Resolução de Problema de Decisão usando Programação em Lógica com Restrições : Centro de Computação

Diogo Yaguas[up201606165] | Eduardo Silva[up201603135]

Faculdade de Engenharia da Universidade do Porto Rua Dr. Roberto Frias, s/n
4200-465 Porto Portugal

Resumo O projeto desenvolvido, no Sistema de Desenvolvimento SICStus Prolog no âmbito da unidade curricular de Programação em Lógica, tem como objetivo resolver um problema de decisão/otimização implementando restrições. O problema de otimização escolhido foi o Centro de Computação que tem como finalidade otimizar um método de alocação de tarefas a servidores, de forma a maximizar o débito de tarefas.

Keywords: FEUP · Prolog · SICStus · Otimização · Restrições .

1 Introdução

O projeto foi desenvolvido no âmbito da unidade curricular de Programação em Lógica de 3º ano do curso Mestrado Integrado em Engenharia Informática e de Computação.

Para tal, foi necessário implementar uma possível resolução para um problema de decisão ou otimização em Prolog, com restrições. O grupo escolheu um problema de otimização, denominado por Centro de Computação.

O problema de otimização escolhido consiste na otimização de um método de alocação de tarefas a servidores, de forma a maximizar o débito de tarefas, ou seja, as tarefas são distribuídas pelos servidores tendo em conta as suas especificações e os planos de pagamento, de forma a minimizar o tempo de custo.

Este artigo tem a seguinte estrutura:

1. **Descrição do Problema:** descrição com detalhe o problema de otimização ou decisão em análise.
2. **Abordagem:** descrição da modelação do problema como um PSR, de acordo com as seguintes subsecções.
   - **Variáveis de Decisão:** descrição das variáveis de decisão e os seus domínios.
   - **Restrições:** descrição das restrições rígidas e flexíveis do problema e a sua implementação utilizando o SICStus Prolog.
   - **Função de Avaliação:** descrição da forma de avaliar a solução obtida e a sua implementação utilizando o SICStus Prolog.
   - **Estratégia de Pesquisa:** descrição da estratégia de etiquetagem (labeling) utilizada ou implementada, nomeadamente no que diz respeito a ordenação de variáveis e valores.
3. **Visualização da Solução:** explicação dos predicados que permitem visualizar a solução em modo de texto.
4. **Resultados:** demonstração de exemplos de aplicação em instâncias do problema com diferentes complexidades e análise dos resultados obtidos.
5. **Conclusões e Trabalho Futuro:** conclusões retiradas deste projeto, resultados obtidos, vantagens e limitações da solução proposta, aspetos a melhorar.

2 Descrição do problema

O Centro de Computação tem como objetivo otimizar o seu método de alocação de tarefas a servidores, de forma a maximizar o débito de tarefas, visto que se trata de um problema de otimização.

O problema possui vários servidores, sendo cada um caracterizado pelos seus componentes, *CPU (número de cores e frequência), Memória e Disco*.

Os utilizadores enviam tarefas para execução nos seus servidores, indicando para cada tarefa as suas necessidades em termos de *CPU, RAM, Disco* e duração esperada da tarefa. A tarefa pode ser concluída num período de tempo maior ou menor consoante a velocidade efetiva do *CPU* alocado.

Os clientes possuem contratos diferenciados, num total de 4 planos, tendo planos de pagamento superiores direito a maior prioridade na execução das suas tarefas.

A alocação de tarefas a servidores é realizada diariamente tendo em conta as características das tarefas a realizar e os servidores disponíveis, assim como as prioridades dos diferentes clientes.

3 Abordagem

De forma a resolver este problema em linguagem *Prolog*, foram tomados em conta dois aspetos, os servidores e as tarefas. Em ambos os casos foram utilizadas listas para os representar e guardar informação sobre eles.

Inicialmente é feito uma separação de cada tarefa e de cada servidor em quatro sub-tarefas e quatro sub-servidores, cada um para um tipo de componente (número de *cores*, frequência, *RAM* e capacidade). O objetivo é usar quatro chamadas ao predicado *cumulatives*. A partir destes quatro componentes são feitos os cálculos necessários para que o tempo de execução de cada tarefa seja mínimo.

3.1 Variáveis de Decisão

A solução do problema vem na forma de três listas: uma para representar a lista de servidores que estão associados às tarefas, a lista de tempo inicial de execução da tarefa, a lista de tempo final de execução da tarefa.

Todas estas listas estão ordenadas de acordo com o grau de prioridade da tarefa em questão. Os domínio das listas correspondentes aos tempos são de 0 até 86400 (24 horas). Já o domínio da lista de servidores vai de 1 até o número de servidores.

