UNIVERSIDADE FEDERAL DO PARANÁ

FERNANDA CASSEMIRO PEREIRA

GERHARD BESSER NETO

PROPOSTA DE FERRAMENTA PARA FAZER DE FORMA AUTOMATIZADA A SELEÇÃO DE REQUISITOS DE *SOFTWARE* EM VERSÕES UTILIZANDO PLANEJAMENTO EM INTELIGÊNCIA ARTIFICIAL PARA PROJETOS COM UMA OU MÚLTIPLAS EQUIPES

CURITIBA

2021

FERNANDA CASSEMIRO PEREIRA

GERHARD BESSER NETO

PROPOSTA DE FERRAMENTA PARA FAZER DE FORMA AUTOMATIZADA A SELEÇÃO DE REQUISITOS DE *SOFTWARE* EM VERSÕES UTILIZANDO PLANEJAMENTO EM INTELIGÊNCIA ARTIFICIAL PARA PROJETOS COM UMA OU MÚLTIPLAS EQUIPES

Trabalho apresentado como requisito parcial para a conclusão do Curso de Bacharelado de Ciência da Computação, setor de Ciências Exatas,da Universidade Federal do Paraná.

Orientadora: Professora Dra. Letícia Mara Peres.

CURITIBA

2021

AGRADECIMENTOS

Gostaríamos de agradecer à nossa orientadora, Professora Dra. Letícia Mara Peres, pela orientação, paciência, e resiliência para nos conduzir através desta pesquisa em tempos tão complicados.

Aos professores do Departamento de Informática (DINF/UFPR) que compartilharam experiências e conhecimentos conosco durante todo o curso.

Às nossas famílias que nos incentivaram e não nos deixaram desistir.

Aos nossos companheiros, amigos e colegas de turma por todo apoio e ajuda nos momentos difíceis e pela amizade incondicional durante todos esses anos.

Aos funcionários da secretaria do DINF que cansaram de realizar ajustes nas nossas matrículas.

À todos que participaram, direta ou indiretamente, do desenvolvimento deste trabalho de pesquisa, enriquecendo nosso processo de aprendizado.

RESUMO

Este trabalho apresenta o processo de ideação e implementação de uma ferramenta que utiliza conceitos e artefatos de planejamento, um ramo da Inteligência Artificial, para organizar requisitos de *software* de forma automatizada, levando em consideração o tamanho das próximas versões onde eles serão desenvolvidos, seus níveis de prioridade e as relações de dependência que existem entre eles. A ferramenta desenvolvida recebe como entrada um arquivo em extensão xml com a lista dos requisitos a serem priorizados, a informação de quantas equipes de desenvolvimento estão disponíveis e qual o tamanho das próximas versões onde os requisitos serão desenvolvidos. Os dados recebidos são convertidos em um par de arquivos em linguagem PDDL (Linguagem de Definição de Domínio de Planejamento, do inglês *Planning Domain Description Language*) que são passados como entrada para o planejador Metric-FF que realiza a priorização dos requisitos e gera um plano. O resultado é devolvido para o usuário na forma de um arquivo em extensão PDF contendo um ou mais grafos que representam os requisitos já priorizados e separados por times. A ferramenta foi testada em um projeto real de desenvolvimento de um *software* e os resultados dos experimentos mostram que, apesar dela depender de fatores externos como o projeto em que está sendo aplicada e o nível de experiência profissional de quem está definindo seus parâmetros de entrada, ela pode contribuir para o planejamento das atividades e do cronograma do projeto.

Palavras-chave: priorização de requisitos de *software*, planejamento clássico, planejamento automatizado, IA, planejadores, PDDL, desenvolvimento ágil.

ABSTRACT

This work presents the ideation and implementation process of a tool that uses planning concepts and artifacts, a field of Artificial Intelligence, to organize software requirements in a automated way, taking into account the size of the next versions where they are going to be developed, their priority levels and the dependency relationships that exist between them. The developed tool receives as input an xml file with the list of requirements to be prioritized, the information of how many development teams are available and the size of the next versions where the requirements will be developed. The received data are converted into a pair of files in the language PDDL (Planning Domain Definition Language), which are passed as input to the Metric-FF planner which prioritizes requirements and generates a plan. The outcome is returned to the user as a PDF extension file containing one or more graphs that represent the requirements already prioritized and separated by teams. The tool was tested on a real software development project and the results of the experiments show that, although it depends on external factors such as the project in which it is being applied and the level of professional experience of those who are defining its input parameters, it can contribute to the planning of activities and the project schedule.

Key-words: software requirements prioritization, classical planning, automated planning, AI, planners, PDDL, agile planning.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Cada vez mais a sociedade moderna tem feito uso das facilidades decorrentes dos avanços da Tecnologia da Informação. De acordo com Mujumdar (2012), o computador tornou-se uma parte crucial em diversos campos da vida das pessoas como, por exemplo, nas indústrias, na medicina, na educação, no comércio e até mesmo na agricultura. Segundo o autor, os computadores ajudam a executar processos que são complexos, demorados e repetitivos de maneira mais eficiente e em curtos períodos de tempo. Nas últimas quatro décadas o *software* evoluiu de uma ferramenta de análise de informações ou de resolução de problemas atrelada a uma máquina e passou a ser vendido por si só.

Além disso, originalmente as equipes de produção de *softwares* eram compostas por times pequenos, com uma única equipe de desenvolvimento que trabalhava alocada em um mesmo espaço físico. Atualmente, como exemplificado em Paasivaara(2011) e em Almutairi(2015), muitas companhias desenvolvem grandes sistemas de *software* com múltiplas equipes localizadas ao redor do mundo.

Segundo Sommerville(1995), a produção de um *software* é feita através de um processo de *software* que pode ser dividido em duas categorias: dirigidos a planos e ágeis. Atualmente, o processo ágil se tornou muito popular, sendo um de seus principais princípios o desenvolvimento incremental, que consiste em desenvolver e entregar o sistema de forma iterativa, implementando e validando novos requisitos a cada versão (Sommerville, 2011). Atualmente, já é possível mesclar o desenvolvimento incremental ao uso de múltiplas equipes permitindo que atividades não diretamente correlacionadas sejam desenvolvidas em paralelo de modo que mais de um conjunto de requisitos seja entregue a cada versão.

Entretanto, no desenvolvimento incremental, uma das etapas é determinar quais requisitos devem ser implementados para a próxima versão. Segundo Bagnall(2001), essa escolha deve levar em conta alguns aspectos, como prioridade dos clientes, dependências e custos dos requisitos. Além disso, essa opção torna-se especialmente complexa quando as partes interessadas possuem interesses

conflitantes. Bagnall definiu o problema de escolher um conjunto de requisitos para ser entregue na próxima versão do produto, satisfazendo os interesses citados acima, como problema da próxima versão (em inglês, *Next Release Problem).*

De acordo com Del Sagrado*(2011),* na maioria dos casos nem todas as necessidades das partes interessadas podem ser atendidas dentro de prazos e restrições dos recursos existentes e, portanto, devem ser limitadas de alguma forma. O autor propõe que essa limitação seja realizada por meio da priorização dos requisitos e da seleção do melhor subconjunto que pode ser escolhido de acordo com os recursos disponíveis.

Os benefícios da priorização de requisitos são vários. Conforme Olaronke *et al.* (*2018),* priorizar os requisitos reduz o custo e o tempo de desenvolvimento de *software* em 40%, ajuda a identificar os requisitos mais valiosos de um conjunto volumoso e ambíguo, além de reduzir falhas e resolver desacordos ou disparidades entre as partes interessadas. Portanto, a priorização de requisitos melhora a qualidade de uma versão e pode ser considerada uma das etapas mais cruciais do processo de desenvolvimento de *software*.

Entretanto, priorizar os requisitos é uma missão complicada: o número de requisitos não precisa ser muito grande para tornar o problema da próxima versão muito complexo. Inclusive, ele pode ser visto como um problema de otimização relacionado ao problema da mochila (*Knapsack problem*) (KARP, *1972*), que pertence a classe de complexidade NP-Completo. Desse modo, conforme o número de requisitos aumenta, buscar a solução ótima torna-se impossível, fazendo necessário o uso de métodos heurísticos e metaheurísticos (AMARAL, 2017).

Simultaneamente, há um interesse crescente na aplicação de Técnicas de Inteligência Artificial (IA) em sistemas tradicionais para resolver problemas comuns. As abordagens de planejamento em IA já são amplamente utilizadas, por exemplo, em áreas como missões espaciais e robótica, como visto em Machado *et al. (*2016). A partir disso, acredita-se que outras áreas também podem se beneficiar da aplicação das técnicas de planejamento em IA.

Para reduzir os desafios relacionados ao problema da próxima versão e aumentar a eficiência e a qualidade da solução fornecida para a priorização de

requisitos, propõe-se o uso da abordagem de planejamento em Inteligência Artificial para auxiliar na alocação dos requisitos a serem desenvolvidos em cada versão.

1.1 Objetivos

Este trabalho tem como objetivo geral apresentar, através de uma abordagem relacionada ao planejamento em Inteligência Artificial, uma ferramenta para apoiar a seleção automatizada de requisitos de *software* a serem entregues nas próximas versões em projetos que contam com uma ou múltiplas equipes de desenvolvimento.

Como objetivos específicos tem-se:

a) Criar uma ferramenta de planejamento de requisitos que receba como entrada uma lista de todos os requisitos de um *software* e as relações existentes entre eles.

b) Fazer de forma automatizada a separação dos requisitos de *software* passados como entrada em subconjuntos, considerando a prioridade, o tamanho, as relações de dependências dos requisitos, o número de equipes de desenvolvimento disponíveis e o balanceamento da quantidade de requisitos atribuídos a cada equipe.

c) Gerar como saída para o usuário um plano com a ordem dos requisitos a serem desenvolvidos nas próximas versões do *software*.

1.2 Estrutura do texto

Este documento contém as informações de desenvolvimento da ferramenta criada e sua estrutura está detalhada a seguir.

O Capítulo 1 compreende a introdução do trabalho, o contexto da pesquisa, seus objetivos e as motivações para o desenvolvimento da ferramenta.

O Capítulo 2 tem como objetivo trazer a fundamentação teórica e definições técnicas usadas como base para compreender o contexto onde a ferramenta desenvolvida será aplicada. Entre os assuntos tratados estão temas como o

desenvolvimento incremental de *softwares* e o problema da próxima versão. Também é feita uma introdução ao conceito de planejamento em IA, à linguagem PDDL e ao uso de planejadores.

O Capítulo 3 apresenta os métodos e técnicas utilizados para consolidar a ferramenta de modo que o planejamento em IA fosse aplicado de forma que a arquitetura final gerasse resultados satisfatórios.

O Capítulo 4 retrata como foi feita a construção do modelo, da ferramenta, como ficou sua arquitetura e traz informações sobre seu funcionamento. Também são apresentados os resultados obtidos através da aplicação da ferramenta em alguns estudos de caso.

O Capítulo 5 apresenta as considerações finais a respeito deste projeto. São analisados se os objetivos propostos foram cumpridos de forma satisfatória, se houve tópicos que não tiveram sucesso e quais as possibilidades de trabalhos a serem realizados ou melhorados em oportunidades futuras.

Ao final deste documento são encontrados alguns códigos implementados na linguagem PDDL e o estudo teórico prático realizado para a concepção da ferramenta desenvolvida.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

Este capítulo aborda alguns tópicos relacionados à revisão bibliográfica. Inicialmente, a Seção 2.1 apresenta o processo de desenvolvimento de um *software* e os conceitos da abordagem incremental. A Seção 2.2 apresenta o problema da próxima versão. Por fim, a Seção 2.3 apresenta os conceitos de planejamento em Inteligência Artificial e introduz a Linguagem de Definição de Domínio de Planejamento e a Seção 2.4 apresenta os *softwares* e bibliotecas utilizadas como apoio para a ferramenta.

2.1 Processos de desenvolvimento de *software*

Segundo Pressman(2015), quando se elabora um produto ou sistema é importante seguir uma sequência de passos previsíveis, um roteiro, que ajude a criar um resultado de alta qualidade dentro do prazo estabelecido e que propicie controle, estabilidade e organização para tarefas que podem se tornar bastante caóticas quando realizadas sem controle. Na engenharia de *software*, esse roteiro é denominado “processo de *software*”.

