Relatório do trabalho prático de Métodos de Apoio à Decisão

Diogo Pereira - up201605323 Frederico Lopes - up201604674
Sílvia Maia - up201605209

1 de Agosto de 2019

Resumo

O trabalho prático de Métodos de Apoio à Decisão consiste na resolução de problemas de satisfação de restrições em domínios finitos e problemas de otimização combinatória. Foram-nos propostos dois problemas de calendarização de tarefas com alguma partilha de recursos (*scheduling problems*). Para automatizarmos o processamento das resoluções destes problemas, fizemos programas usando sistemas de Programação Lógica por Restrições (ECLiPSe CLP) e linguagens imperativas/orientadas a objetos (C++).

1 Problema 1

Um projeto envolve um certo número de tarefas. É conhecida a duração de cada tarefa (dada por um número inteiro de dias) e a relação de precedência entre tarefas. Cada tarefa requer um certo número de trabalhadores (a tempo inteiro) e a contratação de um número de trabalhadores maior do que o necessário em nada altera a duração da mesma. Não há partilha de outros recursos entre tarefas (como, por exemplo, máquinas). Os trabalhadores contratados para o projeto estão habilitados a desempenhar qualquer tarefa. Determinar o número mínimo de trabalhadores a contratar de forma a concluir o projeto o mais cedo possível.

1.1 Modelos matemáticos

1.1.1 Para minimizar a duração do projeto definimos o seguinte modelo:

Dados:

n - número de tarefas do projeto.

d_i - duração da tarefa i, em dias.

P - relação de precedência entre tarefas.

t_i - número de trabalhadores necessários para realizar a tarefa i.

Variáveis de decisão:

D_i - data de inicio da tarefa i.

$Concl$ - data de conclusão do projeto.

Restrições:

$D_i + d_i \leq D_j, \forall (i,j) \in P$

$D_i + d_i \leq Concl, \forall i \in \{1, ..., n\}$

$D_i \in \mathbb{N}, \forall i \in \{1, ..., n\}$

$Concl \in \mathbb{N}$

Função objetivo: minimizar Concl.

1.1.2 Para minimizar o número de trabalhadores:

Dados:

n - número de tarefas do projeto.

d_i - duração da tarefa i, em dias.

P - relação de precedência entre tarefas.

t_i - número de trabalhadores necessários para realizar a tarefa i.

$Concl$ - data de conclusão do projeto.

Variáveis de decisão:

D_i - data de inicio da tarefa i.

T - número de trabalhadores.

Restrições:

$D_i + d_i \leq D_j, \forall (i,j) \in P$

$D_i + d_i \leq Concl, \forall i \in \{1, ..., n\}$

$\sum_{j=0}^{Concl} t_i \leq T$, tal que $j \in [D_i, D_i + d_i[, \forall i \in \{1, ..., n\}$

$D_i \in \mathbb{N}, \forall i \in \{1, ..., n\}$

$T \in \mathbb{N}$

Função objetivo: minimizar T.

1.2 Implementação

A implementação foi dividida em quatro fases:

- Determinar a duração mínima do projeto.
- Determinar o número de trabalhadores necessário se cada tarefa começar o mais cedo possível.
- Determinar o número de trabalhadores para as tarefas críticas.
- Determinar o número mínimo de trabalhadores.

Descrevemos nas próximas secções as implementações em ECLiPSE CLP e C++, respetivamente.

1.2.1 Implementação em ECLiPSe

1.2.1.1 Determinar o número mínimo de dias

Para encontrar o número mínimo de dias começamos por encontrar um *upper bound* desse mesmo número, que consiste em somar a duração de todas as tarefas, o que é equivalente a pensar que as tarefas são realizadas uma de cada vez e de forma sequencial.

Vamos depois estabelecer restrições sobre a data de inicio de cada tarefa usando o predicado "prec_constrs", que tem como argumentos a lista de tarefas, a lista que contém intervalos da data de inicio de cada tarefa e a data de conclusão do projeto, que vamos minimizar mais á frente. Neste predicado, para cada tarefa T vamos percorrer a lista de tarefas que ela precede e limitar as data de inicio destas a ser maior ou igual do que a data de fim da tarefa T. Limitamos também a data de fim de cada tarefa para ser menor do que a data de conclusão do projeto.