3.2 Restrições

Os recursos de uma tarefa têm de ser menores que as dos servidores. Como cada tarefa é constituída por um conjunto de recursos, esses mesmos têm de ser menores que o limite do recurso da máquina com maior limite, tratando assim de uma restrição rígida. Esta restrição é efetuada em cada componente pelo *cumulatives*.

As tarefas têm uma prioridade associada. Cada cliente têm um contrato associado, sendo que os que têm um plano de pagamento superior (Plano 4), têm maior prioridade sobre aqueles que têm um plano de pagamento inferior (Plano 1). Este Planos variam entre 1 e 4, sendo que quanto maior o valor, maior a prioridade de execução da tarefa. Esta restrição é efetuada ordenando a lista de tarefas de forma decrescente, isto é, da tarefa de maior prioridade para a de menor prioridade, logo nos casos em que o tempo para efetuar cada tarefa não tem influência, as tarefas de maior prioridade vão ser realizadas primeiro.

O tempo de execução de um tarefa depende da velocidade do CPU da máquina e do consumo da tarefa. Como se trata de uma restrição flexível, calcula-se o tempo de execução, tendo por base, o *ETA* da tarefa, no predicado *create_prolog_tasks(TaskList, StartTimes, EndTimes, MachineIds, CoresList, FreqList, Tasks1, Tasks2, Tasks3, Tasks4).*

```
create_prolog_tasks([UserTask | Rtl], [StartTime | Rs], [EndTime | Re], [MachineId| Rmid], CoresList, FreqList
    [Task1 | Rt1], [Task2 | Rt2], [Task3 | Rt3], [Task4 | Rt4]) :-
    UserTask = [_, _, NoCores, Frequency, RAM, Storage, ETA],
    TaskCost is (NoCores * Frequency),
    element(MachineId, CoresList, CoreS),
    element(MachineId, FreqList, FreqS),
    CPUSpeed #= (CoreS * FreqS),
    Diff #= (CPUSpeed - TaskCost),
    (Diff #= 0 #/\ CleanDiff #= 1) #\/ (Diff #\= 0 #/\ CleanDiff #= Diff),
    HalfwayPoint #= (CPUSpeed / 2),
    (CleanDiff #>= HalfwayPoint #/\ Duration #= (ETA / (CleanDiff / TaskCost)))
    #\/ (CleanDiff #< HalfwayPoint #/\ Duration #= (ETA * (TaskCost / CleanDiff))),
    Task1 = task(StartTime, Duration, EndTime, NoCores, MachineId),
    Task2 = task(StartTime, Duration, EndTime, Frequency, MachineId),
    Task3 = task(StartTime, Duration, EndTime, RAM, MachineId),
    Task4 = task(StartTime, Duration, EndTime, Storage, MachineId),
    create_prolog_tasks(Rtl, Rs, Re, Rmid, CoresList, FreqList, Rt1, Rt2, Rt3, Rt4).
```

Figura 1. Predicado que cria as tarefas

Todas as sub-tarefas pertencentes a uma tarefa terão o mesmo ID de uma máquina. É necessário garantir que todos recursos de uma tarefa são possíveis numa mesma máquina, para isso é utilizado o predicado *create_machines(ServerList, 1, CoresList, FreqList, Machines1, Machines2, Machines3, Machines4)* em que o ID das máquinas criadas é o mesmo, de forma a assegurar que são atribuídas ao mesmo servidor, sendo assim uma restrição rígida.

```
create_machines([Server | Rsl], MachineId, [NoCores | Rc], [Frequency | Rf], [M1 | Rm1], [M2 | Rm2], [M3 | Rm3],
    [M4 | Rm4]) :-
    Server = [NoCores, Frequency, RAM, Storage],
    M1 = machine(MachineId, NoCores),
    M2 = machine(MachineId, Frequency),
    M3 = machine(MachineId, RAM),
    M4 = machine(MachineId, Storage),
    NextMachineId is (MachineId + 1),
    create_machines(Rsl, NextMachineId, Rc, Rf, Rm1, Rm2, Rm3, Rm4).
```

Figura 2. Predicado que cria os servidores

3.3 Função de Avaliação

Como este problema se trata de um problema de otimização, não é só preciso garantir que se encontra uma solução como também é necessário encontrar a melhor solução. Para isso implica que as tarefas sejam executadas no menor tempo possível, nos servidores que assim o permitem.

