Pesquisa aplicada à gestão de projetos

Universidade do Porto

Faculdade de Engenharia

FEUP

Inteligência Artificial

3^o ano 2^o semestre
Mestrado Integrado em Engenharia Informática e Computação

Turma 5 - Grupo A1_1

Hugo Drumond
hugo.drumond@fe.up.pt
ee11247 (up201102900)

22 de Maio de 2018

Conteúdo

1 Objetivo **2**

2 Especificação **2**
2.1 Descrição 2
2.2 Análise do Problema 2
2.2.1 Cenários 3
2.2.2 Representação do Conhecimento 4
2.3 Abordagem 5
2.3.1 Estados 5
2.3.2 Função de Transição 6
2.3.3 Função de custo 7

3 Desenvolvimento **8**
3.1 Arquitetura da aplicação 9
3.2 Detalhes Relevantes da implementação 10

4 Experiências **10**
4.1 Resultados das Experiências 10
4.1.1 Experiência 1a.json 10
4.1.2 Experiência 1b.json 11
4.1.3 Experiência 2a.json 11
4.1.4 Experiência 2b.json 11
4.1.5 Experiência 3a.json 11
4.1.6 Experiência 3b.json 11
4.2 Análise heurística 12
4.3 Visualização exemplo, experiência 1a.json 12

5 Conclusões **13**

6 Melhoramentos **13**

7 Recursos **13**
7.1 Websites 13
7.2 Software 13

1 Objetivo

Desenvolver um projeto sobre Pesquisa Sistemática e Informada de Soluções para Gestão de Projetos. Com a finalidade de determinar a melhor associação de elementos a uma equipa de projeto, maximizando o desempenho global. Mais concretamente a maximização da soma dos desempenhos das competências dos elementos na atribuição às tarefas de forma a reduzir o tempo total do projeto. Para alcançarmos tal, utilizou-se a seguinte estratégia:

- Implementação do algoritmo A*;
- Visualização gráfica para debug da solução;
- Preparação de diversos dados de input;
- Colheita de dados para análise de resultados. De forma a verificar a qualidade da heurística vs Branch and Bound ($h = 0$) vs Default ($g = 0, h = 0$) (neste caso não é BFS (nem DFS) porque foi utilizado uma Fibonacci heap. A ordem após add não respeita nem append (BFS) nem prepend (DFS)).

2 Especificação

2.1 Descrição

"A alocação dos elementos mais adequados a um projeto é essencial para o seu sucesso. Um projeto é constituído por um conjunto de tarefas a desenvolver por um ou mais elementos. As tarefas podem ter precedências entre si e têm uma duração (pessoa/mês). Cada elemento candidato possui um conjunto de competências, que permitem satisfazer uma ou mais tarefas. É conhecido ainda o desempenho (valor de 1 a 5) de um elemento em cada uma das suas competências. O trabalho consiste em determinar a melhor alocação de elementos às várias tarefas de um projeto de modo a maximizar o desempenho global. Um elemento não pode estar alocado a mais que uma tarefa ao mesmo tempo."

2.2 Análise do Problema

O problema tem como inputs:

- Tarefas. Cada qual com:
  - Precedências;
  - Duração;
  - Competências necessárias.

- Elementos. Cada qual com:
  - Competências. Cada qual com:
    - Pares, Competência-Desempenho, com Desempenho compreendido entre 1 a 5.

Existem algumas restrições inerentes ao problema, nomeadamente:

- Um elemento não pode estar alocado a mais que uma tarefa ao mesmo tempo;
- Um elemento tem de conter todas as competências necessárias para efetuar uma tarefa;
- Uma tarefa só pode ser atribuída após as suas precedências terem sido efetuadas;
- As precedências de tarefas têm de ser acíclicas (têm de ser um DAG).