Um processo neste contexto, de acordo com Pressman(2015), é definido como um conjunto de atividades de trabalho, ações e tarefas realizadas quando algum artefato de *software* deve ser criado. Estas atividades são, segundo Mujumdar(2012):

1) Comunicação: estabelecer as expectativas das partes interessadas;
2) Planejamento: desenvolver um plano bem definido de execução do projeto;
3) Modelagem: desenvolver um modelo do projeto antes de desenvolver o projeto real;
4) Construção: construir o projeto real seguindo o plano de execução definido na etapa de planejamento e teste;
5) Entrega: a entrega do produto final ao cliente e sua manutenção.

Em um mundo ideal, essas atividades poderiam ser sempre executadas sequencialmente seguindo um fluxo de processo linear (Figura 1). Entretanto, projetos reais raramente se adaptam ao modelo linear uma vez que logo no início é difícil estabelecer explicitamente todos os requisitos, recursos e tempo necessários. Portanto as atividades podem ser organizadas de diferentes maneiras em relação à sequência e ao tempo disponível para cada projeto além de fazer uso de um ou mais fluxos de processo apresentados nas Figuras 1, 2, 3 e 4.

Figura 1: Fluxo de processo linear: as cinco atividades são executadas em sequência.

Fonte: Os autores com base em Pressman, 2015.

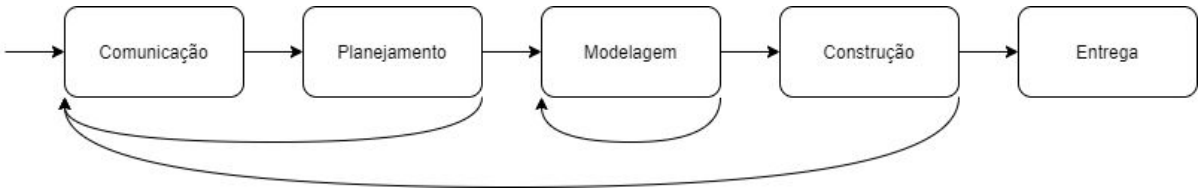

Figura 2: Fluxo de processo interativo: executa-se uma ou mais das atividades antes de prosseguir para a seguinte.

Fonte: Os autores com base em Pressman, 2015.

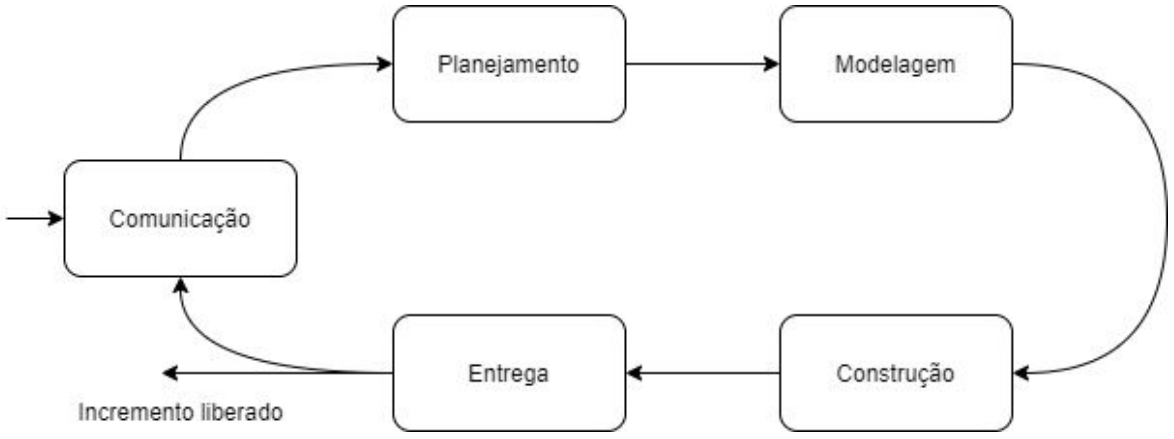

Figura 3: Fluxo de processo evolucionário: executam-se as atividades de forma “circular” e cada volta completa conduz à uma versão mais completa do *software*.

Fonte: Os autores com base em Pressman, 2015.

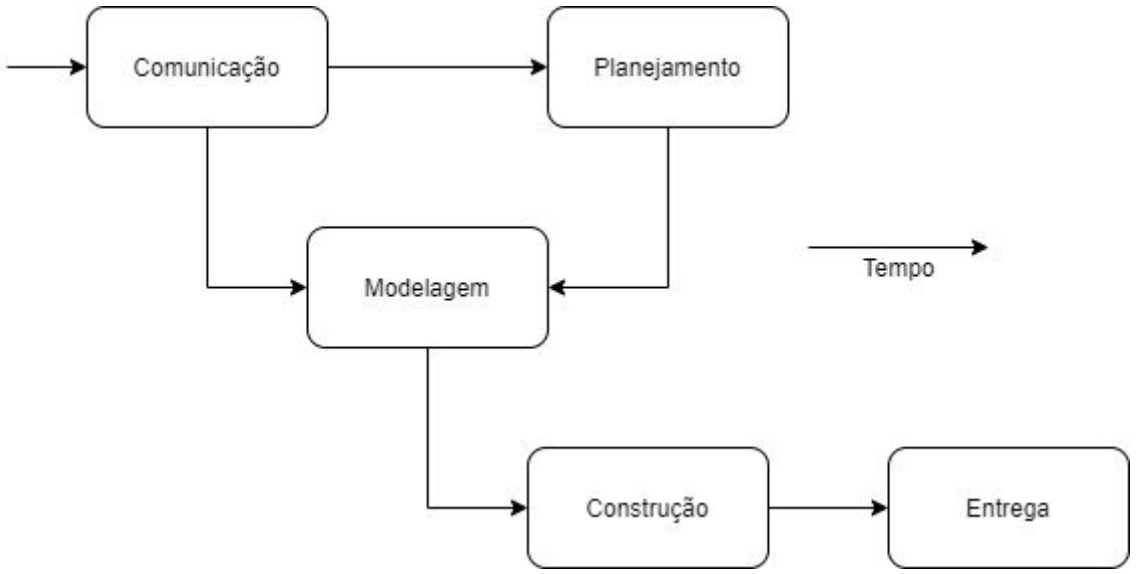

Figura 4: Fluxo de processo paralelo: executa-se uma ou mais atividades em paralelo com outras.
Fonte: Os autores com base em Pressman, 2015.

Mujumdar(2012) classifica algumas dessas diferentes maneiras de organizar um projeto em relação ao tempo e a sequência de atividades e refere-se a elas como diferentes abordagens. Neste documento focaremos em uma delas: a abordagem incremental.

2.1.1 Abordagem incremental

A abordagem incremental enfatiza, segundo Mujumdar (2012), o desenvolvimento em fases, oferecendo uma série de mini-projetos relacionados (referidos como incrementos ou versões) gerados a partir de uma especificação de requisitos pré-definida. O primeiro incremento costuma atuar como o produto principal, fornecendo as funcionalidades necessárias para atender aos requisitos básicos e cada mini-projeto adiciona funcionalidades adicionais produzindo uma versão operacional do sistema que será entregue para uso ou avaliação detalhada do cliente. A filosofia de liberação escalonada permite aprendizado e *feedback* que podem modificar alguns dos requisitos do cliente nas versões subsequentes.

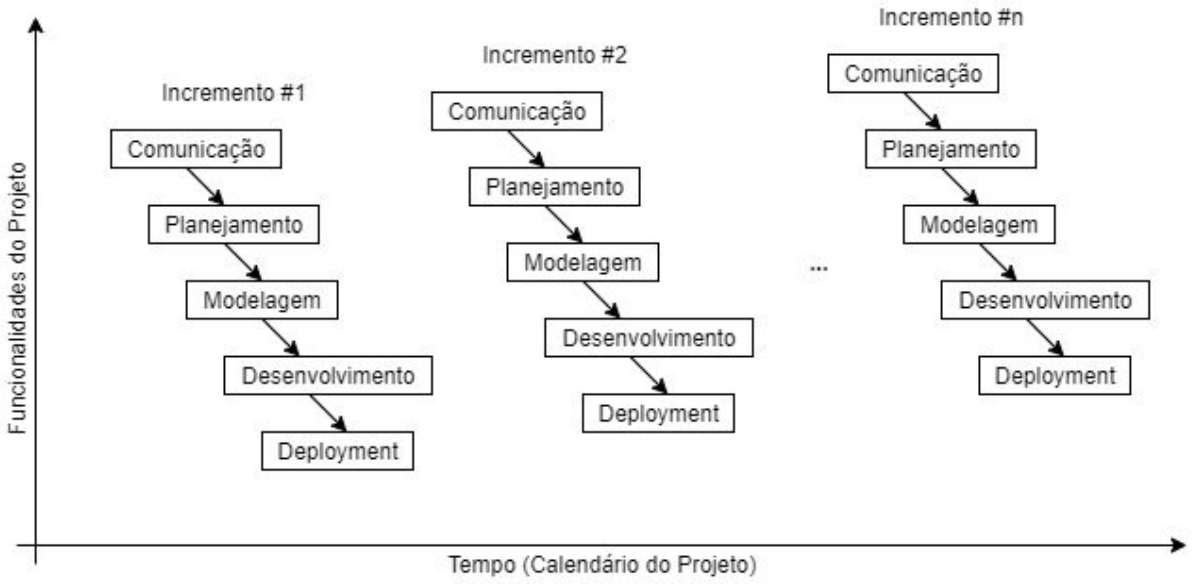

Figura 5: Desenvolvimento incremental.

Fonte: Os autores com base em Mujumdar, 2012.

De acordo com Mujumdar, abordagens incrementais são particularmente úteis quando a equipe necessária para desenvolver o projeto não está inteiramente disponível e quando há dificuldades para especificar completamente o produto desejado ou formular completamente o conjunto de requisitos. Existem muitas situações em que os requisitos iniciais de *software* são razoavelmente bem definidos, mas o escopo geral do esforço de desenvolvimento impede um processo puramente linear. Assim, o desenvolvimento incremental combina elementos do processo de fluxo linear(Figura 1) com elementos do processo interativo (Figura 2).

Ainda de acordo com Mujumdar, algumas vantagens do desenvolvimento incremental são:

- O projeto é dividido em partes menores;
- Um protótipo operacional é gerado mais cedo;
- O *feedback* de uma fase fornece informações de design para a próxima fase;
- Muito útil quando parte da equipe de desenvolvimento não está disponível;
- Permite que requisitos referentes aos projetos dos clientes mais importantes sejam priorizados.

E algumas desvantagens são:

- A comunidade de usuários precisa estar ativamente envolvida no projeto. Isso exige tempo da equipe e aumenta o atraso do projeto;
- As habilidades de comunicação e coordenação são essenciais;
- Solicitações informais de melhoria para cada fase podem levar a confusão.

2.1.2 Uso de múltiplas equipes no desenvolvimento incremental

Segundo Paasivaara(2011), nos dias de hoje muitas companhias têm desenvolvido grandes sistemas de *software* utilizando múltiplas equipes de desenvolvimento alocadas em diferentes localizações geográficas. Dividir um projeto entre várias equipes apresenta alguns desafios como, por exemplo, coordenação entre os times, falta de arquitetura adequada, falta de um levantamento de requisitos adequado, além de todos os desafios pré-existentes em projetos distribuídos.

É possível contornar alguns desses desafios fazendo uso da abordagem incremental. De acordo com Mujumdar(2012), essa abordagem é sustentada pelo pressuposto de que é possível isolar subconjuntos significativos que podem ser desenvolvidos, testados e implementados independentemente, e dessa forma, é permitido trabalhar em diferentes partes e fases se sobrepõem ao longo do uso de vários mini-ciclos em paralelo.

2.2 Problema da próxima versão

Esta seção foi baseada no trabalho de Bagnall *et al*. (2001).

Um problema enfrentado por qualquer entidade envolvida no desenvolvimento e manutenção de grandes e complexos sistemas de *softwares* é o de determinar o que deve estar na próxima versão. Em geral, as empresas se deparam com:

- ampla gama de melhorias por cliente;
- clientes que possuem mais valor para a empresa e, portanto, os seus projetos terão um peso maior sobre os demais;
- situações em que algumas melhorias exigirão (um ou mais) aprimoramentos como pré-requisito;
- requisitos que exigirão quantidades muito diferentes de tempo e esforço.

Como descrito por Bagnall, o problema da próxima versão consiste em selecionar um subconjunto de requisitos a ser desenvolvido e entregue no próximo incremento do produto considerando características específicas do projeto. Conforme Olaronke *et al.* (*2018),* priorizar os requisitos reduz o custo e o tempo de desenvolvimento de *software* em 40%, ajuda a identificar os requisitos mais valiosos de um conjunto volumoso e ambíguo, além de reduzir falhas e resolver desacordos ou disparidades entre as partes interessadas. Portanto, a priorização de requisitos melhora a qualidade de uma versão e pode ser considerada uma das etapas mais cruciais do processo de desenvolvimento de *software*.