É possível que várias tarefa sejam realizadas ao mesmo tempo no mesmo servidor, desde que, os recursos disponibilizados pelo servidor cubram o consumo de cada tarefa na totalidade. Caso isto não seja possível, a tarefa é alocada ao servidor que irá executar a tarefa no menor tempo possível, mesmo que isso implique esperar por uma tarefa que se encontra em execução.

Inicialmente é feito o cálculo da velocidade do *CPU*, multiplicando o número de *Cores* pela frequência do *CPU*. De seguida é calculado o consumo da tarefa, multiplicando os mesmo componentes. É calculada a diferença entre a velocidade do *CPU* e o consumo da tarefa. Caso esta diferença seja maior que o ponto médio da velocidade do *CPU*, a tempo de execução da tarefa será igual ao *ETA* da tarefa a dividir pela divisão da diferença pelo consumo da tarefa. Caso contrário será a multiplicação do *ETA* da tarefa pela divisão do consumo da tarefa pela diferença.[1]

Começa-se por executar as que têm maior prioridade, seguidas das de menor prioridade tendo em conta todos recursos que a tarefa consome e as que os servidores conseguem fornecer.

3.4 Estratégia de Pesquisa

Foram testadas várias opções de pesquisa para a otimização deste problema. Para isso, foi usada sempre a mesma quantidade de tarefas e de servidores, 3 tarefas para 3 servidores. Em anexo, na figura 6, estão representados os dados. É possível concluir que a melhor estratégia de pesquisa é a a utilização das opções *bisect max*, enquanto que a pior estratégia é a utilização das opções *enum ffc*.

[1] Devido à não aceitação de valores de em vírgula flutuante em *clpfd*, a diferença entre o tempo estimado e a tempo de execução da tarefa será considerável.

4 Visualização da Solução

Este programa permite resolver o problema de otimização do método de alocação de tarefas a servidores, de forma a maximizar o débito de tarefas. Para melhor visualização do problema existem quatro predicados que ajudam a sua solução.

De forma a que o problema seja instanciado é necessário inserir na consola o predicado ***ceco***, sem argumentos. Este predicado dá a opção ao utilizador de inserir os componentes dos servidores e das tarefas, através do predicado **manual_input(ServerList, ServerNo, TaskList, TaskNo)**, ou que seja gerado uma lista de servidores e de tarefas do tamanho que o utilizador queira, através do predicado **generate_data(ServerList, ServerNo, TaskList, TaskNo)**.

```
ceco :-
    display_banner,
    write('1 - Manual Input'), nl,
    write('2 - Generate Data'), nl,
    get_option(Option, 1, 2),
    (
        (Option = 1, manual_input(ServerList, ServerNo, TaskList, TaskNo));
        (Option = 2, generate_data(ServerList, ServerNo, TaskList, TaskNo),
        print_data(ServerList, ServerNo, TaskList))
    ), !,
    samsort(compare_tasks, TaskList, NewTaskList),
    schedule(ServerList, ServerNo, NewTaskList, TaskNo, StartTimes, EndTimes, MachineIds),
    print_results(NewTaskList, MachineIds, StartTimes, EndTimes), nl.
```

Figura 3. Predicado principal

Após resolução do problema, é mostrado na consola, através do predicado **print_results(NewTaskList, MachineIds, StartTimes, EndTimes)**, o ID da tarefa, o servidor a que está associada, o tempo de iniciação e o tempo de finalização da tarefa.

```
print_results([[TaskId | _] | Rt], [MachineId | Rmid], [StartTime | Rst], [EndTime | Ret]) :-
    StartTimeMins is (StartTime // 60),
    EndTimeMins is (EndTime // 60),
    write('Task #'), write(TaskId), write(' was associated with server #'), write(MachineId), write(', starting at '),
    write(StartTimeMins), write(' minutes and ending at '), write(EndTimeMins),  write(' minutes.'),  nl,
    print_results(Rt, Rmid, Rst, Ret).
```

Figura 4. Predicado que permite visualizar os resultados

```
Task #3 was associated with server #1, starting at 0 minutes and ending at 76 minutes.
Task #2 was associated with server #3, starting at 0 minutes and ending at 186 minutes.
Task #1 was associated with server #2, starting at 0 minutes and ending at 125 minutes.
```

Figura 5. Exemplo dos resultados

5 Resultados

De forma a se retirar conclusões sobre os resultados obtidos foram medidos os tempos de resolução, o número de retrocessos e o número de restrições criadas.