Para minimizar o número de dias usamos os predicados do ECLiPSe CLP: "minimize"[1] e "labeling"[2]. O que o predicado "minimize"faz é encontrar uma solução para um objetivo, de forma a minimizar o termo que quisermos. Neste caso, como queremos obter também as datas de *earliest start* de cada tarefa, o nosso objetivo vai ser dar valores á data de inicio de cada tarefa e á data de conclusão do projeto (utilizando o "labeling"), minimizando a data de conclusão.

```
get_min_days(Tasks,ESTasksL,Finish) :-
   length(Tasks,NTasks), length(ESTasksL, NTasks),
   max_days(Tasks, MaxD), ESTasksL#::0..MaxD, Finish#::0..MaxD,
   prec_constrs(Tasks,ESTasksL,Finish),
   minimize(labeling([Finish|ESTasksL]),Finish).
```

Ficamos assim com as datas de inicio de cada tarefa na lista *ESTasksL* e com a data mínima de conclusão do projeto em *Finish*.

1.2.1.2 Determinar o *upper bound* do número mínimo de trabalhadores

Para determinar o número de trabalhadores necessários se cada tarefa começar o mais cedo possível precisamos primeiro de ordenar a lista de datas de inicio de cada tarefa. De maneira a facilitar o cálculo, criamos duas listas que vão ser dois do argumentos do predicado "find_nworkers". A primeira vai ser uma lista de pares (Di,T), onde Di é a data de inicio da tarefa T, ordenada por ordem crescente das *Di*. Criamos também um lista de pares (Fi,T), onde Fi é a data de finalização da tarefa T, estando também esta lista ordenada por ordem crescente das *Fi*. Para além destes dois argumentos, o predicado vai ter ainda mais dois, um para manter registo do número de trabalhadores necessários em cada altura e outro para guardar o número máximo de trabalhadores.

O predicado "find_nworkers" consiste então em percorrer as listas ordenadas de inicio e fim de cada tarefa. Quando o elemento na cabeça da lista dos inícios for menor do que o da cabeça da lista dos finais de cada tarefa, significa que uma tarefa começou e por isso o número de trabalhadores aumentar. Caso contrário o número de trabalhadores diminui. Sendo assim, para obter o número de trabalhadores necessários só temos que ver qual é o máximo atingido em cada uma destas alturas.

```
find_nworkers([],[],0,0).
find_nworkers([(_,T)|LT],[],WorkersA,WorkersN) :- tarefa(T,_,_,W),
                      WorkersA2 is WorkersA + W,
                      find_nworkers(LT,[], WorkersA2, WorkersN2),
                      WorkersN is max(WorkersN2,WorkersA2), !.
find_nworkers([],[(_,T)|FL],WorkersA,WorkersN) :- tarefa(T,_,_,W),
                      WorkersA2 is WorkersA - W,
                      find_nworkers([],FL, WorkersA2, WorkersN2),
                      WorkersN is max(WorkersN2,WorkersA2), !.
find_nworkers([(Si,T)|LT],[(Fi,T2)|FL],WorkersA,WorkersN) :- Si < Fi, tarefa(T,_,_,W),
                      WorkersA2 is WorkersA + W,
                      find_nworkers(LT,[(Fi,T2)|FL], WorkersA2, WorkersN2),
                      WorkersN is max(WorkersN2,WorkersA2), !.
find_nworkers([(Si,T)|LT],[(Fi,T2)|FL],WorkersA,WorkersN) :- Si >= Fi, tarefa(T2,_,_,W),
                      WorkersA2 is WorkersA - W,
                      find_nworkers([(Si,T)|LT],FL, WorkersA2, WorkersN2),
                      WorkersN is max(WorkersN2,WorkersA2), !.
```

Conseguimos assim o número de trabalhadores necessários se cada tarefa tiver inicio o mais cedo possível, sendo este um *upper bound* do número mínimo de trabalhadores.

1.2.1.3 Determinar o *lower bound* do número mínimo de trabalhadores

Para determinar o número mínimo de trabalhadores, vamos primeiro determinar o número de trabalhadores necessários para as tarefas críticas, constituindo assim um *lower bound* do número mínimo de trabalhadores.