Entretanto, o problema de escolher os requisitos para a próxima versão do *software* pode ser comparado ao problema da mochila(*Knapsack problem*) exposto por Richard Karp em 1972 cuja complexidade de decisão é conhecida por ser da classe dos problemas NP-completos. Bagnall define (Bagnall, 2001) o seguinte teorema:

Teorema: O problema da próxima versão é NP-difícil mesmo quando o problema é básico e os requisitos do cliente são independentes entre si.

Traduzido de Bagnall (2001), “o significado prático desse resultado é que (exceto nos casos raros onde P = NP) o problema da próxima versão não pode ser resolvido por um algoritmo de tempo polinomial. Apesar disso, é possível encontrar a solução ideal usando algoritmos combinatórios para casos pequenos, mas, como o número de requisitos tende a aumentar, o problema torna-se intratável”.

Seja R o conjunto de todos os requisitos a serem desenvolvidos. Associado a cada $r \in R$, há uma prioridade $p_r \in Z^+$, um tamanho $t_r \in Z^+$ e um conjunto de dependências $D_r \subset R$, referente aos requisitos que devem ser desenvolvidos antes ou juntos à r.

Considere $w \in \{0,1\}$, sendo que o valor 1 (um) representa um requisito que foi selecionado para ser desenvolvido para a próxima versão, e o valor 0 (zero) um requisito que não foi selecionado. Logo, o problema é selecionar um subconjunto de requisitos $P \subseteq R$ a ser desenvolvido para a próxima versão do produto, buscando maximizar a fórmula 1, e respeitando as restrições descritas pelas fórmulas 2 e 3.

(1) $\sum_{r \in R} w_r p_r$

(2) $w_r = 1 \rightarrow w_d = 1, \ \forall r \in R \ e \ \forall d \in D_r$

(3) $\sum_{r \in R} w_r t_r < T$, para algum T ∈ Z^+, referente ao tamanho máximo da próxima versão.

Como exemplo para a formulação descrita acima temos o seguinte problema: considere o conjunto de requisitos R = {A, B, C, D} com tamanhos 1, 4, 2, 7 e prioridades 1, 2, 3, 1, respectivamente. Os requisitos B e C dependem do requisito A (Figura 6) e o tamanho máximo da próxima versão limitado superiormente em 7. A partir do exposto, o objetivo é escolher um subconjunto de R com tamanho máximo 7, respeitando as restrições de dependência e buscando maximizar a prioridade.

Sendo assim, os possíveis subconjuntos formados para serem desenvolvidos são: S_1={ A, B, C } e S_2={D}. Esses, por sua vez, têm prioridade 6 e 1, respectivamente. Portanto, o subconjunto que melhor soluciona o problema é o S_1.

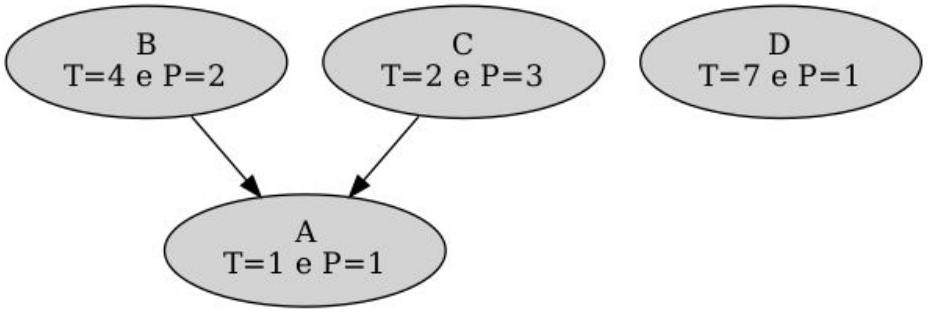

Figura 6: Representação do problema da próxima versão. Os vértices representam os requisitos e uma aresta e(r, r') representa dependência de r' por r.

2.3 Planejamento em inteligência artificial

O planejamento é o ramo da IA que diz respeito à síntese automatizada de comportamentos autônomos, ou seja, planos, (na forma de estratégias ou sequências de ações) para classes específicas de modelos matemáticos representados de forma compacta. Nos últimos anos, a comunidade de planejamento automatizado desenvolveu uma infinidade de sistemas de planejamento (também conhecidos como planejadores) que incorporam heurísticas independentes de domínio muito eficazes (ou seja, escalonam até grandes problemas), que foram empregadas para resolver coleções de problemas desafiadores em vários domínios da Ciência da Computação (Marrella, 2017).

Existem várias formas de modelos de planejamento na literatura da IA, segundo Marrella, que resultam da aplicação de algumas dimensões ortogonais: a incerteza no estado inicial (total ou parcialmente conhecido) e na dinâmica das ações (determinísticas ou não), o tipo de *feedback* (completo, parcial ou inexistente) e se a incerteza é representada por conjuntos de estados ou distribuições de probabilidade.

Como mencionado por Marrella, a forma mais simples de planejamento em que as ações são determinísticas, o estado inicial é conhecido e os planos são sequências de ações calculadas com antecedência é denominada Planejamento Clássico. Para o planejamento clássico o problema geral de apresentar um plano é NP-difícil, uma vez que o número de estados do problema é exponencial no número de variáveis do problema.

O modelo de estados básico de planejamento contém (Geffner e Bonet, 2013):

- Um espaço de estados discreto e finito S.
- Um estado inicial conhecido $s_0 \in S$.
- Um conjunto de estados não-vazio $S_G \subseteq S$ de estados objetivos.
- Ações $A(s) \in A$ de ações aplicáveis no estado $s \in S$.
- Uma função de transição determinística f(a,s) que retorna o estado resultante ao aplicar a ação $a \in A$ no estado $s \in S$.
- Uma função de custo de ação C(a,s).

A partir da definição de um domínio para esse modelo, um plano pode ser gerado. Um plano consiste em uma sequência de ações $P \in A$ que , ao ser aplicada no estado inicial s_0, leva a um estado objetivo $s_f \in S_G$. Para avaliar a qualidade de um plano, os planejadores podem maximizar ou minimizar uma métrica, como a soma dos custos das ações do plano, a quantidade de ações realizadas ou uma função composta por diferentes métricas.

Apesar de sua complexidade, o campo do planejamento clássico experimentou grandes avanços (em termos de escalabilidade) nos últimos vinte anos, levando a uma variedade de planejadores concretos capazes de calcular planos com milhares de ações para problemas mundiais contendo centenas de proposições. Tais avanços foram possíveis porque os planejadores clássicos de ponta empregam funções heurísticas que são derivadas automaticamente pelo problema específico e permitem direcionar inteligentemente a busca em direção à meta.

Um método comum de modelar os problemas para que sejam entendidos pelos planejadores é usando PDDL (Linguagem de Definição de Domínio de Planejamento, do inglês *Planning Domain Definition Language*).

2.3.1 Linguagem de Definição de Domínio e Planejamento (PDDL)

PDDL é uma linguagem criada em 1998 que buscou unificar diversos métodos usados em planejamento, como STRIPS e ADL, e seus modelos consistem em dois arquivos: um arquivo de domínio (*domain.pddl*) e um arquivo com problema (*problem.pddl*).

O arquivo de domínio contém o conhecimento a respeito do problema a ser resolvido. Esse conhecimento deve ser traduzido em tipos de objetos, predicados, funções e ações. Já no arquivo de problema, é descrita uma instância de um problema a ser resolvido. A instância é composta pelos estados inicial e final e por uma métrica (opcional). Sendo assim, um mesmo arquivo de domínio pode ser utilizado por diferentes instâncias (arquivos de problema) e essa é a principal razão da separação de um problema de planejamento em dois arquivos (CANTONI *et al.,*

2010). Ambos os arquivos são passados como parâmetros para um planejador que gera, se possível, um plano (Figura 7).

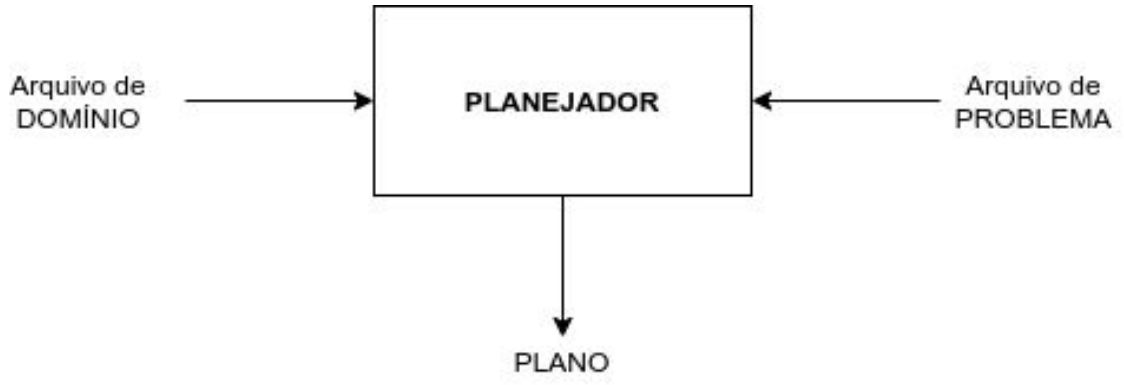

Figura 7: Fluxo para geração de um plano de um problema descrito em PDDL.
Fonte: Cantoni, 2010.

De acordo com Marrella, PDDL é um padrão para formular uma representação compacta de um problema de planejamento PR = 〈I, G, PD〉, onde I é a descrição do estado inicial do mundo, G é o estado de meta desejado e PD é o domínio de planejamento. Um PD de domínio de planejamento é construído a partir de um conjunto de proposições que descrevem o estado do mundo em que o planejamento ocorre (um estado é caracterizado pelo conjunto de proposições verdadeiras) e um conjunto de ações que podem ser executadas no domínio.

As pré-condições e os efeitos são declarados em termos das proposições no PD, que podem ser representadas através de predicados booleanos. O PDDL também inclui a capacidade de definir objetos de domínio e de digitar os parâmetros que aparecem em ações e predicados. Em um estado, somente ações cujas pré-condições são cumpridas podem ser executadas. Os valores das proposições no PD podem mudar como resultado da execução de ações, que, por sua vez, levam o PD a um novo estado. Um planejador que recebe um problema de planejamento PR é capaz de produzir automaticamente um plano P, isto é, um controlador que especifica quais ações são necessárias para transformar o estado inicial I em um estado que satisfaça a meta G.

De acordo com Cantoni, a PDDL é a linguagem oficial das competições internacionais de planejamento que acontecem de dois em dois anos dentro da principal conferência de planejamento da área, denominada *International Conference*

on Automated Planning and Scheduling (ICAPS[1]). O objetivo principal da linguagem, no contexto das competições, é permitir comparações de desempenho entre os planejadores em diferentes domínios. Devido ao sucesso obtido nas competições, a linguagem evoluiu de forma significativa e já é utilizada em aplicações reais.

Atualmente, existem cinco versões da PDDL. A cada versão novas características e mais expressividade são adicionadas à linguagem, permitindo que uma diversidade maior de problemas sejam especificados. As evoluções são propostas por um grupo de pesquisadores com amplo conhecimento da área. Após a publicação das novas características e da gramática, planejadores são desenvolvidos ou modificados para operarem sobre a nova versão.

2.3.1.1 Conceitos relacionados à PDDL

Para compreender a forma como os requisitos de *software* foram modelados em PDDL faz-se necessário conhecer os principais conceitos e definições relacionados a esta linguagem. Os componentes de um arquivo escrito em PDDL são:

- Estado Inicial: estado antes de iniciar o processo de planejamento;
- Estado Objetivo: estado com as propriedades que são as metas do planejamento;
- Objetos: lista de termos que serão usadas para instanciar os elementos do domínio, tais como predicados e ações;
- Tipo: tipo dos objetos;
- Predicados: fatos sobre os objetos que podem ser verdadeiros ou falsos;
- Ações: formas de mudar os estados e ir do estado inicial ao estado objetivo;
- Métrica: é utilizada para especificar quais funções do domínio deve-se tentar minimizar ou maximizar.

2.4 Ferramentas utilizadas

Nesta seção são apresentados os três *softwares* que foram utilizados como apoio para o desenvolvimento da ferramenta referente a este trabalho: um planejador, um analisador XML e um visualizador de grafos.

[1] Disponível em https://www.icaps-conference.org/. Acesso em 25 Fev. 2021.

2.4.1 Planejador

O planejador de código aberto escolhido para ser utilizado neste trabalho foi o Metric-FF[2] por apresentar um desempenho melhor que os demais planejadores testados em estudo realizado (Apêndice A).