Variação do número de tarefas com número de servidores constantes: Tanto o número de retrocessos como o número de restrições criadas variam de forma linear. Por outro lado a variação do tempo é pouco conclusiva tendo em conta a diferença de tempo, tornando-se praticamente constante. (Figura 7 - 10)

Variação do número de servidores com número de tarefas constantes: Como no caso anterior, tanto o número de retrocessos como o número de restrições criadas variam de forma linear. Mais uma vez, a variação do tempo é pouco conclusiva tendo em conta a diferença de tempo, tornando-se praticamente constante. (Figura 11 - 14)

O número de restrições criadas mantém-se constante quando o número de tarefas é o mesmo, isto deve-se ao facto de quê as restrições serem aplicadas apenas às tarefas e não aos servidores.

Como o crescimento das restrições e de retrocessos é linear, a diferença do tempo de execução da tarefa é praticamente igual, independentemente do número de tarefas ou do número de servidores.

6 Conclusões e Trabalho Futuro

Ao realizar este projeto, tivemos oportunidade de aplicar os conhecimentos adquiridos nas aulas teóricas e práticas, concluindo que a linguagem Prolog, mais em particular, o módulo de restrições, é bastante útil para determinadas situações, como por exemplo, na resolução de problemas de decisão e de otimização.

O projeto foi concluído com sucesso, visto que soluciona o problema inicial de otimizar um método de alocação de tarefas a servidores, de forma a maximizar o débito de tarefas. O seu desenvolvimento contribuiu para uma melhor compreensão do funcionamento de *labeling* e variáveis de decisão, assim como na aplicação de restrições.

7 Anexos

	leftmost	min	max	first_fail	anti_first_fail	occurrence	ffc	max_regret	most_constrained
step	0,19	0,02	0,02	0,12	0,02	0,18	0,12	0,18	0,12
enum	2,2	0,08	0,08	1,62	0,08	2,08	1,78	2,08	1,64
bisect	0,02	0,03	0,02	0,03	0,02	0,02	0,02	0,03	0,03
median	1,29	0,03	0,02	1,77	0,03	1,42	1,66	1,39	1,08
middle	0,47	0,02	0,02	0,95	0,03	0,51	0,92	0,48	0,6

Figura 6. Testes de estratégia de pesquisa

Numero de servidores: 3			
Número de tarefas	Tempo(s)	Retrocessos	Restrições criadas
1	0,02	20	31
2	0,01	41	59
3	0,02	62	86
4	0,02	83	128
5	0,03	104	172