Para isso, vamos primeiro determinar quais são as tarefas críticas. Pela definição, uma tarefa crítica é uma tarefa que condiciona a duração total do projeto. Assim, para determinarmos quais são as tarefas críticas temos que voltar a usar o predicado "prec_constrs", para construir a lista de datas de inicio possíveis para cada tarefa, sendo que agora a data de conclusão de projeto que damos como argumento já está definida. Depois disso, apenas temos que analisar valores possíveis para a data de inicio de cada tarefa (usando o predicado de ECLiPSE CLP: "get_bounds"[3]) e se só houver um valor possível é porque é uma tarefa critica. É isto que o predicado "get_critical_tasks"está a fazer, sendo que um dos argumentos é uma lista de pares (Xi,T), onde Xi é a data de inicio da tarefa crítica T, que vai ser construída pelo predicado. Para determinar o número de trabalhadores, basta fazer como fizemos para determinar o número de trabalhadores na secção anterior.

```
get_critical_tasks(_,[],[]).
get_critical_tasks(T, [Xi|L], [(Xi,T)|CritTasksL]) :- get_bounds(Xi,LB,UB), LB=UB,
                      T2 is T + 1, get_critical_tasks(T2,L,CritTasksL), !.
get_critical_tasks(T, [Xi|L], CritTasksL) :- get_bounds(Xi,LB,UB), not(LB=UB),
                      T2 is T + 1, get_critical_tasks(T2,L,CritTasksL), !.
```

1.2.1.4 Determinar o número mínimo de trabalhadores

Depois de termos o número de trabalhadores para as tarefas críticas, vamos calcular o número mínimo de trabalhadores. Para isso vamos usar os predicados "minimize", "cumulative"[4] e "search"[5] do ECLiPSE CLP. Usando o predicado "cumulative", que vai impôr uma restrição, dado uma lista do dias de inicio de cada tarefa , tendo as tarefas determinada duração e utilizando cada tarefa um certo número de trabalhadores, em nenhuma altura irão existir mais do que o número mínimo de trabalhadores dado.

Usando esta restrição, basta procurar valores para o número trabalhadores e para as datas de início das tarefas, usando o predicado "search", de forma a minimizar o número de trabalhadores.

```
minimize_workers(STasksL, DurationL, WorkersL, WorkersN, WorkersCrit, WorkersMin) :-
                WorkersMin#::WorkersCrit..WorkersN,
                cumulative(STasksL,DurationL,WorkersL,WorkersMin),
                minimize(search([WorkersMin|STasksL],0,first_fail,indomain_min,complete,[]),
                    WorkersMin).
```

Para ver se a solução é única ou não, basta repôr os valores do vetor das datas de início das tarefas de maneira a fazer uma busca, usando novamente o "cumulative"e o "search"á procura de uma solução diferente da anterior. Se existir, significa que existe pelo menos mais uma solução ótima e portanto não é uma solução única.

1.2.2 Implementação em C++

1.2.2.1 Determinar o número mínimo de dias

Para encontrar o número mínimo de dias usamos o método do caminho crítico (*CPM*) para sabermos qual é a data de início mais cedo para cada tarefa. O *CPM* calcula a data de inicio mais cedo e mais tarde para todas as tarefas, e com esses dados conseguimos obter o número mínimo de dias que um projeto vai necessitar para estar concluído, pois esse resultado será a tarefa em que a soma da data de inicio mais cedo e a duração da mesma seja o maior valor de todas as tarefas. Este processo determina quais as tarefas são "críticas" (i.e., pertencem ao caminho mais longo e a data de inicio mais cedo é igual à data de inicio mais tarde) e quais tem folga na sua data de início (i.e., podem começar mais tarde sem alterar a data de conclusão do projeto). Com o *earlieststart* e o $latestfinish$, podemos calcular o *lateststart*.

```
// CPM_ES calcula e retorna a duração mínima do projeto sendo que o earliest start de cada
    tarefa irá para o vetor ES[] e as precedências para o vetor prec[].
int CPM_ES(Task* tasks[], int nTasks, int ES[], int prec[]);

// CPM_LF calcula a latest finish de cada tarefa e os resultados irão para o vetor LF[].
void CPM_LF(Task* tasks[], int nTasks, int LF[], int minDur);
```

Ficamos assim com as datas de inicio mais cedo de cada tarefa no vetor *ES[]* e a função retorna o valor da data mínima de conclusão do projeto que está alojado na variável *minDur*.