O Metric-FF é um sistema de planejamento independente de domínio desenvolvido por Joerg Hoffmann. O sistema é uma extensão do planejador FF(Fast-Forward) para variáveis de estado numéricas e é implementado na linguagem C. Esse planejador participou dos domínios numéricos da 3ª Competição Internacional de Planejamento demonstrando desempenho bastante competitivo.

2.4.2 Analisador XML

O analisador XML escolhido para ser utilizado neste trabalho foi a biblioteca Expat[3] cujas primeiras versões foram lançadas pelo desenvolvedor de *software* James Clark em 1998. Em 2000 o projeto da Expat foi entregue a um grupo liderado por Clark Cooper e Fred Drake. O novo grupo lançou a versão 1.95.0 em setembro de 2000 e continua a lançar novas versões para incorporar correções de defeitos e melhorias.

Como um dos primeiros analisadores XML de código aberto disponíveis, Expat encontrou um lugar em muitos projetos de código aberto. Esses projetos incluem o servidor HTTP Apache, Mozilla, Perl, Python e PHP. Também está disponível em muitos outros idiomas.

Para usar a biblioteca Expat, os programas primeiro registram funções de manipulador com Expat. Quando Expat analisa um documento XML, ele chama os manipuladores registrados à medida que encontra *tokens* relevantes no fluxo de entrada. Esses *tokens* e suas chamadas de manipulador associadas são chamados de eventos. Normalmente, os programas registram funções de manipulador para eventos de início ou parada de elemento XML e eventos de caractere. Embora Expat seja principalmente um analisador baseado em fluxo (*push*), ele oferece suporte para interromper e reiniciar a análise em momentos arbitrários, tornando a implementação de um analisador *pull* relativamente fácil também.

---

[2] Disponível em https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html. Acesso em 25 Fev. 2021.
[3] Disponível em https://libexpat.github.io/. Acesso em 25 Fev. 2021.

2.4.3 Visualizador de grafos

A visualização de grafos é uma forma de representar informações estruturais como diagramas de grafos abstratos e redes. Possui importantes aplicações em redes, bioinformática, engenharia de *software*, banco de dados e web design, aprendizado de máquina e em interfaces visuais para outros domínios técnicos.

O visualizador de grafos escolhido para ser utilizado neste trabalho foi o Graphviz[4]. Graphviz é um *software* de visualização de grafos de código aberto que obtêm descrições de grafos em uma linguagem de texto simples e fazem diagramas em formatos úteis, como imagens e SVG para páginas da web; PDF ou Postscript para inclusão em outros documentos; ou exibir em um navegador gráfico interativo. Graphviz possui muitos recursos úteis para diagramas concretos, como opções de cores, fontes, layouts, estilos de linha, hiperlinks e formas personalizadas. A Figura 8 mostra um exemplo de layout gerado pela biblioteca e que será usado como padrão para os resultados obtidos nesta pesquisa.

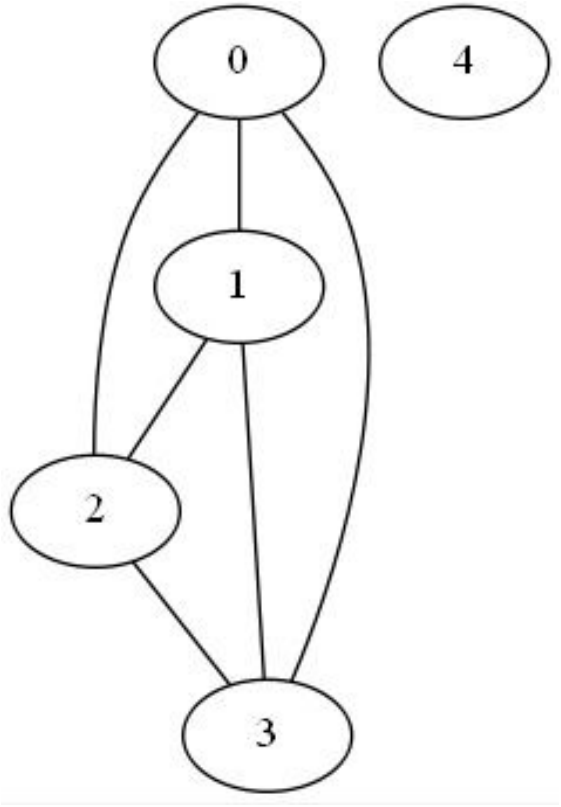

Figura 8. Exemplo de formato de saída gerado pela biblioteca Graphviz.

[4] Disponível em https://graphviz.org/. Acesso em 25 Fev. 2021.

2.5 Considerações finais do capítulo

Este capítulo apresentou os seguintes conceitos: processos de desenvolvimento de *software,* problema da próxima versão, abordagem incremental, planejamento em IA, PDDL. A integração desses conceitos possibilita o entendimento das motivações que levaram à implementação da ferramenta que é o objetivo deste estudo. Também foi feita uma introdução ao planejador Metric-FF, responsável por gerar o plano a partir dos arquivos de domínio e problema gerados em PDDL e às bibliotecas Expat e Graphviz, responsáveis por interpretar a lista de requisitos em XML passada como entrada para a ferramenta e por gerar uma imagem em PDF do plano gerado, respectivamente.

CAPÍTULO 3

METODOLOGIA

Esse capítulo apresenta a metodologia usada durante o desenvolvimento da ferramenta. A Seção 3.1 apresenta a metodologia aplicada para decidir qual planejador seria utilizado na ferramenta e como foi feita a modelagem dos requisitos em PDDL. A Seção 3.2 mostra como foi desenvolvida a parte da interação da ferramenta com os usuários e quais *softwares* e bibliotecas foram utilizados nesse processo. Por fim, a Seção 3.3 apresenta a validação do trabalho e as considerações finais do capítulo estão na Seção 3.4.

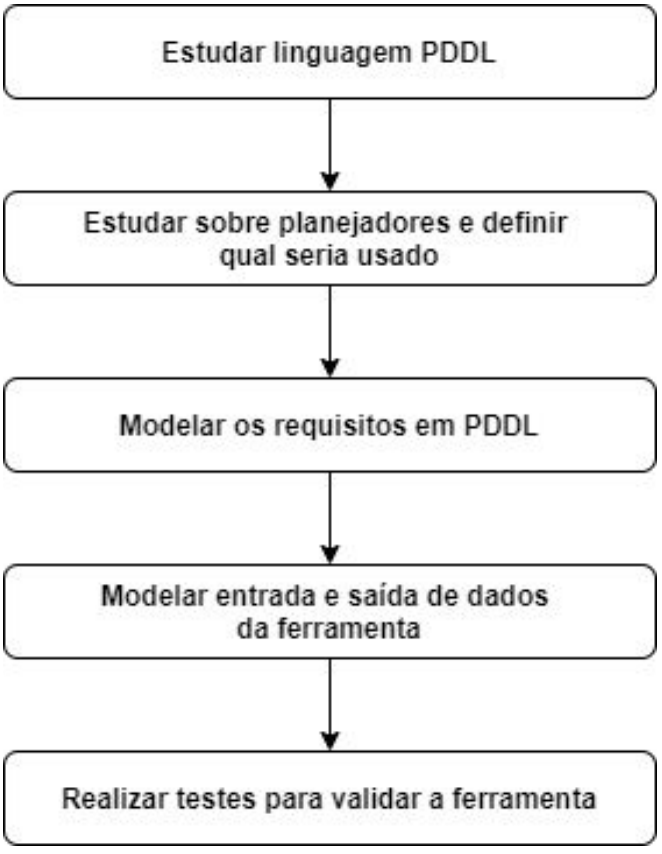

Figura 9: Fluxo de atividades relacionadas à metodologia.

3.1 Estudo de planejadores e implementação dos arquivos em PDDL

A primeira etapa da pesquisa consistiu em entender como funcionam os planejadores de *software* livre que estão disponíveis atualmente e em aprender a programar na linguagem PDDL para modelar uma lista de requisitos de *software* em uma entrada que pudesse ser interpretada por esses planejadores.

3.1.1 Definir ferramenta planejador

Foi realizado o levantamento de cinco planejadores diferentes que possivelmente gerariam um plano que maximizaria a prioridade, levando em conta o tamanho limite da versão e precedências entre os requisitos. Dentre eles, apenas dois conseguiram receber a lista de requisitos modelada em PDDL 2.1 como entrada, priorizá-los levando em consideração os parâmetros necessários e gerar uma saída: Metric-FF e SMTPlan.

Para decidir entre os dois planejadores foram realizados testes de performance utilizando três casos de teste:

1) Um caso básico com o intuito de gerar um problema simples, sem induzir a um caso complexo;
2) Um caso onde um requisito de baixa prioridade ocupa grande parte do tamanho da sprint;
3) Um caso para verificar como os planejadores se comportam nas situações onde existem requisitos de alta prioridade que dependem de requisitos de baixa prioridade.

Os detalhes e resultados obtidos durante esses testes podem ser encontrados no Apêndice A deste documento. Por fim, o planejador escolhido para ser utilizado no desenvolvimento desta pesquisa foi o Metric-FF.

3.1.2 Versões do Modelo

Primeiramente, foi necessário expressar as características dos requisitos (tamanho, dependências e prioridade) na linguagem PDDL para que o planejador Metric-FF pudesse interpretá-las, uma parte fundamental no processo da geração dos planos. Para isso, foram testadas diversas maneiras de como representar os requisitos em PDDL. Aqui serão apresentadas as principais diferenças entre as versões iniciais do modelo e a versão final.

3.1.2.1 Modelagem dos requisitos

Em uma primeira tentativa, cada requisito foi representado através de uma ação (*action*) no arquivo domain.pddl. Essa ação tinha como pré-condições que o requisito ao qual se referia ainda não tivesse sido tratado, que os requisitos dos quais ele depende já tivessem sido tratados e que a capacidade total disponível do time de desenvolvimento (que era decrementada para cada requisito tratado) ainda fosse maior ou igual ao tamanho do requisito ao qual a ação se refere. O efeito (*effect*) gerado pela ação seria atualizar o predicado referente ao requisito em questão para dizer que ele já havia sido tratado e decrementar a capacidade do time de desenvolvimento.

Além de uma ação para cada requisito que teria que ser tratado, o arquivo de domínio também englobava uma ação de estado final cuja pré-condição era que a capacidade do time fosse igual a zero, ou seja, que não houvesse espaço na versão para mais tarefas, ou que todos os requisitos já tivessem sido tratados.

No caso do arquivo de problema, a seção início (*init*) declarava um predicado para cada requisito (falso para quando o requisito ainda não havia sido tratado e verdadeiro caso contrário) e armazenava a capacidade total do time de desenvolvimento. A seção objetivo (*goal)* afirmava que o objetivo do plano era que o estado final (ação declarada no arquivo de domínio) fosse tratado, ou seja, que todos os requisitos tivessem sido tratados ou que a capacidade do time tivesse se esgotado.

Com relação às características de dependências, tamanho e prioridade, a primeira era representada na ação de cada requisito e era considerada um pré-requisito para que ele fosse concluído, o segundo também era armazenado nas

ações e utilizados para decrementar a capacidade total do time de desenvolvimento e a terceira era tratada uma a uma, em ordem decrescente, através de *tokens*.

Por fim, na versão atual do projeto foi possível modelar uma única ação generalizada que trata todos os requisitos reduzindo a quantidade de código duplicado. Também foi aplicado um sistema de atribuição e soma de valores por prioridade, além do uso dos *tokens* que foram mantidos, de modo que maximizar a soma desses valores faz parte do objetivo do plano. Com relação às dependências e tamanho dos requisitos, as primeiras passaram a ser tratadas como predicados no arquivo de problema e para o segundo foi criada uma função que retorna o valor referente a cada requisito.

Dessa maneira, o código fonte da ferramenta ficou mais limpo, as prioridades foram melhor tratadas e os resultados foram mantidos.

3.2 Modelagem da interação com os usuários

A segunda etapa da pesquisa consistiu em modelar como seria feita a interação da ferramenta com os usuários e como o plano gerado pelo planejador seria devolvido de maneira para que fosse possível entendê-lo sem muito esforço por parte dos usuários.

3.2.1 Modelagem da entrada de dados da ferramenta

Inicialmente, pensou-se em associar os arquivos desenvolvidos em PDDL e o planejador diretamente em algum *software* onde fosse possível modelar os requisitos através de desenhos ou grafos. Assim, os usuários fariam uma representação dos requisitos e das dependências existentes entre eles e essa modelagem seria salva em um arquivo com extensão XML cujas informações seriam tratadas e passadas como entrada para o planejador. Nesse momento foi realizada uma tentativa de usar o Freeplane[5], um programa de *software* livre muito utilizado para criar mapas mentais. Todavia, isso limitaria as opções dos usuários uma vez que seria obrigatório o uso do Freeplane, ferramenta que apesar de possuir uma boa documentação não é um *software* tão intuitivo e demandaria certo esforço dos usuários para que fosse usado no começo.

