A' com 'B', 'B' com 'C' e 'A' com 'C', mas esta última relação poderia ser inferida pelas duas primeiras.

O segundo conjunto relaciona as duas variáveis adotadas, restringindo o valor de *makespan* para ser maior ou igual ao tempo de término de todos os *jobs*.

A terceira restrição é redundante em relação ao primeiro conjunto de restrições, entretanto, é construída utilizando a restrição global *disjunctive* do Minizinc com os parâmetros *jobStartTime* e as respectivas durações de cada *job*. Assim *jobs* de mesma ordem não podem ocorrer ao mesmo tempo.

O quarto conjunto de restrições é semelhante ao primeiro. Se uma relação de precedência entre dois *jobs* 'A' e 'B' é dada, então 'A' deve terminar antes do início de 'B' no escalonamento final.

A última restrição, quinta, limita o número máximo de trabalhadores em todo instante de tempo. Note que este problema se reduz ao problema de empacotamento (*bin packing problem*) no qual os objetos são as tarefas, o tamanho de cada objeto equivale ao número de trabalhadores necessários para realização da dada tarefa e cada instante de tempo é um recipiente (*bin*) com tamanho limite igual ao número de trabalhadores L. Assim, utiliza-se a restrição global *bin_packing* do Minizinc com os parâmetros descritos para limitar o número máximo de trabalhadores em todo instante de tempo.

1.3 Estratégia de Busca

Como o objetivo do modelo é minimizar a variável *makespan* e esta, por sua vez, possui seu domínio dentro dos inteiros, utiliza-se a estratégia de busca *int_search()* do Minizinc. Além disso, como o vetor de tempos de início de trabalhos tendem a gerar um *makespan* menor, também utiliza-se *int_search()* com este vetor. Para associar ambas estratégias, utiliza-se *seq_search()*.

O primeiro parâmetro da estratégia apenas define a variável na qual a estratégia será implementada. Já o segundo parâmetro impõe que as primeiras variáveis a serem atribuídas serão as com menor tamanho de domínio, este método é denominado (*first_fail*). Esta escolha se deveu a resultados práticos observados que demonstraram sua superioridade em relação ao método mais intuitivo (*smallest*) de escolher a variável que possuí o menor valor. Isto é provavelmente uma consequência do número de restrições impostas ao problema. Como são muitas, o método *smallest* falha em encontrar uma solução viável muitas vezes, prejudicando seu tempo de execução.

O terceiro parâmetro institui a atribuição do menor valor de domínio possível (*indomain_min*) para as variáveis. Esta escolha é natural uma vez que tratamos de um problema de minimização. Outras estratégias foram utilizadas, tal como *indomain_median*, *indomain_random* e *indomain_split*. Contudo, com base nos testes realizados, os melhores resultados obtidos de fato foram utilizando *indomain_min*.

O último parâmetro, com keyword *complete*, apenas define que toda árvore de busca deve ser percorrida até que se encontre uma solução.

2 Otimizações Realizadas

2.1 Identificação e otimização da constraint mais custosa

Para sabermos como deveríamos realizar as otimizações, tivemos que primeiro identificar a constraint que estava consumindo a maior parte do tempo de execução. Chegamos a conclusão que limitar a quantidade de workers que podem ser utilizados em cada instante de tempo, se mostrou como a constraint mais custosa. Isto pois, removendo esta constraint, os problemas se tornavam extremamente simples, sendo que na última instância *Ins_12o_109j_A* o tempo necessário para resolvê-la foi menos de $1s$. Por isto, a mesma foi um dos principais alvos de nossas otimizações.

Para uma primeira implementação do limitante de workers por instante de tempo citada no parágrafo anterior, utilizamos a constraint 5.1, comentada em nosso código. Apesar de aparentemente correta, o grau de ineficiência da mesma foi inimaginável, não nos permitindo resolver em optimalidade ao menos a instância *Ins_4o_23j_A* (estávamos obtendo o valor 70 para esta instância, ao invés de 58). Além disso, para instâncias a partir de *Ins_6o_41j_A* nenhuma solução foi encontrada.

Em uma segunda tentativa, buscamos melhorar a implementação anterior. Ou seja, produzimos a constraint 5.2, também comentada em nosso código. Com ela, passamos a iterar somente pelos tempos onde jobs estavam executando (ao invés de todos os possíveis tempos descritos por *indexTime*). Assim, pudemos encontrar em optimalidade o valor da solução referente a instância *Ins_4o_23j_A*. Porém, ainda com esta melhora, continuamos não encontrando respostas para instâncias maiores, como *Ins_6o_41j_A* ou maior.