1.2.2.2 Determinar o *upper bound* do número mínimo de trabalhadores

Para determinar o número de trabalhadores necessários se cada tarefa começar o mais cedo possível precisamos de um vetor, *time*, com o mesmo tamanho que a duração mínima do projeto, em que todas as posições estão inicializadas a zero. Percorrendo todas as tarefas, somamos o número de trabalhadores que são necessários para essa tarefa ao vetor desde a posição de inicio da tarefa até ao fim da sua duração, ao valor que já se encontra no vetor. Quando todas as tarefas forem analisadas, podemos começar a percorrer o vetor com o auxílio de uma variável, *maxWorkers*, que se encontra inicialmente a zero. Percorrendo todos os dias da duração do projeto (posições do vetor *time*), vamos adicionando à variável *maxWorkers* o valor que está no vetor *time*. Em todas as posições, testamos se o vetor *time* contém algum número maior que a variável *maxWorkers*, e se for, o valor de *maxWorkers*

é atualizado para o valor de *time[i]*.

```
// calculateWorkers calcula o número de trabalhadores necessários se as tarefas começarem
    no tempo que está no vetor ES[].
int calculateWorkers(Task* tasks[], int nTasks, int* ES, int minDur);
```

Conseguimos assim o número de trabalhadores necessários se cada tarefa tiver inicio o mais cedo possível, sendo este um *upper bound* do número mínimo de trabalhadores.

1.2.2.3 Determinar o *lower bound* do número mínimo de trabalhadores

Para determinar o número de trabalhadores somente para as tarefas críticas podemos usar uma função idêntica à feita para determinar o número de trabalhadores se todas as tarefas começarem no seu tempo de começo mais cedo. A única diferença é que só adicionamos ao vetor *time* o número de trabalhadores das tarefas que são críticas, (i.e., earliest start da tarefa é igual ao latest start da mesma).

```
    // calculateCriticalWorkers retorna o número de trabalhadores necessários somente para
        as tarefas críticas.
    int calculateCriticalWorkers(Task* tasks[], int nTasks, int* ES, int* LS, int minDur);
```

1.2.2.4 Determinar o número mínimo de trabalhadores

Depois de termos os limites do número mínimo de trabalhadores necessários, vamos calcular o número mínimo de trabalhadores para o projeto (solução ótima). Para isso vamos usar um algoritmo de pesquisa parecido com o DFS (*Depth First Search*). Trata-se de uma função recursiva que a cada iteração vai escolher a melhor tarefa para analisar usando uma heurística denominada de *slack time*, diferença entre *latest start* e *earliest start*. Quando é escolhida uma tarefa para analisar, é usado um ciclo para chamar a função de pesquisa testando todos os valores possíveis para a data de início da a tarefa escolhida. A função retorna um valor booleano, *true* caso tenha sido encontrada uma solução com número de trabalhadores menor ou igual à variável *maxWorkers*, *false* caso contrário.

```
    // Função de pesquisa que retorna true caso tenha sido encontrado uma solução com
        número de trabalhadores menor ou igual a maxWorkers e false caso contrário.
    bool DFS(int maxWorkers, Task* tasks[],
        int nTasks, int* ES, int* LS, int minDur,
        priority_queue<pair<int, int>, vector<pair<int, int> >, decltype(pqComp)>& slack,
        int* points, int* start,
```

```
    int curMaxWorkers, int* bestStart);

int main(){
/*--------------------------------------------------------------------------------*/
for(int i=criticalWorkers; i<esWorkers; i++){
    if(DFS(i, tasks, nTasks, ES, LS, minDur, slack, points, start, criticalWorkers,
        bestStart)){
        esWorkers = i;
        break;
    }
}
if(DFS(esWorkers, tasks, nTasks, ES, LS, minDur, slack, points, start,
    criticalWorkers, bestStart))
    printf("There are alternative optimal solutions\n");
/*--------------------------------------------------------------------------------*/
}
```

Para ver se a solução é única ou não, basta correr a função recursiva de pesquisa outra vez e só retornar *true* caso tenha sido encontrada uma solução com valores de datas de início das tarefas diferentes das que foram encontradas anteriormente.