Figura 7. Variação de tarefas com número de servidores constantes

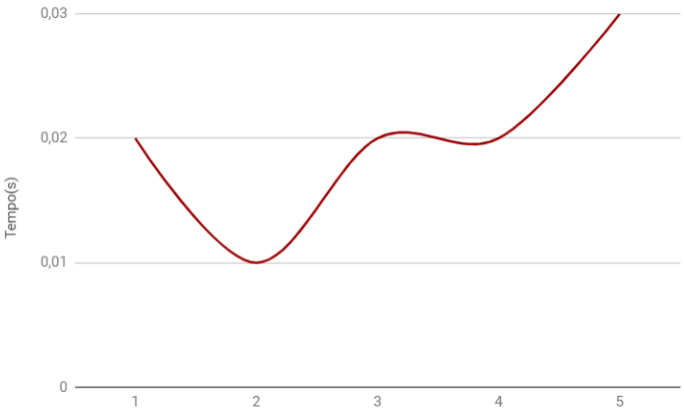

Figura 8. Variação do tempo em função do número de tarefas

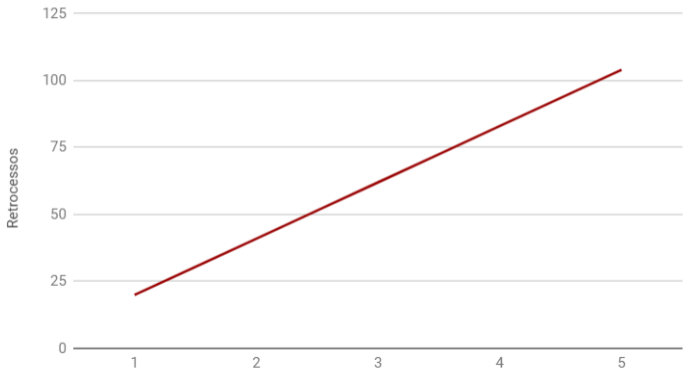

Figura 9. Variação do número de retrocessos em função do número de tarefas

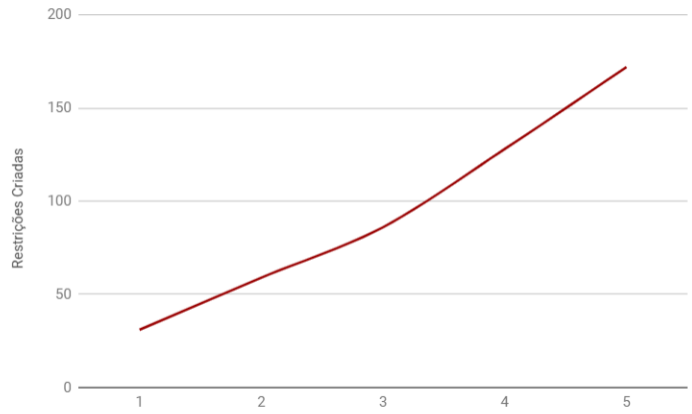

Figura 10. Variação do número de restrições criadas em função do número de tarefas

Numero de tarefas: 3			
Número de servidores	**Tempo(s)**	**Retrocessos**	**Restrições criadas**
1	**0,01**	**57**	**80**
2	**0,01**	**53**	**86**
3	**0,01**	**62**	**86**
4	**0,01**	**66**	**86**
5	**0,02**	**80**	**86**

Figura 11. Variação de servidores com número de tarefas constantes

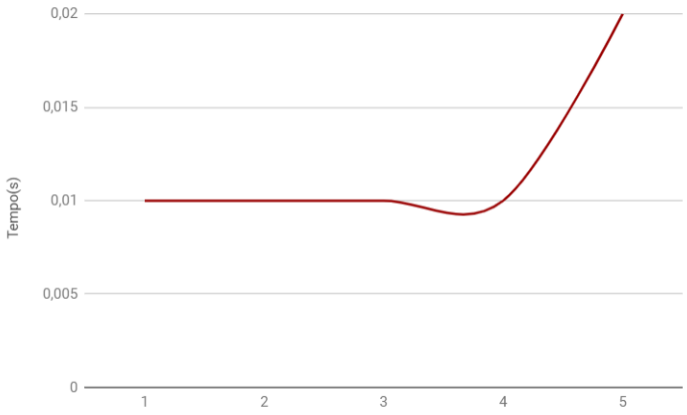

Figura 12. Variação do tempo em função do número de servidores

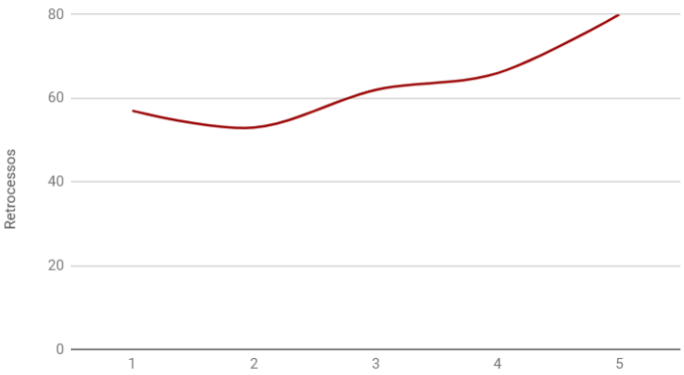

Figura 13. Variação do número de retrocessos em função do número de servidores

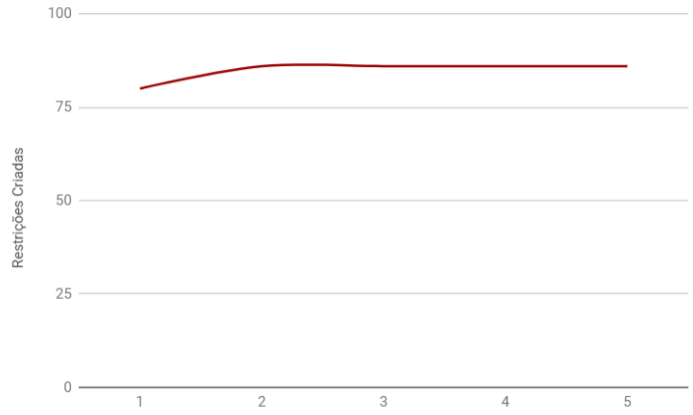

Figura 14. Variação do número de restrições criadas em função do número de servidores

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%% computing_centre.pl %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- use_module(library(clpfd)).
:- use_module(library(lists)).
:- use_module(library(samsort)).
:- use_module(library(random)).

:- include('tools.pl').
:- include('interface.pl').