[5] Disponível em https://www.freeplane.org/wiki/index.php/Home. Acesso em 25 Fev. 2021.

Portanto, optou-se por manter a entrada como sendo um arquivo em extensão XML, que é bastante comum e utilizada em diversas aplicações, mas removeu-se a obrigatoriedade do uso do Freeplane ou de qualquer outro programa. Assim, os usuários podem usar qualquer ferramenta de modelagem de grafos ou mapas mentais que tiverem à disposição e exportá-la em XML ou, até mesmo, criar um arquivo nessa mesma extensão manualmente.

3.2.2 Modelagem da saída de dados da ferramenta

Uma vez gerado pelo planejador Metric-FF, o plano com todos os requisitos priorizados e separados por versão é convertido em um arquivo em extensão .dot, ou seja, em uma linguagem de descrição de grafos. Por fim, a biblioteca Graphviz é chamada para converter este arquivo .DOT para PDF com o objetivo de facilitar a leitura e o entendimento para o usuário. Maiores detalhes e exemplos dos arquivos envolvidos nesse processo de apresentação do plano gerado para o usuário podem ser encontrados no Capítulo 4 onde é apresentada toda a arquitetura da ferramenta.

3.3 Validação e testes

A terceira etapa do trabalho consiste em validar as funcionalidades da ferramenta através de testes, os quais são apresentados e discutidos nesta seção.

3.3.1 Validação da ferramenta

A fim de validar a ferramenta, garantindo a geração de um plano correto como saída (dada uma entrada válida pelo usuário), foram realizados testes para verificar o comportamento da ferramenta e o plano gerado em relação a quatro parâmetros: prioridade, dependência entre requisitos, tamanho máximo de plano por equipe e versão e balanceamento entre equipes.

Como saída, espera-se um plano de desenvolvimento no qual os requisitos estejam separados em uma ou mais entregas de tamanho menor ou igual ao tamanho máximo (configurado pelo usuário). Além disso, os requisitos de maior prioridade devem estar na frente dos requisitos de menor prioridade, mas sempre respeitando as relações de dependências. Outro ponto importante é garantir, quando possível e especificado pelo usuário através dos parâmetros de linha de comando, o balanceamento do número de requisitos atribuídos a cada equipe.

Os testes realizados são apresentados na próxima seção deste documento. Com eles é possível verificar que todos os objetivos específicos definidos no Capítulo 1 foram cumpridos de forma satisfatória.

3.3.2 Testes realizados

Com o intuito de verificar o funcionamento da ferramenta, foram criados diversos casos de teste que serão apresentados nesta subseção. Alguns deles foram pensados para induzir a casos mais complexos. Em todos os casos o número de prioridades foi limitado em cinco, sendo que a maior prioridade foi codificada com valor um (1) e a menor com valor cinco (5).

Todos os testes foram realizados em uma máquina virtual com Intel i5-9300H CPU @ 2.40GHz × 4, 4GB de memória RAM e sistema operacional Ubuntu 20.04.LTS.

3.3.2.1 Caso de teste 1 - Priorização de requisitos

O primeiro caso de testes procurou testar a priorização de requisitos. Foi usada uma entrada com cinco requisitos, cada um com uma prioridade diferente. O tamanho máximo por plano foi configurado com três.

Considere o conjunto de requisitos R={A, B, C, D, E} com tamanhos T={3, 3, 1, 1, 1} e prioridades P={1, 2, 3, 4, 5}, respectivamente, conforme Figura 10. Nesse caso, não há dependência entre os requisitos.

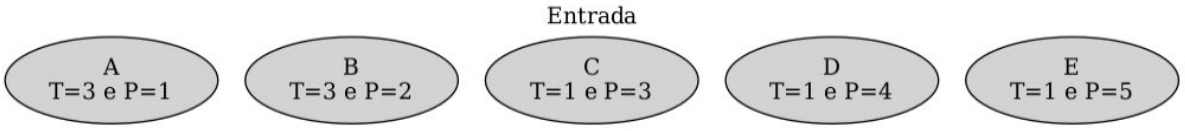

Figura 10. Caso de teste 1.

O plano gerado é mostrado na Figura 11. É possível verificar que a ferramenta priorizou os requisitos corretamente. Seria possível colocar os requisitos C, D e E na primeira versão, entretanto dessa maneira os requisitos de maior prioridade (A e B) seriam implementados apenas em versões futuras. O tempo de execução foi de 0,075.

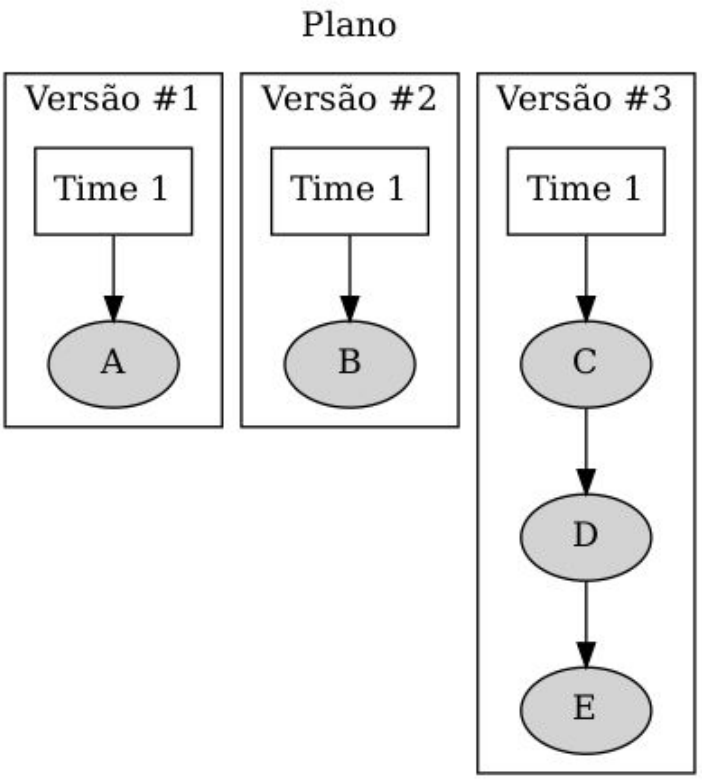

Figura 11. Resultado do caso de teste 1.

3.3.2.2 Caso de teste 2 - Dependências entre requisitos

O segundo caso de testes procurou verificar se a ferramenta está respeitando as relações de dependência entre os requisitos. Ele foi testado em dois casos, com uma única equipe e com duas equipes. Além disso, o tamanho máximo foi configurado com três nos dois casos, e a opção de balanceamento foi ativada no caso com duas equipes.

Esse caso de teste consiste no conjunto de requisitos R={A, B, C, D, E, F} com tamanhos T={1, 1, 1, 1, 1, 1} e prioridades P={1, 1, 1, 1, 1, 1}, respectivamente. As dependências (juntamente com tamanhos e prioridades) estão representadas no grafo da Figura 12, onde uma aresta *(r, r')* representa dependência do requisito *r* por *r'*.

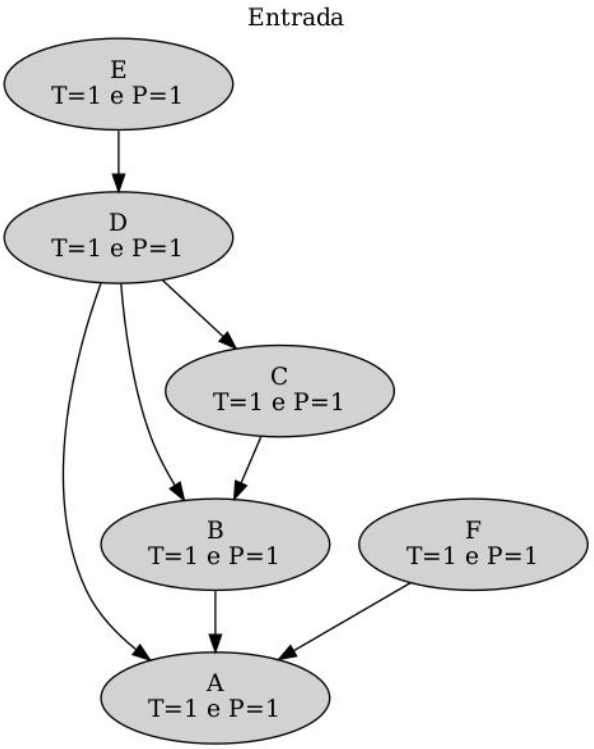

Figura 12. Caso de teste 2. Uma aresta *(r, r')* representa dependência do requisito *r* por *r'*.

O resultado para o caso com uma única equipe é apresentado na Figura 13. Foi gerado um plano com 2 entregas, respeitando o tamanho máximo, todas as dependências e priorizando corretamente. O tempo de execução foi de 0,074s.

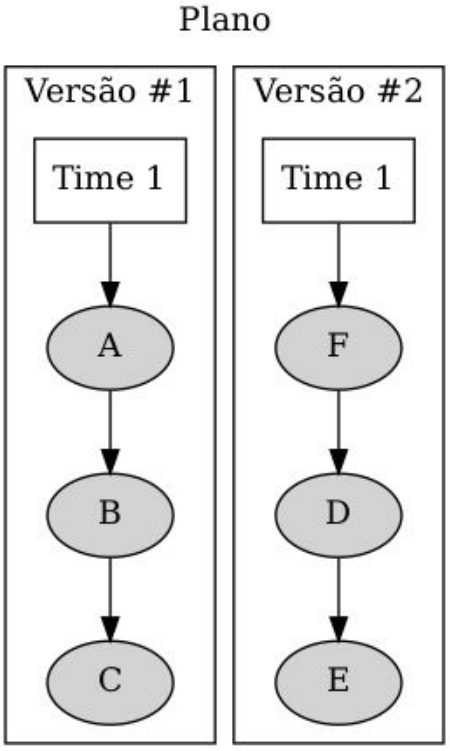

Figura 13. Resultado do caso de teste 2, com uma única equipe.

Para duas equipes, o resultado é apresentado na Figura 14. Foi gerado um plano com 2 entregas, respeitando o tamanho máximo de sprint, todas as dependências e priorizando corretamente. Na primeira entrega, apenas um time recebe requisitos, pois as dependências não permitem desenvolvimento paralelo antes do requisito A estar desenvolvido. O tempo de execução foi de 0,109s.

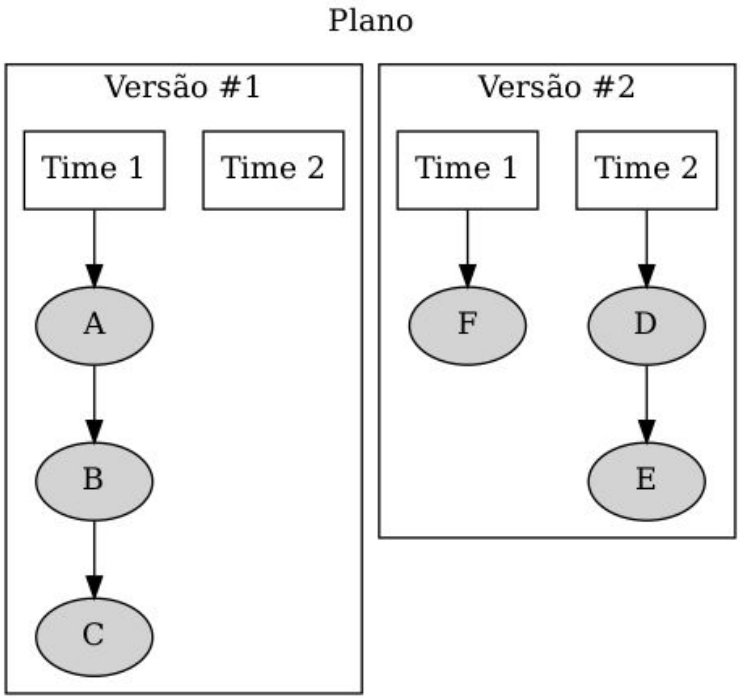

Figura 14. Resultado do caso de teste 2, com duas equipes.

3.3.2.3 Caso de teste 3 - Tamanho

O terceiro caso de testes procurou verificar se a ferramenta está respeitando o tamanho máximo de cada versão. Ele foi testado em dois casos, com uma única equipe e com duas equipes. Além disso, o tamanho máximo foi configurado com doze e a opção de balanceamento foi ativada.