1.3 Resultados

Testamos ambos os programas para inputs com 7,9,13,258 e 1432 tarefas. Para os primeiros 3 casos, o ECLiPSE demorou cerca de 0.01 segundos e o C++ demorou em média 0.2 segundos. Para as 258 tarefas, obtiveram resultados muito parecidos, cerca de 3.7 segundos. A grande diferença está no último caso. Testamos para dois casos diferentes, e no ECLiPSE demoraram 71 e 102 segundos, enquanto que no C++ apenas demorar 2.3 e 1.7 segundos. Esta diferença deve-se muito ao facto de no C++ estarmos a fazer a pesquisa a partir do *lower_bound*, enquanto que no ECLiPSE, o predicado existente faz a pesquisa a partir do *upper_bound*, tendo muitas vezes que fazer mais pesquisas, pois nos casos em que testamos o mínimo de trabalhadores estava mais perto do *lower_bound*.

2 Problema 2

Um projeto envolve um certo número de tarefas. É conhecida a duração de cada tarefa (dada por um número inteiro de horas) e a relação de precedência entre tarefas. Cada tarefa requer uma equipa de trabalhadores de várias especialidades, sendo conhecido o número de trabalhadores necessários de cada especialidade. Cada trabalhador tem habilitações em uma ou mais especialidades. Se for escolhido para fazer parte da equipa que realizará uma certa tarefa, então nessa equipa só terá a seu cargo trabalho de uma especialidade. Podem existir tarefas que terão de começar dentro de um

intervalo de tempo após a conclusão de uma outra tarefa de que dependem. O projeto deverá estar concluído até ao fim de um certo dia. O horário de trabalho é de 8 horas por dia, de segunda a sexta-feira, exceto feriados (das 8:00 às 12:00 e das 13:00 às 17:00). Na definição do início de cada tarefa, deverá ser considerado um calendário real. Se não for possível concluir o projeto dentro do prazo previsto, terá de ser determinado o número mínimo de trabalhadores a contratar (se desse modo se puder concluir o projeto no prazo). Nesse caso, assumir-se-á que cada um dos novos trabalhadores só tem habilitações para uma especialidade. Pode ser útil também minimizar a duração do projeto, se for possível concluí-lo no prazo sem contratar mais trabalhadores.

2.1 Modelo matemático

2.1.1 Para minimizar o número de dias do projeto:

Dados:

n - número de tarefas do projeto.

nt - número de trabalhadores disponíveis.

ne - número de especialidades

e_{ij} - trabalhador i tem habilitação na especialidade j.

t_{ij} - número de trabalhadores da especialidade j necessários para realizar a tarefa i.

h_i - duração da tarefa i, em horas.

$Prazo$ - data limite para conclusão do projeto.

P - relação que define a precedência entre tarefas.

I - relação que define o intervalo de tempo em duas tarefas.

min_{ij} - atraso mínimo da tarefa j depois do fim da tarefa i, para $(i, j) \in I$.

max_{ij} - atraso máximo da tarefa j depois do fim da tarefa i, para $(i, j) \in I$.

Variáveis de decisão:

D_i - dia em que a tarefa i vai ser executada.

H_i - hora de início da tarefa i.

t_{ije} - trabalhador j executa a tarefa i na especialidade e.

$Concl$ - data de conclusão do projeto.

Restrições:

$(D_i < D_j) \vee (D_i = D_j \wedge H_i + d_i < H_j), \forall (i,j) \in P$

$D_i \leq Concl, \forall i \in \{1, ..., n\}$

$H_i \in [8, 11] \vee [13, 16], \forall i \in \{1, ..., n\}$

$H_i + h_i \in [9, 12] \vee [14, 17], \forall i \in \{1, ..., n\}$

$H_i + h_i + min_{ij} \leq H_j \leq H_i + h_i + max_{ij}, \forall (i,j) \in I$

$y_{ije} \neq y_{i'je'}, \forall (i \neq i')$ tal que $D_i = D_{i'} \wedge H_i \in \{H_{i'} H_{i'} + d_{i'}\}$ para qualquer e, e'.

$\sum_{j=1}^{nt} y_{ije} = t_{ie}, \forall i \in \{1, ...n\}, \forall e \in \{1, ..., e\}$

$\sum_{e=1}^{ne} y_{ije} \leq 1, \forall i \in \{1, ...n\}, \forall j \in \{1, ..., nt\}$

$Concl \leq Prazo$

$y_{ije} \in \{0, 1\}$, com $i = \{1, .., n\}$, $j = \{1, ..., nt\}$ e $e = \{1, ..., ne\}$

$D_i \in N, \forall i \in \{1, ..., n\}$

Função objetivo: minimizar Concl.