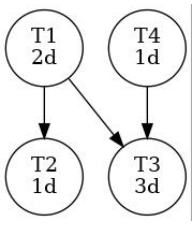

Figura 1: Exemplo de Grafo acíclico de Tasks

O programa analisa estes dados e determina que elementos atribuir a cada tarefa respeitando as restrições do problema. A dificuldade está, portanto, em modelar todas as características do problema: competências, tarefas, precedências, compatibilidade de combinações de elementos, duração, etc de um modo eficiente (em espaço e tempo). A solução final deverá maximizar o desempenho global, que corresponde à alocação ótima dos vários elementos às várias tarefas de forma a minimizar o tempo total do projeto.

2.2.1 Cenários

A alocação de um elemento com muitas competências a uma tarefa simples (com poucas competências) quando esta poderia ter sido atribuída a um elemento mais restrito poderá bloquear a execução em paralelo de tarefas mais complexas. Aumentando assim a duração do projeto.

A escolha de um elemento muito polivalente para uma tarefa simples longa quando esta poderia ter sido atribuída a um elemento com um conjunto de competências mais restrito poderá bloquear a execução de tarefas até que

o elemento polivalente esteja novamente disponível.

Não será possível encontrar uma solução caso haja pelo menos uma tarefa que necessite de competências que nenhum dos elementos possui.

É possível esperar pela disponibilidade de mais elementos para então começar uma tarefa.

2.2.2 Representação do Conhecimento

A informação sobre as tarefas e elementos, e todas as suas características, é capturada de ficheiros JSON com a seguinte estrutura:

```
{
    "skills": ["design", "programming", "statistics"],
    "tasks": [{
            "name": "T1",
            "duration": 2,
            "precedences": [],
            "skills": [0, 1]
        },
        {
            "name": "T2",
            "duration": 1,
            "precedences": [0],
            "skills": [2]
        },
        {
            "name": "T3",
            "duration": 3,
            "precedences": [0, 3],
            "skills": [0]
        },
        {
            "name": "T4",
            "duration": 1,
            "precedences": [],
            "skills": [1]
        }
    ],
    "elements": [{
            "name": "E1",
            "skills": [[0, 5], [1, 5], [2, 4]]
        },
        {
            "name": "E2",
            "skills": [[1, 3], [0, 4]]
        }
    ]
}
```

Listing 1: Ficheiro de input exemplo JSON

2.3 Abordagem

O algoritmo A* resulta da combinação do algoritmo branch and bound (g) e greedy (h), ou seja, soma dos custos acumulados e estimativas do custo até à solução (deverá ser otimista). $f(n) = g(n) + h(n)$. Para garantir que se chega à solução ótima é necessário que a heurística seja consistente, $h(x) <= d(x, y) + h(y)$ (monótona).

2.3.1 Estados

Utilizou-se a seguinte representação para o estado:

```
private PersistentHashSet<Task> remainingTasks;
private PersistentHashSet<Element> elementsNeverAllocated;
private PersistentHashSet<Element> elementsBeenAllocated;
private PersistentHashMap<Task, Integer>
 ↪  taskPrecedencesCounter;
private PersistentHashMap<Integer, TaskResult>
 ↪  taskIdTaskResult;
private PersistentTreeMap<Double,
 ↪  PersistentVector<TaskResult>> startTimeTaskResult;
private double maxEndTime = 0;
```

Foram usadas estruturas de dados persistentes de modo a: evitar desperdícios de memória e facilitar a gestão dos dados na geração de novos estados (evitar partilhas de dados não desejadas, problemas com clones). O estado inicial é:

```
State state = new State();

state.remainingTasks =
 ↪  PersistentHashSet.of(problemData.getTasks());

state.elementsNeverAllocated =
 ↪  PersistentHashSet.of(problemData.getElements());

PersistentHashMap.MutableHashMap<Task, Integer>
 ↪  taskPrecedencesCounterMut =
 ↪  PersistentHashMap.emptyMutable();
for (Task task : problemData.getTasks()) {
        taskPrecedencesCounterMut.assoc(task,
         ↪  task.getPrecedences().size());
}
```

```
state.taskPrecedencesCounter =
↪  taskPrecedencesCounterMut.immutable();

state.elementsBeenAllocated = PersistentHashSet.empty();

state.taskIdTaskResult = PersistentHashMap.empty();

state.startTimeTaskResult = PersistentTreeMap.empty();
```