Considere o conjunto de requisitos R={A, B, C, D, E, F, G} com tamanhos T={3, 3, 6, 4, 5, 5, 9} e prioridades P={1, 1, 1, 1, 1, 1, 1}, respectivamente, conforme Figura 15. Não há relações de dependências nesse caso.

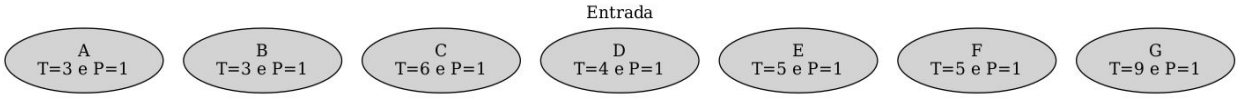

Figura 15. Caso de teste 3.

O resultado para o caso com uma única equipe é apresentado na Figura 16. Foi gerado um plano com 4 entregas, respeitando o tamanho máximo corretamente. O tempo de execução foi de 0,074s.

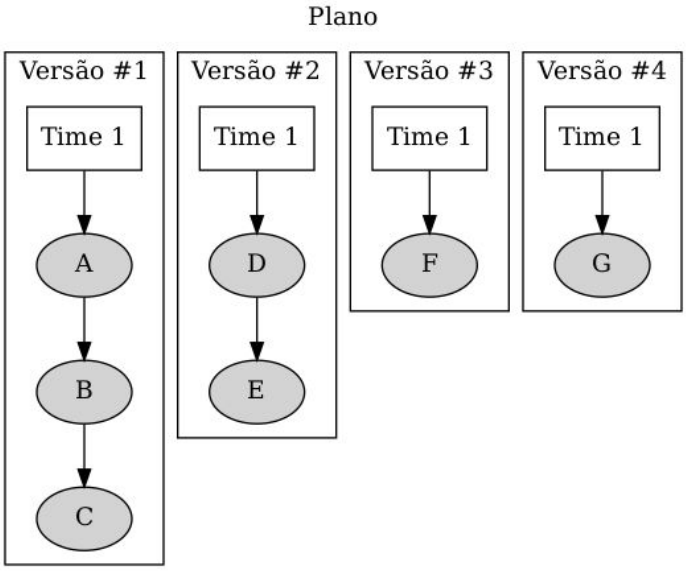

Figura 16. Resultado do caso de teste 3, para uma equipe.

Para duas equipes, o resultado é apresentado na Figura 17. Foi gerado um plano com 2 entregas, respeitando o tamanho máximo do plano corretamente. O tempo de execução foi de 0,117s.

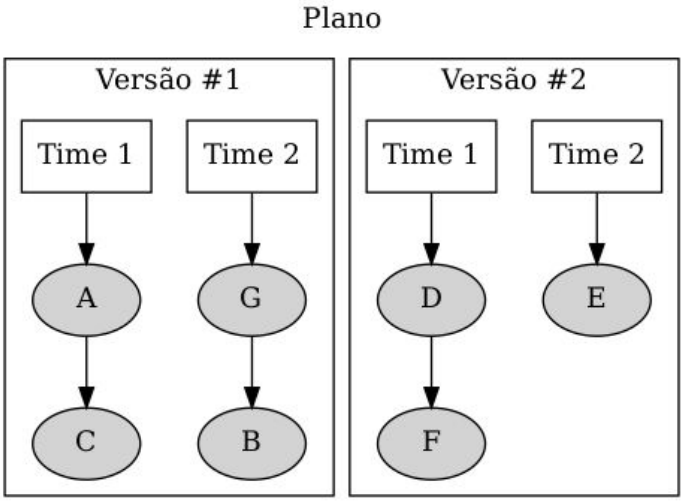

Figura 17. Resultado do caso de teste 3, com duas equipes.

3.3.2.4 Caso de teste 4 - Balanceamento entre equipes

O quarto caso de testes procurou testar o balanceamento entre os times nos casos em que a opção de balanceamento está ligada e quando está desligada. A entrada usada permite que todos os requisitos sejam desenvolvidos por uma única equipe, em uma única versão. Foi usada uma entrada com oito requisitos, todos

com tamanho um e prioridade um. O tamanho máximo de versão foi configurado com oito e de equipes com quatro 4.

Considere o conjunto de requisitos R={ A, B, C, D, E, F, G, H } com tamanhos T={ 1, 1, 1, 1, 1, 1, 1, 1 } e prioridades P={ 1, 1, 1, 1, 1, 1, 1, 1 }, respectivamente, conforme Figura 18. Nesse caso, não há dependência entre os requisitos.

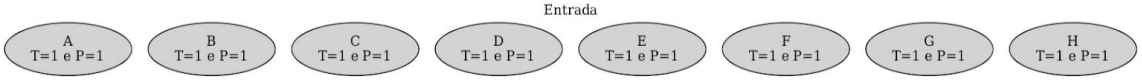

Figura 18. Caso de teste 4.

O resultado com a opção de balanceamento ligada é mostrado na Figura 19. A ferramenta dividiu os requisitos entre as equipes de maneira correta.

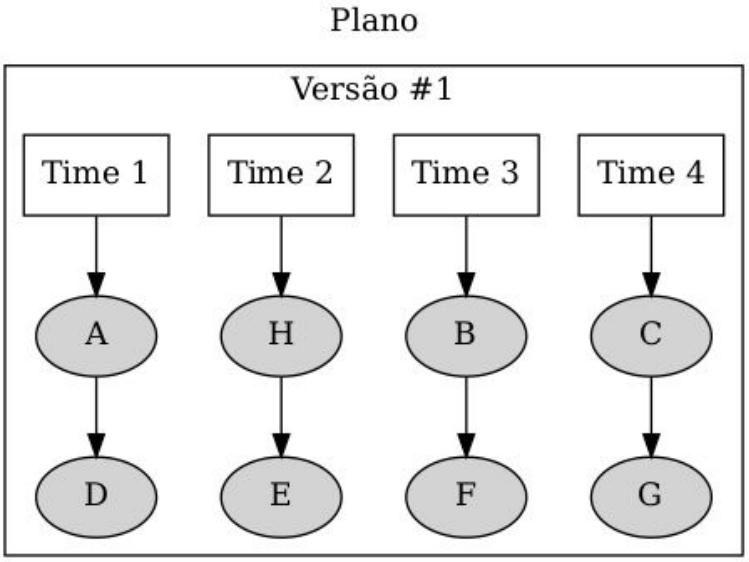

Figura 19. Resultado do caso de teste 4 com a opção de balanceamento ligada.

O resultado para o caso sem balanceamento é mostrado na Figura 20. Nesse caso, a ferramenta minimiza o número de equipes usadas.

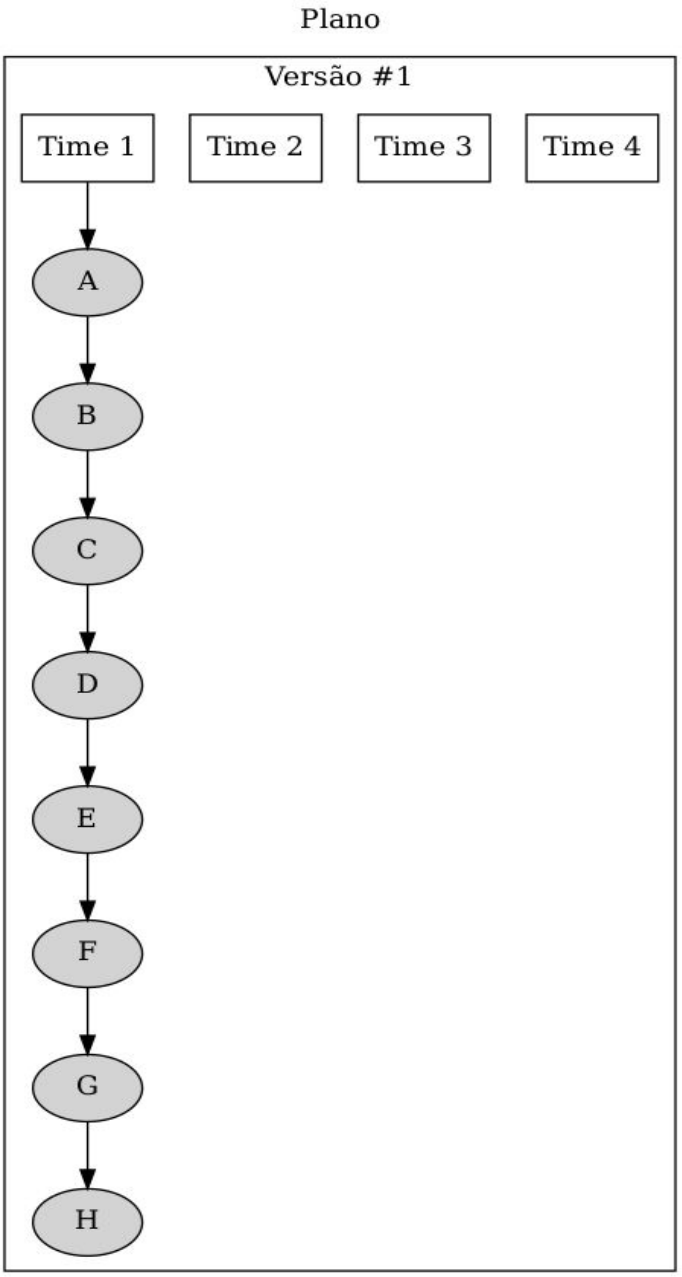

Figura 20. Resultado do caso de teste 4 com a opção de balanceamento desligada.

3.4 Considerações finais do capítulo

Este capítulo apresentou as atividades realizadas para o desenvolvimento da ferramenta: como foi escolhido o planejador utilizado e as versões de modelo criadas para tratar os requisitos. Também foram apresentados os testes realizados para validar o funcionamento do projeto através de casos de uso baseados nos objetivos específicos desta pesquisa, apresentados no Capítulo 1 deste documento. A arquitetura final da ferramenta e os testes gerais são apresentados no próximo capítulo.

CAPÍTULO 4

RESULTADOS

Este capítulo está organizado em quatro seções principais. A Seção 4.1 mostra como as características dos requisitos utilizadas como parâmetros para a ferramenta foram tratadas no modelo. A seção 4.2 descreve a arquitetura do método proposto, visa definir seus parâmetros de funcionamento e detalhar as camadas e atividades de processamento necessárias desde sua entrada até a obtenção da saída. Uma visão geral é exibida nas Figuras 22 e 23. A Seção 4.3 descreve um estudo de caso onde a ferramenta foi aplicada em um projeto real e as considerações finais do capítulo estão na Seção 4.4. A ferramenta foi disponibilizada no *link*: https://github.com/gbessern/NRP.

4.1 Modelo

Foram criadas estratégias para tratar as diferentes propriedades do problema abordado por esse trabalho: dependência entre requisitos, tamanho de versão e de requisitos, prioridade dos requisitos e balanceamento do número de requisitos atribuídos a cada equipe. Cada solução para tratar cada propriedade é apresentada nesta subseção.

4.1.1 Dependência

As dependências dos requisitos foram representadas por predicados no formato "*dependent ?r ?req - requirement*", onde *?r* e *?req* são objetos do tipo *requirement*. Como exemplo, considere dois requisitos, R01 e R02, e que o segundo depende do primeiro. Essa dependência seria modelada como "*dependent R02 R01*".

Além disso, para garantir que um requisito seja atribuído a uma equipe apenas quando todas as suas dependências tenham sido satisfeitas, foi adicionada uma regra nas pré-condições da ação responsável por delegar requisitos às equipes. Essa pré-condição tem o seguinte formato:

```
1.    forall (?req - requirement)
2.         (imply (dependent ?r ?req)
3.                (or (done_team ?req ?t) (done_all ?req))
4.         )
```

Sendo que o predicado “*done_team ?req ?t*” indica que o requisito *?req* foi atribuído ao time *?t* na versão corrente, mas ainda não foi entregue, portanto apenas o time *?t* pode receber o requisito *?r*. Por outro lado, o predicado “*done_all ?req*” indica que *?req* já foi entregue, portanto *?r* pode ser atribuído para qualquer equipe.

4.1.2 Tamanho

Os tamanhos dos requisitos foram representados pela função “*size ?r - requirement*". Para garantir que o tamanho máximo das versões sejam respeitados, foi adicionada a regra “*(>= (capacity ?t) (size ?r))*” nas pré-condições da ação responsável por delegar requisitos às equipes. Essa regra garante que a capacidade do time *?t* seja maior ou igual ao tamanho do requisito *?r*.