Em uma terceira tentativa, decidimos recorrer a global constraint *cumulative*, aplicada no código como constraint 6, estando esta comentada. Para utilizá-la, tivemos que reformular a forma como descrevíamos os jobs, de forma que cada job j fosse decomposto em k tarefas de duração 1 e requisição de trabalhadores X, sendo k a duração do job j. Com isto, conseguimos impor que a utilização de workers para cada unidade de tempo não ultrapassasse o limite L. Contudo, apesar de promissor o uso, não notamos grandes melhoras no tempo despendido, de forma que a quantidade de instâncias sem solução ainda continuasse alta.

Por fim, em uma quarta e última tentativa, decidimos utilizar uma outra global constraint chamada *bin_packing*. Com ela, nosso cenário de soluções

obtidas melhorou consideravelmente, de forma com que boas soluções puderam ser encontradas para qualquer uma das instâncias no intervalo de $15s$. A comparar, citamos novamente a instância *Ins_4o_23j_A*. Antes de utilizarmos a constraint *bin_packing*, apesar de já estarmos obtendo a solução ótima, o minizinc não foi capaz de em 5s inferir se ela de fato era ótima. Já com a constraint *bin_packing* ativada, este tempo caiu para $0.018s$.

2.2 Otimizações realizadas com a estratégia de busca

Após otimizarmos a constraint mais custosa, decidimos forcar na segunda região do código que mais impactava na produção de boas soluções: as estratégias de busca escolhidas.

Para tanto, em uma primeira tentativa de implementação, decidimos seguir o enfoque mais intuitivo: smallest, que busca escolher a variável com o menor valor em seu domínio. Tal enfoque funcionou bem enquanto estávamos utilizando as constraint 5.1, 5.2 e 6. Contudo, ao considerarmos apenas a constraint 5, *bin_packing*, os resultados não foram muito promissores para instâncias grandes. Isto é, mesmo com $3m$, não estávamos conseguindo resolver certas instâncias, mesmo com a melhora obtida com o *bin_packing*. Por este motivo, decidimos testar outras combinações.

Uma segunda tentativa, foi o uso do *dom_w_deg*. Os resultados, obtidos por esta estratégia foram os que produziram os piores resultados, fazendo com que soluções boas passassem a ter seu valor piorado em mais de duas vezes.

Assim, numa terceira tentativa, decidimos utilizar a estratégia *first_fail*. Esta foi o que nos possibilitou obter os melhores resultados (assumindo o uso do *bin_packing*). A citar, novamente tomando a instância *Ins_4o_23j_A*, conseguimos reduzir o tempo de $0.018s$ para $0.014s$. Além disso, em instâncias maiores, tal como a *Ins_12o_109j_A*, usando a estratégia de *first_fail* pudemos reduzir o valor do makespan de 1790 para 1622, assumindo para ambas o tempo de $3m$. Mesmo fato se repetiu para a instância *Ins_12o_108j_A*, onde pudemos reduzir o makespan de 1508 para 1461.

2.3 Resumo das otimizações realizadas

Em suma, quatro técnicas de otimização foram aplicadas ao longo do projeto com o objetivo da geração de melhores soluções.

A primeira delas se baseia no fato de que a redução do domínio de variáveis resulta em um número de possíveis soluções menor, e assim, reduz-se o tempo de execução. Como o domínio de ambas variáveis é igual (*indexTime*), a técnica consiste apenas em reduzir este domínio. Para isso, observa-se que existem ordens nas quais todas as suas tarefas possuem uma demanda de trabalhadores menores ou iguais a metade do total disponível. Assim, considerando essas ordens, metade delas podem ser executadas simultaneamente com a outra metade reduzindo assim o tempo máximo necessário para que todos os trabalhos sejam concluídos. Entretanto, essa técnica se mostrou desnecessária uma vez que a estratégia de busca sempre inicia com os menores valores de tempo possível e errada, pois caso haja uma relação de precedência que obrigue os trabalhos a serem executados sequencialmente, o domínio possuiria um tamanho menor.

A segunda técnica é a determinação da estratégia de busca através de resultados experimentais. Foi utilizado a estratégia de busca *first_fail* e *indomain_min*, como já discutido na seção 1.3. Ou seja, a melhor estratégia encontrada empiricamente foi utilizando o método de iniciar com a atribuição dos menores valores possíveis às variáveis com menor tamanho de domínio.

A terceira técnica é o uso de restrições globais o que tende a otimizar o algoritmo, pois utiliza as implementações eficientes dos próprios resolvedores. Durante o projeto foram aplicados 3 diferentes restrições globais sendo elas a *disjunctive*, *bin_packing* e a *cumulative*. A adição da restrição *disjuctive* possuiu um pequeno impacto, porém benéfico, no tempo de execução do algoritmo para as instâncias testadas, o que implica que o primeiro conjunto de restrições descreve bem o modelo. A restrição *bin_packing* resultou em uma grande otimização em relação às restrições equivalentes escritas explicitamente. Finalmente, a restrição *cumulative* se mostrou ineficaz e contraprodutiva, provavelmente uma consequência do número de variáveis presente (igual ao número de tarefas).