As variáveis *remainingTasks*, *elementsNeverAllocated* e *taskPrecedencesCounter* foram inicializadas com, respetivamente: todas tarefas do problema, elementos do problema, contabilidade de precedências para cada tarefa. É encontrada uma solução possível quando *state.remainingTasks.size == 0*.

2.3.2 Função de Transição

Cada novo estado gerado é resultado da atribuição de um dado conjunto de elementos a uma dada tarefa que agora já não tem precedências.

```
public ImmutableList<State> generateSuccessors() {

        ImmutableList<Task> tasksCurrentlyWithNoPrecedences =
        ↪  pickTasksCurrentlyWithNoPrecedences();

        ImmutableList.Builder<State> successorsStatesBuilder =
        ↪  new ImmutableList.Builder<>();

        for (Task selectedTask :
        ↪  tasksCurrentlyWithNoPrecedences) {
                ImmutableList<ElementsCombination>
                ↪  elementsCombinations =
                ↪  getCompatibleElementsCombinations(selectedTask);

                generateNewStates(selectedTask,
                ↪  elementsCombinations,
                ↪  successorsStatesBuilder);
        }

        return successorsStatesBuilder.build();
}
```

A função *pickTasksCurrentlyWithNoPrecedences()* retorna as tarefas que neste estado já não têm precedências. Para cada tarefa são encontradas todas as combinações de elementos compatíveis, *getCompatibleElementsCombinations*. Uma combinação, *ElementsCombination* possui a seguinte informação:

```
private static class ElementsCombination {
        private final double startTime;
        private final double duration;
        private final ImmutableSet<Element> elements;
        private final PersistentHashSet<Element>
         ↪  elementsNeverAllocated;
        private final PersistentHashSet<Element>
         ↪  elementsBeenAllocated;
}
```

O tempo de começo (*starTime*) e duração (*duration*) dependem, respetivamente, de:

- disponibilidade de cada elemento (aguardar por todos do conjunto *elements*) e a sua precedência que termina mais tarde
- valores das competências de cada um (só para *task.skills*)

A duração de cada tarefa é dada por:

$$\frac{T}{\sqrt{m_K + ...}}$$

Em que T representa a duração da tarefa a atribuir e m_K a média dos valores das competências do elemento K que são utilizadas nesta tarefa.

getCompatibleElementsCombinations trata de chamar a função *getStartTime* que faz shifts da range *[startTime, startTime+duration]* (*starTime* começa com *endTime* da sua precedência que acaba mais tarde) e intercepta com ranges das tarefas já alocadas e seus elementos de modo a encontrar uma range válida (usa o *TreeMap startTimeTaskResult*).

As variáveis *elementsNeverAllocated* e *elementsBeenAllocated* representam a nova configuração de elementos não alocados e alocados para o novo estado a ser gerado a partir desta combinação.

A lista de combinações, *ImmutableList<ElementsCombination>*, é então passada para *generateNewStates* onde é criado um estado para cada ElementsCombination com *remainingTasks*, *elementsNeverAllocated*, *elementsBeenAllocated*, *taskPrecedencesCounter*, *taskIdTaskResult*, *startTimeTaskResult*, *maxEndTime* atualizados.