4.1.3 Prioridade

Cada nível de prioridade foi modelado como um objeto do tipo *priority*. Além disso, cada requisito teve sua prioridade representada com o predicado “*r_priority ?r - requirement ?p - priority*”, onde *?r* é um objeto do tipo *requirement* e *?p* um objeto do tipo *priority*.

Para que o planejador priorizasse a seleção de requisitos de forma correta foi criado um sistema de *token* e um de pesos.

4.1.3.1 *Token*

Foi usado o predicado “*priority ?p - priorities*”, onde *?p* é um nível de prioridade, para criar uma ordem de precedência entre os níveis de prioridade. Dessa forma, quando um nível de prioridade está com o *token*, ou seja, que faz o predicado “*priority ?p*” ser avaliado como verdadeiro, apenas requisitos desse mesmo nível de prioridade podem ser atribuídos para um time. Conforme os requisitos são desenvolvidos, o *token* é passado para o próximo nível de prioridade,

isto é, o predicado “*priority ?p*” passa a ser falso com o nível de prioridade corrente e o próximo nível de prioridade torna o predicado verdadeiro. O efeito responsável pela passagem do *token* do nível de prioridade 1 para o 2 é apresentado no código abaixo.

```
1.    (when (and (priority ?p - priorities))
2.         (and (not (priority P1)) (priority P2)))
```

4.1.3.2 Pesos

Além do sistema de token, foi atribuído a cada nível de prioridade um peso. Quanto maior o nível de prioridade, maior o peso recebido, conforme mostra o código:

```
1.         (= (weight P5) 1)
2.         (= (weight P4) 10)
3.         (= (weight P3) 100)
4.         (= (weight P2) 1000)
5.         (= (weight P1) 10000)
```

Dessa forma, quanto maior o nível de prioridade de um requisito, maior o seu peso. Além disso, foi criada uma função (“*sum*”) referente à soma dos pesos dos requisitos já atribuídos a uma equipe. Toda vez que um requisito é atribuído a uma equipe, o valor dessa função é incrementado com o peso do respectivo requisito. Assim, o estado objetivo do modelo é ter essa função avaliada com a soma dos pesos de todos os requisitos do problema. Como cada requisito só pode ser atribuído a uma equipe uma única vez, é necessário que todos sejam delegados à alguma equipe para atingir o estado final. Entretanto quanto maior o nível de prioridade de um requisito, mais próximo chega-se do estado objetivo quando atribuído a alguma equipe de desenvolvimento.

Como exemplo, considere um problema com 5 requisitos R = {A, B, C, D, E} com prioridades P={1, 1, 2, 4, 5}, respectivamente, conforme Figura 21. Para atingir o estado objetivo, a função “sum” deve ser igual a 21011 (10000*2 + 1000+ 10 +1).

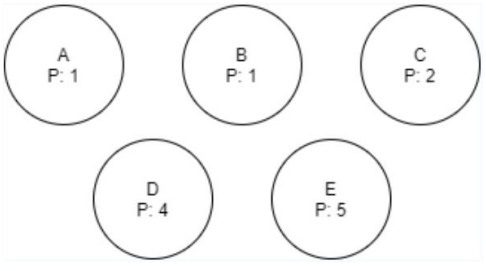

Figura 21. Exemplo de conjunto de requisitos.

4.1.4 Balanceamento

Nos casos de múltiplas equipes, é possível ligar a opção de balanceamento para equilibrar o número de requisitos atribuídos a cada equipe. Esse balanceamento foi feito com um sistema de *token*, similar à solução usada para prioridades. O predicado "*team ?t - team*", onde *?t* é um objeto do tipo "*team*", cria uma ordem de precedência entre os times. Apenas o time que torna o predicado verdadeiro pode receber requisitos. Além disso, foi criado um efeito passar o *token* de times, conforme o código abaixo.

```
1.    (when (and (team team1))
2.          (and (not (team team1)) (team team2)))
```

4.2 Arquitetura da ferramenta

A ferramenta foi idealizada com base nos conceitos de desenvolvimento de *software* e de planejamento em Inteligência Artificial abordados no Capítulo 2 deste documento.

A linguagem de programação utilizada foi a linguagem C. Esta é uma linguagem compilada e que possui características de linguagens estruturadas e procedurais. Além de ser uma das linguagens de programação mais populares, ela é simples de ser utilizada e encontra-se estável.

O desenvolvimento e os testes foram realizados no sistema operacional Linux usando a distribuição Ubuntu na versão 20.04. Para utilizar a ferramenta é necessária uma máquina onde estejam instalados o compilador GNU Compiler Collection (GCC), a biblioteca Expat, o pacote de ferramentas Graphviz, mencionados na seção 2.4 deste documento e o planejador Metric-FF introduzido na mesma seção das outras dependências.

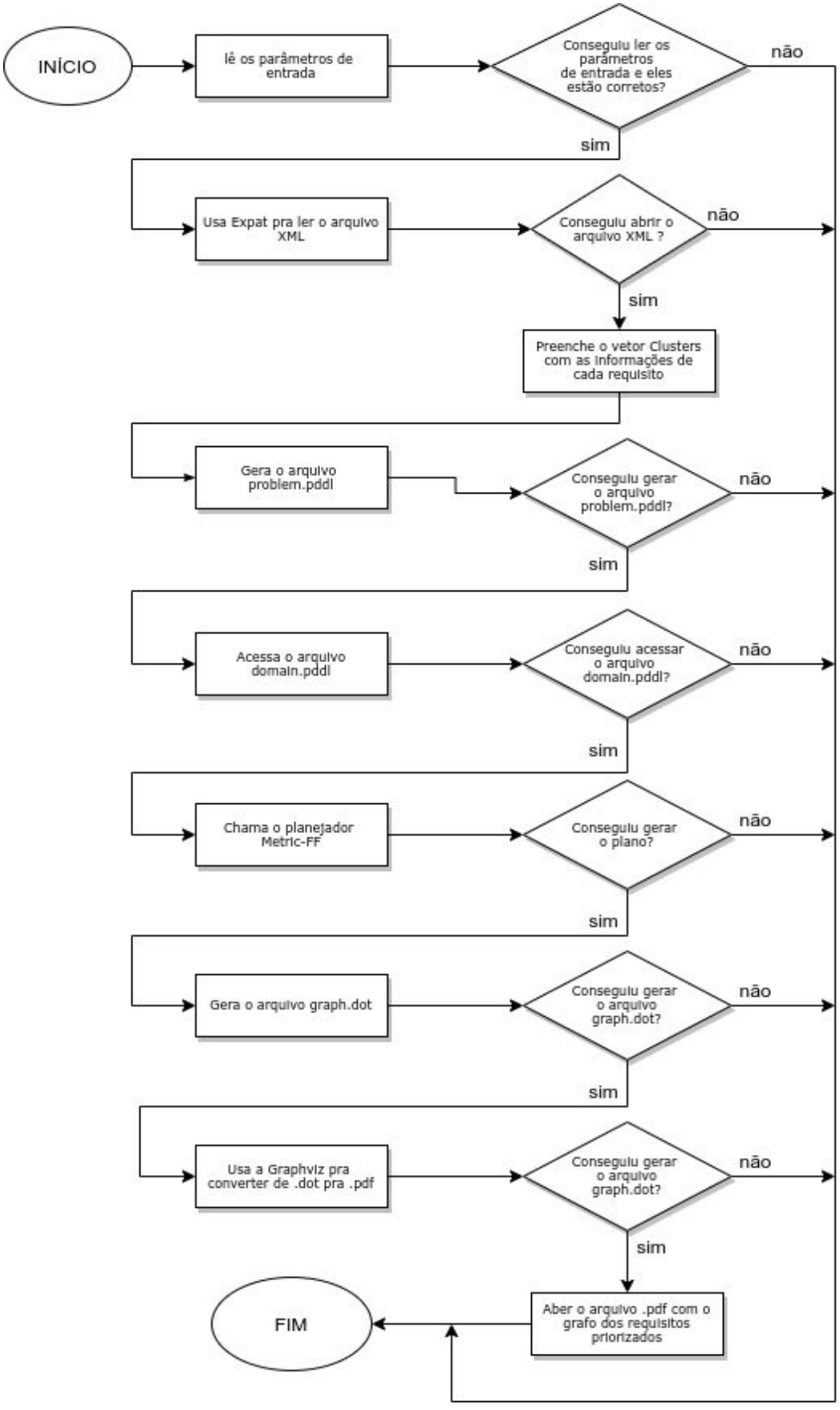

Figura 22: Diagrama de fluxo referente ao funcionamento da ferramenta desenvolvida.

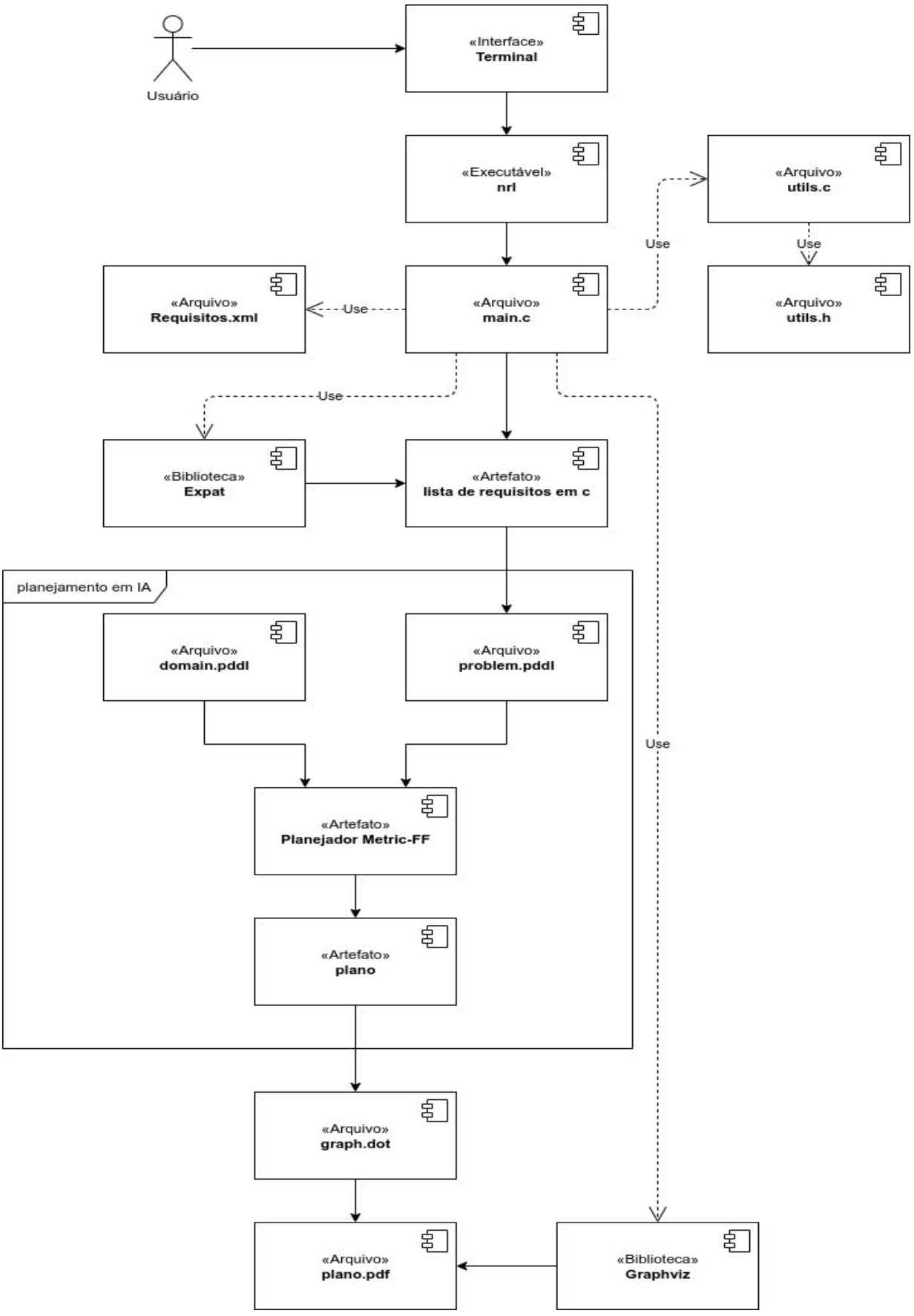

Figura 23: Diagrama de componentes referente ao funcionamento da ferramenta desenvolvida.