2.1.2 Para minimizar o número de trabalhadores a contratar:

Dados:

n - número de tarefas do projeto.

nt - número de trabalhadores disponíveis.

ne - número de especialidades.

e_{ij} - trabalhador i tem habilitação na especialidade j.

t_{ij} - número de trabalhadores da especialidade j necessários para realizar a tarefa i.

h_i - duração da tarefa i, em horas.

$Prazo$ - data limite para conclusão do projeto.

P - relação que define a precedência entre tarefas.

I - relação que define o intervalo de tempo em duas tarefas.

min_{ij} - atraso mínimo da tarefa j depois do fim da tarefa i, para $(i,j) \in I$.

max_{ij} - atraso máximo da tarefa j depois do fim da tarefa i, para $(i, j) \in I$.

Variáveis de decisão:

D_i - dia em que a tarefa i vai ser executada.

H_i - hora de início da tarefa i.

y_{ije} - trabalhador j executa a tarefa i na especialidade e.

y_k - trabalhadores da especialidade k a contratar.

Restrições:

$(D_i < D_j) \vee (D_i = D_j \wedge H_i + d_i < H_j), \forall (i, j) \in P$

$D_i \leq Concl, \forall i \in \{1, ..., n\}$

$H_i \in [8, 11] \vee [13, 16], \forall i \in \{1, ..., n\}$

$H_i + h_i \in [9, 12] \vee [14, 17], \forall i \in \{1, ..., n\}$

$H_i + h_i + min_{ij} \leq H_j \leq H_i + h_i + max_{ij}, \forall (i, j) \in I$

$y_{ije} \neq t_{i'je'}, \forall (i \neq i')$ tal que $D_i = D_{i'} \wedge H_i \in \{H_{i'} H_{i'} + d_{i'}\}$, para qualquer e,e'

$\sum_{j=1}^{nt} y_{ije} + y_e = t_{ie}, \forall i \in \{1, ...n\}, \forall e \in \{1, ..., e\}$

$\sum_{e=1}^{ne} y_{ije} \leq 1, \forall i \in \{1, ...n\}, \forall j \in \{1, ..., nt\}$

$y_{ije} \in \{0, 1\}$, com $i = \{1, .., n\}$, $j = \{1, ..., nt\}$ e $e = \{1, ..., ne\}$

$D_i \in N, \forall i \in \{1, ..., n\}$

Função objetivo: minimizar $\sum_{k=1}^{ne} y_k$.

2.2 Implementação em ECLiPSe

2.2.1 Minimização dos dias do projeto

Começamos por verificar a sequência de dias úteis e o dia limite que corresponde ao dia equivalente á data do prazo, usando o predicado "build_calendar". Ficamos então com uma lista dos dias úteis. De seguida, usando o predicado "prec_days_constrs"vamos limitar as horas possíveis de começo de tarefas que são precedidas por outras. De referir que o número de horas vai desde 0, que corresponde ás 8 do primeiro dia até 8 * dia limite, que corresponde ás 17 horas do última dia. De seguida, fazemos restrições sobre os intervalos de tempo em que uma tarefa tem começar após a outra, se essa relação existir no conjunto I definido em cima. Fazemos isso usando o predicado "interval_constrs",