2.3.3 Função de custo

O custo acumulado representa o impacto de não se conseguir fazer uma tarefa em paralelo. É dado por:

```
double getTotalTimeDelta(State destinationState) {
        return destinationState.maxEndTime - maxEndTime;
}
```

O custo posterior foi calculado pegando na tarefa atualmente sem precedências com o maior endTime assumindo que todos os elementos estão disponíveis.

```
double heuristic() {
        double taskMaxEndTime = maxEndTime;
        for (Task task :
         ↪  pickTasksCurrentlyWithNoPrecedences()) {
                double startTime =
                 ↪  getTaskStartTimeFromPrecedences(task);
                double elementsCompetenceSum =
                 ↪  getCompatibleElementsSum(task);
                double duration =
                 ↪  getTaskReducedDuration(task.getDuration(),
                 ↪  elementsCompetenceSum);
                taskMaxEndTime = Math.max(taskMaxEndTime,
                 ↪  startTime + duration);
        }
        return taskMaxEndTime - maxEndTime;
}
```

A heurística presente no relatório intercalar: resolução do problema relaxado por remoção da restrição de um elemento não poder estar alocado a mais que uma tarefa ao mesmo tempo, respeitando as precedências; revelou ser demasiado complexa piorando o tempo de execução. Portanto, foi desprezada.

3 Desenvolvimento

Utilizou-se a linguagem de programação Java para a codificação do algoritmo; e Javascript para criar a interface de visualização. Foi usado o sistema de compilação Gradle. As bibliotecas Java usadas foram:

- com.google.code.gson:gson — deserializar e serializar JSON
- org.graphstream:gs-algo — fibonacci heap
- com.google.guava:guava — ranges e coleções imutáveis
- org.organicdesign:Paguro — coleções persistentes

Para executar o programa basta ter o gradle instalado e correr `gradle run -Pargs="../data/1a.json"` na pasta *project_management*. Os ficheiros resultado são colocados na mesma pasta que o input `"../data/1a_{{algo}}.json"`.

Para visualizar a solução é necessário abrir o ficheiro *index.html*, pasta *web_visualizer*, e importar, por exemplo, 1a.json e 1a_astar.json.