% Starter function
ceco :-
    display_banner,
    write('1 - Manual Input'), nl,
    write('2 - Generate Data'), nl,
    get_option(Option, 1, 2),
    (
        (Option = 1, manual_input(ServerList, ServerNo, TaskList, TaskNo));
        (Option = 2, generate_data(ServerList, ServerNo, TaskList, TaskNo),
        print_data(ServerList, ServerNo, TaskList))
    ), !,
    samsort(compare_tasks, TaskList, NewTaskList),
    schedule(ServerList, ServerNo, NewTaskList, TaskNo, StartTimes, EndTimes,
MachineIds),
    print_results(NewTaskList, MachineIds, StartTimes, EndTimes), nl.

% Scheduling function, takes into account all the machines and tasks and their
respective parameters
schedule(ServerList, ServerNo, TaskList, TaskNo, StartTimes, EndTimes, MachineIds)
:-
    length(StartTimes, TaskNo),
    length(EndTimes, TaskNo),
    domain(StartTimes, 0, 86400),
    domain(EndTimes, 0, 86400),
    %Machines
    length(Machines1, ServerNo),
    length(Machines2, ServerNo),
    length(Machines3, ServerNo),
    length(Machines4, ServerNo),
    create_machines(ServerList, 1, CoresList, FreqList, Machines1, Machines2,
Machines3, Machines4),
    %Tasks
    length(MachineIds, TaskNo),
    domain(MachineIds, 1, ServerNo),
    length(Tasks1, TaskNo),
    length(Tasks2, TaskNo),
    length(Tasks3, TaskNo),
    length(Tasks4, TaskNo),
    create_prolog_tasks(TaskList, StartTimes, EndTimes, MachineIds, CoresList,
FreqList, Tasks1, Tasks2, Tasks3, Tasks4),
    cumulatives(Tasks1, Machines1, [bound(upper)]),
    cumulatives(Tasks2, Machines2, [bound(upper)]),
    cumulatives(Tasks3, Machines3, [bound(upper)]),
    cumulatives(Tasks4, Machines4, [bound(upper)]),
    %write('Passed cumulatives'), nl,
    maximum(End, EndTimes),
    domain([End], 0, 86400),
    append([MachineIds, StartTimes], Vars),
```

```
    labeling([minimize(End), bisect, max], Vars).

% Creates the prolog machines, 4 for each server
create_machines([], _, [], [], [], [], [], []).
create_machines([Server | Rsl], MachineId, [NoCores | Rc], [Frequency | Rf], [M1 | Rm1], [M2 | Rm2], [M3 | Rm3],
    [M4 | Rm4]) :-
    Server = [NoCores, Frequency, RAM, Storage],
    M1 = machine(MachineId, NoCores),
    M2 = machine(MachineId, Frequency),
    M3 = machine(MachineId, RAM),
    M4 = machine(MachineId, Storage),
    NextMachineId is (MachineId + 1),
    create_machines(Rsl, NextMachineId, Rc, Rf, Rm1, Rm2, Rm3, Rm4).

% Creates the prolog tasks, 4 for each task
create_prolog_tasks([], [], [], [], _, _, [], [], [], []).
create_prolog_tasks([UserTask | Rtl], [StartTime | Rs], [EndTime | Re], [MachineId| Rmid], CoresList, FreqList,
    [Task1 | Rt1], [Task2 | Rt2], [Task3 | Rt3], [Task4 | Rt4]) :-
    UserTask = [_, _, NoCores, Frequency, RAM, Storage, ETA],
    TaskCost is (NoCores * Frequency),
    element(MachineId, CoresList, CoreS),
    element(MachineId, FreqList, FreqS),
    CPUSpeed #= (CoreS * FreqS),
    Diff #= (CPUSpeed - TaskCost),
    (Diff #= 0 #/\ CleanDiff #= 1) #\/ (Diff #\= 0 #/\ CleanDiff #= Diff),
    HalfwayPoint #= (CPUSpeed / 2),
    (CleanDiff #>= HalfwayPoint #/\ Duration #= (ETA / (CleanDiff / TaskCost)))
    #\/ (CleanDiff #< HalfwayPoint #/\ Duration #= (ETA * (TaskCost / CleanDiff))),
    Task1 = task(StartTime, Duration, EndTime, NoCores, MachineId),
    Task2 = task(StartTime, Duration, EndTime, Frequency, MachineId),
    Task3 = task(StartTime, Duration, EndTime, RAM, MachineId),
    Task4 = task(StartTime, Duration, EndTime, Storage, MachineId),
    create_prolog_tasks(Rtl, Rs, Re, Rmid, CoresList, FreqList, Rt1, Rt2, Rt3, Rt4).