onde para cada tarefa T, vemos se existem factos "intervalo(Idt,Idtprev,Min,Max)", onde Idt = T, e se existirem vamos restringir o intervalo de tempo de começo da tarefa T conforme o indicado. Após todas estas restrições vamos criar uma lista de 3 dimensões, que corresponde ao y_{ije} indicado nas variáveis de decisão. Usando o predicado "build_3d"criamos a lista, que vai ter tantas sublistas quanto tarefas existirem. Cada tarefa tem uma lista para cada trabalhador que existe, que por sua vez tem uma lista para cada especialidade existente. Cada posição corresponde a um trabalhador fazer parte de uma equipa que realiza a tarefa, com uma certa especialidade. Depois disto, são feitas restrições sobre a essa mesma lista, usando o predicado "workers_constrs"que podemos ver no código em baixo. No predicado "limit_domain", para cada trabalhador de cada tarefa, vamos percorrer a lista de especialidades que esse trabalhador tem e as especialidades que a tarefa atual precisa. Se uma especialidade estiver em ambos então a posição correspondente na lista pode ser 0 ou 1, ou seja o trabalhador pode ou não pertencer realizar a especialidade se pertencer á equipa que realiza a tarefa. Se a especialidade não estiver na lista de especialidades que esse trabalhador tem ou nas especialidades que a tarefa atual precisa, então a posição na lista vai ter que ser 0. Com o predicado "limit_ocurrences"vamos,para cada tarefa, fazer com que na lista de cada trabalhador só exista no máximo um "1", ou seja, que se o trabalhador pertencer á equipa que realiza a tarefa, só realiza uma especialidade. Por fim, vamos garantir que para cada tarefa o número de 1's é igual ao número total de trabalhadores necessários para realizar a tarefa.

```
workers_constrs(_,[],_,_).
workers_constrs([TiL|TWMatrix], [T|OTasksL], OWorkers, ListSpecialties) :-
              tarefa(T,_,_,WL,_),
              limit_domain(TiL,WL,OWorkers,ListSpecialties),
              count_workers(WL, WorkersN),
              limit_ocurrences(TiL),fill_domain(TiL),flatten(TiL,TiL2),occurrences(1,
                  TiL2, WorkersN),
              workers_constrs(TWMatrix,OTasksL,OWorkers,ListSpecialties).
```

Por fim, vamos fazer com que se o tempo em que duas tarefas se realizam se intersectar, então os trabalhadores que fazem parte das equipas que realizam as duas tarefas são todos distintos. Para isso, no predicado "check_sobr", vamos percorrer todas as tarefas e para cada tarefa vemos se a sua realização intersecta a realização de outras tarefas. Se intersectar, vamos buscar as listas correspondentes a cada tarefa e concatena-las. Depois verificamos se o número de uns na lista resultante é igual á soma do número de trabalhadores necessário para cada tarefa.

2.3 Resultados

Por razões que explicaremos na secção de melhoramentos, não conseguimos obter resultados significativos.

2.4 Possíveis melhoramentos

Um dos melhoramentos, seria limitar as horas de cada tarefa, para garantir que todas as tarefas ocorrem em períodos de tempo contínuos. Seria uma ideia parecida com o código apresentado em baixo, onde iríamos tentar limitar as tarefas a períodos de 4 horas contínuos. X teria como valores possíveis $[0, L]$, onde L seria o número máximo de horas possível a dividir por 4, ou seja, iríamos obter todos os sub-intervalos de 4 possíveis.

```
hour_constrs([],_,_).
hour_constrs([T|RTarefas], Hours, X) :-
    tarefa(T,_,Hi,_,_),
    selec_elemento(1,T,Hours,HourI),
    ((Hi = 4, HourI #= X * 4);(Hi = 3, HourI#>=X*4, HourI #=<(X*4+1));
    (Hi = 2, HourI#>=X*4, HourI #=<(X*4+2));(HourI#>=X*4, HourI #=<(X*4+3))),
    hour_constrs(RTarefas, Hours, X).
```

Outro melhoramento, seria para obter o número de trabalhadores a contratar se não fosse possível acabar o projeto antes da data limite. Para isso, iríamos ter que percorrer a lista de cada tarefa e se o número de 1's fosse menor do que o necessário adicionar um trabalhador da categoria que faltasse. Para tarefas em que o tempo de execução se intersectasse teríamos de contratar trabalhadores diferentes, caso fossem necessários trabalhadores para as mesmas especialidades, mas, caso contrário, poderíamos aproveitar os trabalhadores já contratados se fossem necessários mais trabalhadores para a mesma especialidade.

Referências

[1] http://eclipseclp.org/doc/bips/lib/branch_and_bound/minimize-2.html

[2] http://eclipseclp.org/doc/bips/lib/ic/labeling-1.html

[3] http://eclipseclp.org/doc/bips/lib/ic/get_bounds-3.html

[4] http://eclipseclp.org/doc/bips/lib/ic_cumulative/cumulative-4.html

[5] http://eclipseclp.org/doc/bips/lib/ic/search-6.html

[6] https://en.wikipedia.org/wiki/Critical_path_method