3.1 Arquitetura da aplicação

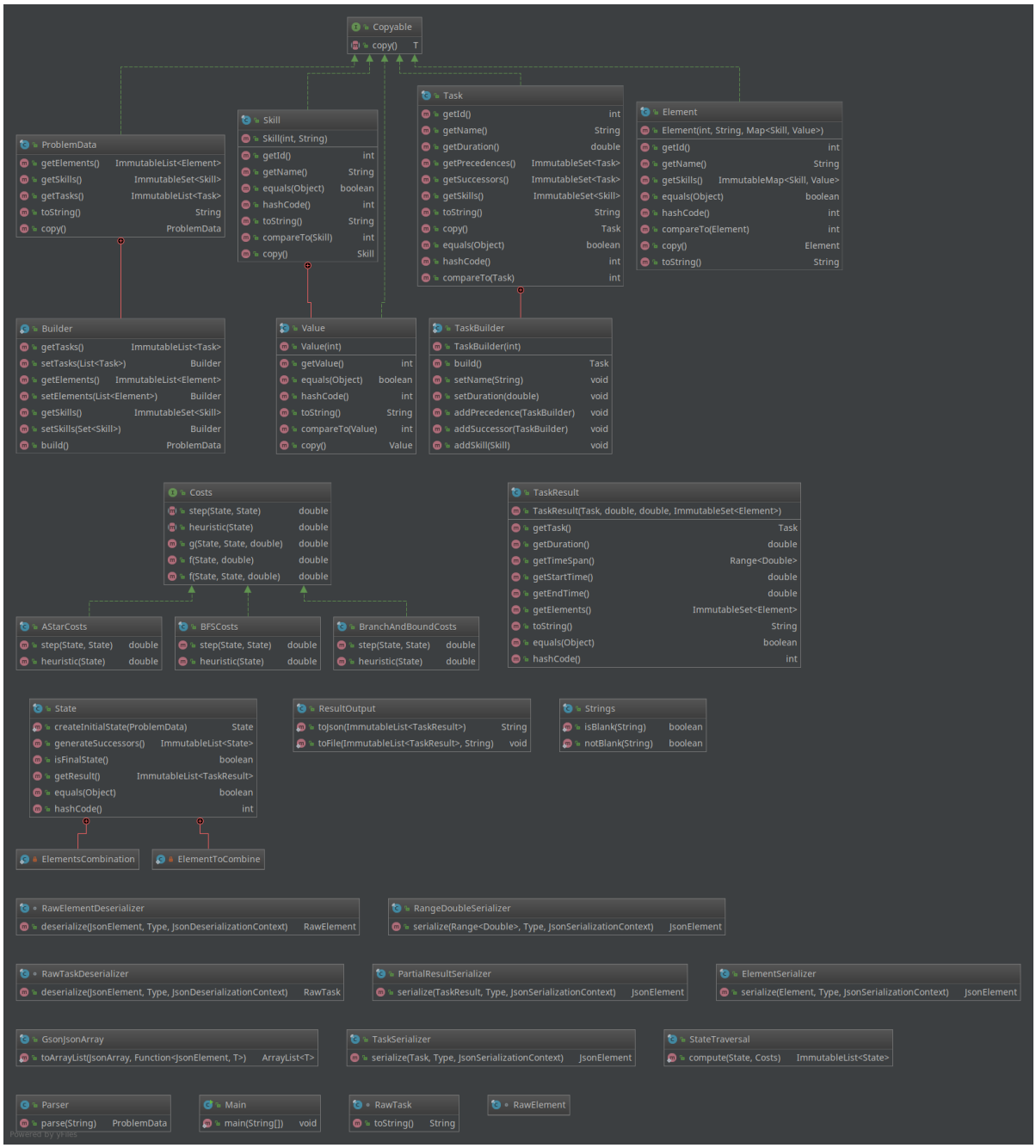

Figura 2: Project Classes (only public methods)

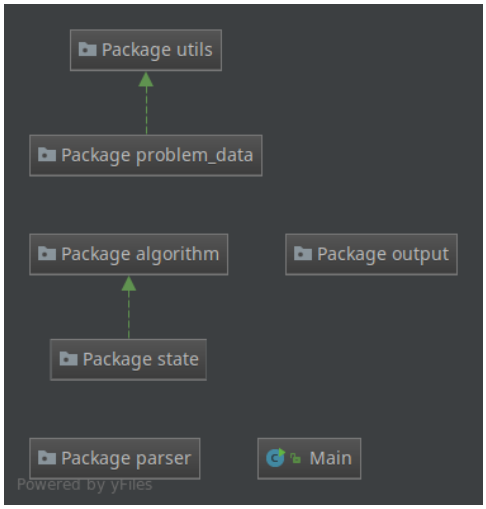

Figura 3: Project Packages

3.2 Detalhes Relevantes da implementação

Foi tido em atenção as complexidades espaciais e temporais da implementação, respetivamente, estruturas de dados persistentes e Fibonacci Heap (decreaseKey, O(1) amortized running time, atualização gScores da openList).

4 Experiências

A validação dos resultados é feita através da visualização gráfica de uma timeline presente numa página web. São usados vários datasets com diferentes dimensões e topologias de forma a melhor compreender o comportamento do algoritmo A* vs Branch and Bound vs Default. No início do projeto foram usados ficheiros de input relativamente pequenos de modo a confirmar a boa operação do programa. Numa fase posterior foram realizados testes de dimensão média. Os tempos foram calculados a partir da média de amostras de dimensão 5 (outliers desprezados).

4.1 Resultados das Experiências

4.1.1 Experiência 1a.json

Tarefas com precedências, elementos com conjunto de competências diferentes.

Time (microseconds)		
Default	B&B	A^*
29416	49164	47223

4.1.2 Experiência 1b.json

Tarefas sem precedências, elementos com conjunto de competências diferentes.