A quarta técnica é o uso de restrições redundantes que explicitam associações que antes eram implícitas. Esta técnica foi aplicada no primeiro conjunto de restrições. Primeiramente, este conjunto apenas associava um trabalho com seu sucessor, verifica-se ambos estavam na mesma ordem e caso positivo, restringia-os para que o sucessor começasse apenas após o término do primeiro. Como descrito na seção 1.2, esta abordagem gerava muitas restrições implícitas para ordens com 3 ou mais trabalhos. Ao explicitá-las obtém-se otimizações, pois as restrições de domínio de variáveis são encontradas mais eficientemente.

3 Resultados do Experimento

3.1 Dados da máquina

O experimento foi realizado em uma máquina com 15.5 GiB de RAM e Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8, rodando Ubuntu 18.04.2 LTS. O programa foi executado com Minizinc em sua versão *2.2.3*. Além disso, utilizamos o gecode em sua versão *6.1.0* como constraint solver.

3.2 Análise dos resultados obtidos

Os resultados demonstrados na tabela 1 da próxima página mostram que o algoritmo projetado para resolver este problema encontra soluções, não necessariamente ótimas, para todas as instâncias testadas. Em particular, para as 3 primeiras instâncias, determina-se a solução ótima com tempos de execução muito baixo (menores do que 0.01 segundos). A partir da quarta instância (com exceção da instância *Ins_4o_27j_A*) os tempos de execução atingiram o tempo de execução limite (3 minutos). Assim nota-se uma mudança drástica entre as instâncias *Ins_4o_23j_A* e *Ins_4o_24j_A*.

As hipóteses que explicam esse comportamento são formuladas a seguir. Como ambas instâncias possuem um número igual de ordens e um número muito próximo de trabalhos e tarefas, infere-se que os fatores determinantes se devem a duração de cada trabalho e ao número de trabalhadores necessários para cada tarefa. Este último fato é reforçado pelas observações descritas na seção 2.1 que discutem a restrição de limitar o número de trabalhadores por tarefa como o fator determinante do tempo de execução do algoritmo. A hipótese é de que quanto mais baixo o número de trabalhadores necessário por tarefa, mais combinações de execução entre diferentes trabalhos ao mesmo tempo são possíveis, e, portanto, mais valores de atribuição de variáveis são possíveis, o que prejudica o tempo de execução.

Também é notável o alto número de nós explorados a partir da quarta instância. Como o tempo de execução limite é de 3 minutos, tem-se um tempo de execução por nó muito baixo. Esta característica, embora não seja suficiente para a determinação da solução ótima, é benéfica pois pelo menos possui mais chances de se encontrar uma solução viável. Assim, caso o objetivo do modelo seja a sua aplicação em um problema prático que possui restrição de tempo, é importante que pelo menos uma solução viável seja encontrada, como é o caso do algoritmo projetado.

3.3 Resultados Obtidos

Table 1: Valores de *makespan*, *solveTime* e *nodes* para cada instância.

Instância	***makespan***	***solveTime (s)***	***nodes***
Ins_3o_7j_A	**10**	**0.00**	**9**
Ins_4o_21j_A	**82**	**0.06**	**1545**
Ins_4o_23j_A	**58**	**0.015**	**297**
Ins_4o_24j_A	**69**	**180.00**	**5898292**
Ins_4o_24j_B	**75**	**180.00**	**8723505**
Ins_4o_27j_A	**67**	**113.29**	**2948986**
Ins_6o_41j_A	**155**	**180.001**	**4728851**
Ins_6o_41j_B	**124**	**180.001**	**3604064**
Ins_6o_41j_C	**138**	**180.00**	**4687917**
Ins_6o_44j_A	**123**	**180.00**	**4529337**
Ins_6o_44j_B	**154**	**180.00**	**2257661**
Ins_8o_63j_A	**290**	**180.00**	**2029487**
Ins_8o_63j_B	**387**	**180.01**	**1957863**
Ins_8o_63j_C	**342**	**180.01**	**2318117**
Ins_8o_65j_A	**430**	**180.01**	**2848063**
Ins_8o_65j_B	**451**	**180.01**	**1951956**
Ins_10o_84j_A	**739**	**180.03**	**1609545**
Ins_10o_84j_B	**648**	**180.02**	**1440798**
Ins_10o_85j_A	**1044**	**180.03**	**1173174**
Ins_10o_87j_A	**667**	**180.02**	**1411184**
Ins_10o_88j_A	**529**	**180.04**	**1039313**
Ins_10o_100j_A	**1668**	**180.10**	**751618**
Ins_10o_102j_A	**1385**	**180.10**	**928789**
Ins_10o_106j_A	**1222**	**180.10**	**976882**
Ins_12o_108j_A	**1461**	**180.10**	**978049**
Ins_12o_109j_A	**1622**	**180.11**	**200476**