% Function used to compare the tasks in a task list in order to sort them by plan
compare_tasks([_, Plan1 | _], [_, Plan2 | _]) :-
    Plan1 > Plan2.

% Generates the servers and tasks automatically
generate_data(ServerList, NoServers, TaskList, TaskNo) :-
    write('Server Amount: '),
    get_clean_int(NoServers), nl,
    generate_servers(NoServers, ServerList),
    write('Number of Tasks: '),
    get_clean_int(TaskNo), nl,
    AvgTime is (86400 div TaskNo),
    generate_tasks(TaskNo, 1, AvgTime, TaskList).

% Generates the user tasks
generate_tasks(TaskNo, TaskId, _, []) :-
    TaskId is (TaskNo + 1).

generate_tasks(NoTasks, TaskId, AvgTime, [Task | RestOfTaskList]) :-
    random(1, 4, Plan),
    random(1, 4, Cores),
    random(1, 2, Frequency),
    random(1, 8, RAM),
    random(1, 122, Storage),
```

```
    random(1, AvgTime, ETAHours),
    Task = [TaskId, Plan, Cores, Frequency, RAM, Storage, ETAHours],
    NextTaskId is (TaskId + 1),
    generate_tasks(NoTasks, NextTaskId, AvgTime, RestOfTaskList).

% Generates the servers
generate_servers(0, []).
generate_servers(NoServers, [Server | RestOfServerList]) :-
    random(4, 8, Cores),
    random(2, 3, Frequency),
    random(8, 16, RAM),
    random(122, 1000, Storage),
    NoServersAux is (NoServers - 1),
    Server = [Cores, Frequency, RAM, Storage],
    generate_servers(NoServersAux, RestOfServerList).

```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% interface.pl %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Displays the main menu banner
display_banner :-
    write('\e[2J'),
    write('-------'), nl,
    write(' CeCo'), nl,
    write('-------'), nl.

% Prints the final results in a user friendly way
print_results([], [], [], []).
print_results([[TaskId | _] | Rt], [MachineId | Rmid], [StartTime | Rst], [EndTime | Ret]) :-
    StartTimeMins is (StartTime // 60),
    EndTimeMins is (EndTime // 60),
    write('Task #'), write(TaskId), write(' was associated with server #'), write(MachineId), write(', starting at '),
    write(StartTimeMins), write(' minutes and ending at '), write(EndTimeMins), write(' minutes.'),  nl,
    print_results(Rt, Rmid, Rst, Ret).

% Prints the data generated automatically
print_data(ServerList, ServerNo, TaskList) :-
    print_servers(ServerList, ServerNo, 1), nl,
    print_tasks(TaskList), nl.

% Prints the tasks generated
print_tasks([]).
print_tasks([Task | Rt]) :-
    Task = [TaskId, Plan, NoCores, Frequency, RAM, Storage, ETA],
    ETAmins is (ETA // 60),
    write('-----------'), nl,
    write('Task #'), write(TaskId), nl,
    write('-----------'), nl,
    write('Client Plan: '), write(Plan), nl,
    write('Number of Cores: '), write(NoCores), nl,
    write('Frequency (GHz): '), write(Frequency), nl,
    write('RAM (GB): '), write(RAM), nl,
    write('Storage (GB): '), write(Storage), nl,
    write('ETA (mins): '), write(ETAmins), nl,
    print_tasks(Rt).

% Prints the servers generated
print_servers([], ServerNo, Counter) :-
    Counter is (ServerNo + 1).
print_servers([Server | Rs], ServerNo, Counter) :-
    Server = [NoCores, Frequency, RAM, Storage],
    write('-----------'), nl,
    write('Server #'), write(Counter), nl,
    write('-----------'), nl,
    write('Number of Cores: '), write(NoCores), nl,
    write('Frequency (GHz): '), write(Frequency), nl,
    write('RAM (GB): '), write(RAM), nl,
    write('Storage (GB): '), write(Storage), nl,
    NextCounter is (Counter + 1),
    print_servers(Rs, ServerNo, NextCounter).

% Retrieves the input from the user
```

```
manual_input(ServerList, NoServers, TaskList, TaskNo) :-
    write('Server Amount: '),
    get_clean_int(NoServers), nl,
    ServerAux is (NoServers + 1),
    create_servers(NoServers, ServerAux, ServerList),
    write('Task Amount: '),
    get_clean_int(TaskNo), nl,
    length(TaskList, TaskNo),
    get_tasks(TaskNo, ServerList, TaskList).

% Creates the servers with inputes from the user
create_servers(0, _, []).
create_servers(NoServers, ServerAux, [Server | RestOfServerList]) :-
    ServerNumber is (ServerAux - NoServers),
    write('-----------'), nl,
    write('Server #'), write(ServerNumber), nl,
    write('-----------'), nl,
    write('Number of Cores: '), get_clean_int(NoCores), nl,
    write('Frequency (GHz): '), get_clean_int(Frequency), nl,
    write('RAM (GB): '), get_clean_int(RAM), nl,
    write('Storage (GB): '), get_clean_int(Storage), nl,
    Server = [NoCores, Frequency, RAM, Storage],
    NoServersAux is (NoServers - 1),
    create_servers(NoServersAux, ServerAux, RestOfServerList).

% Creates the tasks with inputs from the user
get_tasks(TaskNo, ServerList, TaskList) :-
    get_task_list(1, TaskNo, TaskList),
    check_tasks_compatibility(ServerList, TaskList, 0).

get_tasks(TaskNo, ServerList, TaskList) :-
    write('2nd Pass'), nl,
    get_tasks(TaskNo, ServerList, TaskList).

% Checks if the tasks are accepted by at least one server and that their total time
doesn't exceed 24h
check_tasks_compatibility(_, [], TotalTime) :-
    (TotalTime =< 86400);
    (write('Total task time must not exceed 24 hours (1440 mins)'), nl, fail).

check_tasks_compatibility(ServerList, [Task | Rt], TotalTime) :-
    (
        check_server_compatibility(ServerList, Task),
        Task = [_, _, _, _, _, _, TaskTime],
        AccumulatedTime is (TotalTime + TaskTime),
        check_tasks_compatibility(ServerList, Rt, AccumulatedTime)
    );
    (write('There is at least one task that cannot be serviced by any server!'), nl,
fail).

% Checks if there is at least one server able to process a given task
check_server_compatibility([], _) :-
    fail.
check_server_compatibility([Server | Rs], Task) :-
    Server = [NoCoresS, FrequencyS, RAMS, StorageS],
    Task = [_, _, NoCoresT, FrequencyT, RAMT, StorageT, _],
    ((NoCoresS >= NoCoresT, FrequencyS >= FrequencyT, RAMS >= RAMT, StorageS >=
StorageT);
    check_server_compatibility(Rs, Task)).