Time (microseconds)		
Default	B&B	A^*
34413	72616	74578

4.1.3 Experiência 2a.json

Tarefas com precedências, elementos com conjunto de competências igual.

Time (microseconds)		
Default	B&B	A^*
30111	57619	52724

4.1.4 Experiência 2b.json

Tarefas sem precedências, elementos com conjunto de competências igual.

Time (microseconds)		
Default	B&B	A^*
43403	883716	902495

4.1.5 Experiência 3a.json

Tarefas com precedências, elementos com conjunto de competências igual e diferente.

Time (microseconds)		
Default	B&B	A^*
39130	448284	312086

4.1.6 Experiência 3b.json

Tarefas sem precedências, elementos com conjunto de competências igual e diferente.

Time (microseconds)		
Default	B&B	A^*
56205	5542214	5254838

4.2 Análise heurística

O algoritmo com Default ($g = 0, h = 0$) é o mais rápido visto não encontrar a solução ótima (neste caso não é BFS (nem DFS) porque foi utilizado uma Fibonacci heap. A ordem após add não respeita nem append (BFS) nem prepend (DFS)). Neste problema, o BFS também não encontraria a solução ótimo visto todas as soluções estarem no mesmo nível.

O Branch and Bound e A^* (heurística consistente) encontram sempre a melhor solução. Para tarefas com precedências a heurística é tanto melhor quanto maior for a dimensão do problema.

4.3 Visualização exemplo, experiência 1a.json

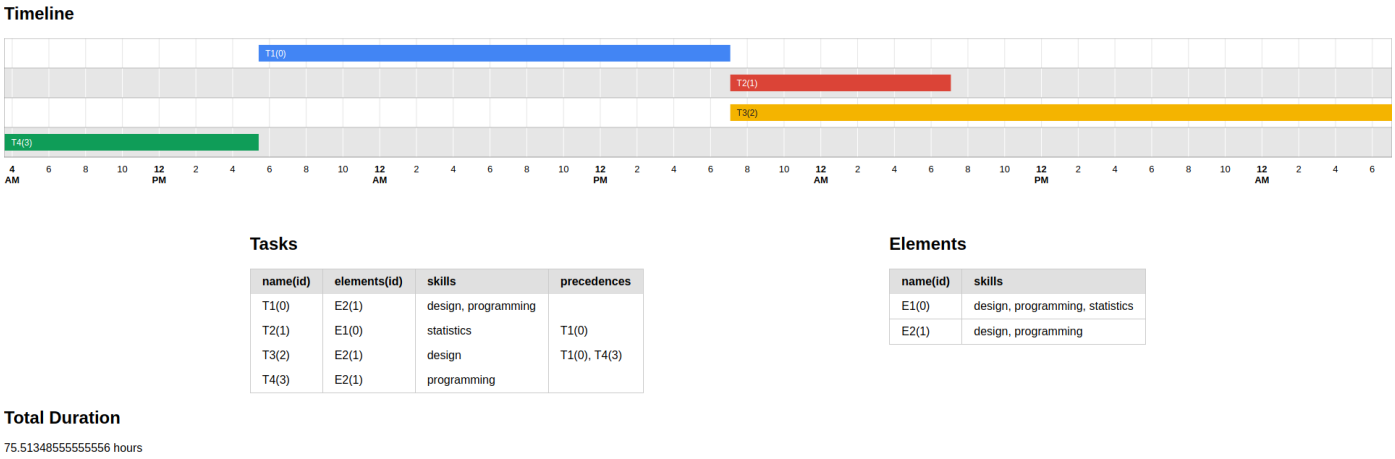

Figura 4: Visualização 1a.json, 1a_default.json

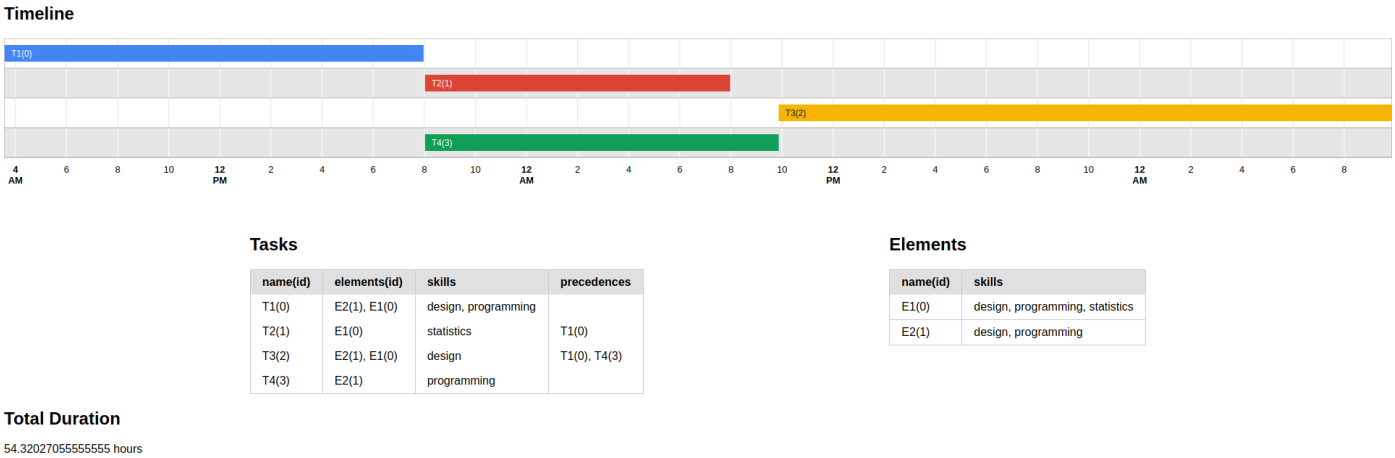

Figura 5: Visualização 1a.json, 1a_astar.json

5 Conclusões

Chegou-se à conclusão que a dificuldade nestes algoritmos centra-se na modelação do problema (estados, função de transição, ações, custos) e na descoberta de uma heurística não muito complexa que garanta o resultado ótimo (consistente) em menor tempo. Como era de prever o Branch and Bound e o A^* (visto a nossa heurística ser consistente) encontram a solução ótima. A heurística discriminada no relatório intercalar (sem dados de experiências presentes) era pior que o Branch and Bound em todos os casos testados. Ou seja, a heurística era demasiado complexa. A nova, descrita com pormenor no relatório, apresenta dados temporais semelhantes ao Branch and Bound quando as tarefas não têm precedências (1b, 2b, 3b). No entanto, para tarefas com precedências, a heurística é tanto melhor quanto maior for a dimensão do problema (1a, 2a, 3a).

6 Melhoramentos

Um possível melhoramento seria construir um gerador de estados mais inteligente que eliminasse combinações de elementos que se sabe à partida não poderem levar à solução ótima. Removendo assim o número excessivo de estados. Uma tarefa com N elementos compatíveis tem $\sum \binom{N}{k} - 1, k \in [1, N] = 2^N - 1$ combinações destes (power set - conjunto vazio). Seria também útil fazer um estudo comparativo entre várias heurísticas, visto umas serem melhores para projetos de planeamento com diferentes topologias. Poderiam também ser introduzidas outras restrições como tempo máximo de trabalho por semana. Outro cenário que não foi contabilizado é a atribuição de um elemento a uma tarefa quando esta já está em execução.

7 Recursos

7.1 Websites

- Slides das teóricas
- Artificial Intelligence - Berkeley (Spring 2018)
- Amit's A* Pages From Red Blob Games

7.2 Software

- Intellij IDEA
- GraphStream
- Gradle

- Paguro
- Gson
- Google Charts
- pure-css
- lodash
- handlebars