```

```
% Creates a task list with inputs from the user
get_task_list(Counter, TaskNo, []) :-
    Counter is (TaskNo + 1).

get_task_list(Counter, TaskNo, [Task | Rt]) :-
    write('-----------'), nl,
    write('Task #'), write(Counter), nl,
    write('-----------'), nl,
    write('Client Plan (1,2,3 or 4): '), get_option(Plan, 1, 4), nl,
    write('Number of Cores: '), get_clean_int(NoCores), nl,
    write('Frequency (GHz): '), get_clean_int(Frequency), nl,
    write('RAM (GB): '), get_clean_int(RAM), nl,
    write('Storage (GB): '), get_clean_int(Storage), nl,
    write('ETA (mins): '), get_clean_int(ETAMins), nl,
    ETASeconds is (ETAMins * 60),
    Task = [Counter, Plan, NoCores, Frequency, RAM, Storage, ETASeconds],
    NextCounter is (Counter + 1),
    get_task_list(NextCounter, TaskNo, Rt).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% tools.pl %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Reads an option from the console as an int and checks if it is within the bounds
stipulated.
get_option(Opt, LowerBound, UpperBound) :-
    get_clean_int(Opt),
    Opt >= LowerBound,
    Opt =< UpperBound.

get_option(Opt, LowerBound, UpperBound) :-
    write('Invalid Option'), nl,
    get_option(Opt, LowerBound, UpperBound).

% Reads a char from the input stream, discarding everything after it
get_clean_char(X) :-
    get_char(X),
    read_line(_),
    nl.

% Reads an int from the input stream, verifying if it's actually an int
get_clean_int(I) :-
    catch(read(I), _, true),
    get_char(_),
    integer(I).

get_clean_int(I) :-
    write('<<< Invalid Input >>>\n\n'), get_clean_int(I).

% Reads a number from the console.
get_clean_number(N) :-
    catch(read(N), _, true),
    get_char(_).

get_clean_number(N) :-
    write('<<< Invalid Input >>>\n\n'), get_clean_number(N).

% Prints a list
print_list([]) :-
    !.

print_list([H|T]) :-
    write(H),
    write('|'),
    print_list(T).
```