4.2.1 Processo de entrada de dados na ferramenta

Como exposto na Figura 22, as primeiras etapas do processamento da ferramenta são referentes à sua interação com o usuário e a leitura dos requisitos a serem priorizados. A entrada é composta por uma linha de comando que possui informações referentes ao arquivo executável da ferramenta, o tamanho máximo das versões onde os requisitos serão implementados, quantos níveis de prioridade deverão ser tratados, se deverá ser feito o balanceamento da quantidade de requisitos atribuídos a cada equipe e a quantidade de planos paralelos que devem ser gerados, ou seja, a quantidade de equipes disponíveis para trabalhar naquela versão; e por um arquivo em linguagem Extensible Markup Language (XML) que contém uma lista de requisitos formatados seguindo o seguinte padrão: o nome do requisito que está sendo definido seguido dos seus valores de prioridade e tamanho, respectivamente e, por último, uma lista do nome dos requisitos dos quais ele depende.

A Figura 24 exemplifica um caso onde os requisitos devem ser priorizados (as setas representam as dependências entre os requisitos) e a Figura 25 mostra um exemplo de como o arquivo XML deve estar estruturado.

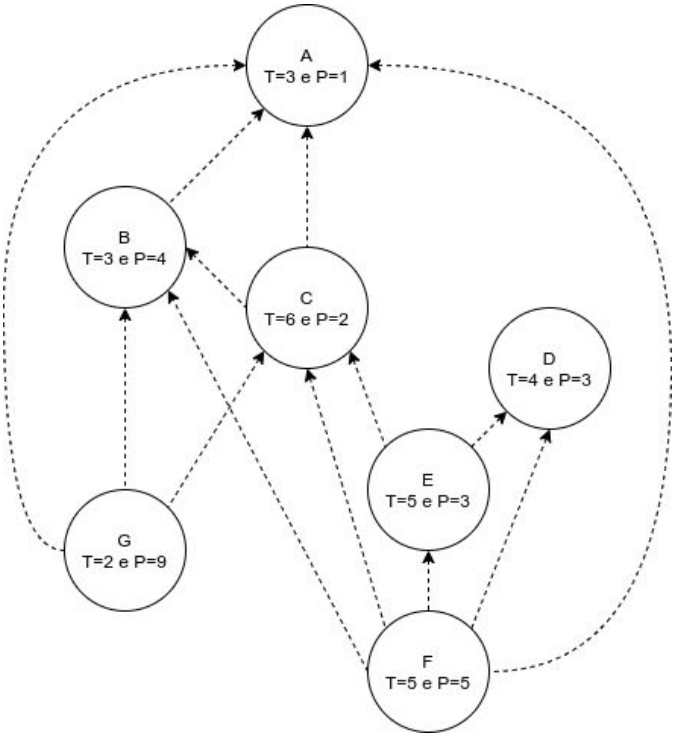

Figura 24. Exemplo de problema

```
1.    <?xml version="1.0" encoding="UTF-8"?>
2.    <EXEMPLO>
3.      <A prioridade="1" tamanho="3" dependencia="" />
4.      <B prioridade="4" tamanho="3" dependencia="A" />
5.      <C prioridade="2" tamanho="6" dependencia="A,B" />
6.      <D prioridade="3" tamanho="4" dependencia="" />
7.      <E prioridade="3" tamanho="5" dependencia="C,D" />
8.      <F prioridade="5" tamanho="5" dependencia="A,B,C,D,E" />
9.      <G prioridade="2" tamanho="9" dependencia="A,B,C" />
10.   </EXEMPLO>
```

Figura 25. Exemplo de arquivo de entrada com a lista dos requisitos a serem priorizados

A análise do arquivo XML e a conversão das informações para a linguagem de programação C é feita através da biblioteca Expat que foi introduzida na seção 2.4.2. Após serem lidos do arquivo XML os requisitos são separados por prioridade e salvos em listas encadeadas, como demonstrado na Figura 24, usando ainda o exemplo de entrada apresentado no código da Figura 25.

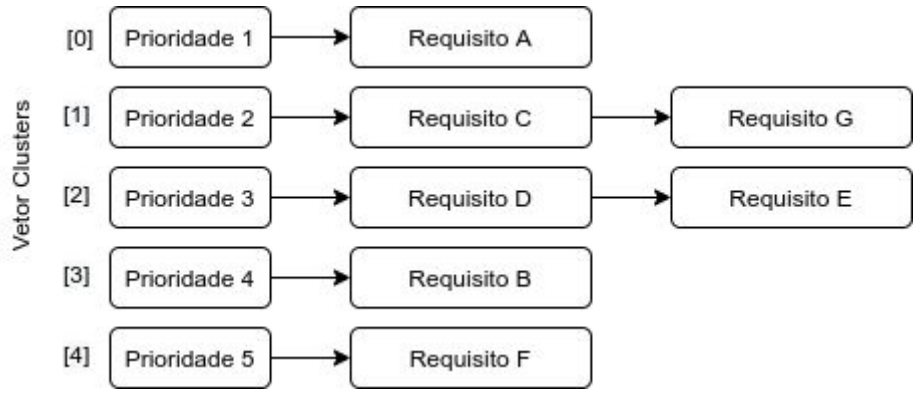

Figura 26: Representação das listas onde os requisitos da Figura 24 foram separados por prioridade.

4.2.2 Processo de criação do plano

Uma vez que os requisitos e suas características já estão salvos em um vetor na linguagem C foi possível gerar os arquivos problem.pddl e domain.pddl.

4.2.2.1 Arquivo de domínio

O arquivo de domínio define os aspectos imutáveis do problema, como tipos de objetos, predicados, funções e ações. Como exposto na Seção 2.3.1 deste

documento, uma das principais razões para que sejam criados dois arquivos distintos para o domínio e o problema é a possibilidade de um mesmo arquivo domain.pddl ser utilizado por diferentes instâncias (arquivos de problema). Entretanto, buscando proporcionar maior flexibilidade ao usuário, a ferramenta traz duas versões de domínio, uma que visa um plano que minimiza a quantidade de equipes usadas e outra que busca balancear a quantidade de requisitos atribuída a cada equipe. A estrutura do domínio que minimiza a quantidade de equipes usadas é apresentada abaixo. As regras adicionais do domínio que busca igualar a quantidade de requisitos entre as equipes estão marcadas com vermelho.

```
1.  (define (domain nrl)
2.          (:requirements :typing :fluents :negative-preconditions
    :disjunctive-preconditions)
3.          (:types requirement team priorities)
4.
5.          (:predicates
6.                  (done_all ?r - requirement)
7.                  (done_team ?r - requirement ?t - team)
8.                  (dependent ?r ?req - requirement)
9.                  (r_priority ?r - requirement ?p - priorities)
10.                 (priority ?p - priorities)
11.                 (team ?t - team)
12.         )
13.
14.         (:functions
15.                 (max_capacity)
16.                 (capacity ?t - team)
17.                 (size ?r - requirement)
18.                 (sum)
19.                 (weight ?p - priorities)
20.         )
```

Três tipos de objetos foram definidos (seção types): requisitos, times e prioridades (linha 3).

Foram criados cinco predicados para a versão sem balanceamento da quantidade de requisitos atribuída a cada equipe (linhas 5 a 12):

- *done_all ?r - requirement*: indica que o requisito r foi entregue;
- *done_team ?r - requirement ?t - team*: indica que o requisito r foi desenvolvido pela equipe t, porém ainda não foi entregue, estando visível apenas para o time t;

- *dependent ?r ?req - requirement*: indica que o requisito r depende do requisito req;
- *r_priority ?r - requirement ?p - priorities*: indica que o requisito r possui prioridade p;
- *priority ?p - priorities*: indica que os requisitos de prioridade p podem ser desenvolvidos.

Para o domínio que busca balancear a quantidade de requisitos atribuída a cada equipe foi criado um sexto predicado: *team ?t*. Ele indica que está na vez do time *t receber* um requisito.

Foram criadas cinco funções (linhas 14 a 20):

- *(max_capacity):* retorna a capacidade máxima por plano;
- *(capacity ?t - team)*: retorna a capacidade restante do time t;
- *(size ?r - requirement):* retorna o tamanho do requisito r;
- *(sum)*: retorna a soma dos pesos das prioridades dos requisitos desenvolvidos;
- *(weight ?p - priorities)*: retorna o peso de cada prioridade.

```
21. (:action DEV_REQ
22.              :parameters (?r - requirement ?t - team ?p - priorities)
23.              :precondition (and
24.                     (priority ?p)
25.                     (team ?t)
26.                     (r_priority ?r ?p)
27.                     (>= (capacity ?t) (size ?r))
28.                     (forall (?team - team)
29.                            (not (done_team ?r ?team))
30.                     )
31.                     (forall (?req  - requirement)
32.                            (imply (dependent ?r ?req) (or (done_team ?req ?t)
    (done_all ?req)))
33.                     )
34.              )
35.              :effect (and
36.                     (done_team ?r ?t)
37.                     (decrease (capacity ?t) (size ?r))
38.                     (increase (sum) (weight ?p))
39.                     (not (priority ?p))
40.                     (priority P1)
41.                     (when (and (team team1)) (and (not (team team1)) (team
    team2)))
42.                     (when (and (team team2)) (and (not (team team2)) (team
    team3)))
43.                     …
```

```
44.                     (when (and (team teamX)) (and (not (team teamX)) (team
    team1)))
45.
46.                 )
47.         )
48.
```

A ação *DEV_REQ* (linhas 21 a 43) é responsável por atribuir um requisito a uma equipe para desenvolvimento. Ela possui três parâmetros:

- ?r - *requirement*: o requisito a ser desenvolvido;
- ?t - *team:* o time que irá implementar o requisito ?r;
- ?p - *priorities*: uma prioridade.

As pré-condições são:

- *priority* ?p: Deve ser a vez dos requisitos de prioridade ?p serem desenvolvidos;
- r*_priority* ?r ?p: o requisito ?r deve ter prioridade ?p;
- >= (*capacity* ?t) (size ?r): o time ?t deve possuir capacidade para desenvolver o requisito ?r;
- *forall* (?*team* - *team*) (*not* (*done_team* ?r ?*team*)): nenhum time pode ter implementado o requisito ?r;
- *forall* (?req - *requirement*) (*imply* (*dependent* ?r ?req) (or (*done_team* ?req ?t) (*done_all* ?req))): garante que todos os requisitos dos quais ?r depende foram entregues ou desenvolvidos pelo time ?t.

No caso do domínio que busca balancear a quantidade de requisitos atribuída a cada equipe foi criado uma pré-condição adicional: *team* ?t. Ela garante que estava na vez do time ?t receber um requisito.

A ação possui cinco efeitos:

- *done_team* ?r ?t: indica que o time ?t desenvolveu o requisito ?r;
- decrease (*capacity* ?t) (*size* ?r): subtrai o tamanho do requisito ?r da capacidade do time ?t;
- *increase* (*sum*) (*weight* ?p): soma o peso da prioridade de ?r a soma total das prioridades dos requisitos desenvolvidos.
- *not* (*priority* ?r) e *priority* P1: volta a vez para os requisitos de maior prioridade.

Para o domínio com balanceamento, foram criados efeitos adicionais (marcadas em vermelho no código) para trocar a vez entre os times de receber requisitos.

```
49.       (:action NEW_SPRINT
50.             :parameters ()
51.             :precondition (and
52.                   (forall (?t - team)
53.                         (= (capacity ?t) 0)
54.                   )
55.             )
56.             :effect (and
57.                   (forall (?r - requirement ?t - team)
58.                         (when (and (done_team ?r ?t)) (and (done_all ?r)))
59.                   )
60.                   (forall (?t - team)
61.                         (and (assign (capacity ?t) (max_capacity))
62.                               (not (team ?t))
63.                         )
64.                   )
65.                   (team team1)
66.                   (forall (?p - priorities)
67.                         (not (priority ?p))
68.                   )
69.                   (priority P1)
70.             )
71.       )
```

A ação *NEW_SPRINT (linhas 49 a 71)* indica que o plano de uma versão está pronto e prepara o ambiente para planejar o plano de mais uma versão. Como pré-condição, todas as equipes devem estar com capacidade igual a zero. Os efeitos são:

- Tornar os requisitos desenvolvidos visíveis para todos os times, representando uma entrega;
- Atribuir a capacidade máxima a todos os times;
- Devolver a vez para os requisitos de maior prioridade;

No caso do domínio com balanceamento, efeitos adicionais foram criados para passar a vez entre os times.

```
72. (:action INCREASE_PRIORITY
73.             :effect (and
74.                   (when (and (priority P1)) (and (not (priority P1))
    (priority P2)))
```

```
                    (when (and (priority P2)) (and (not (priority P2))
(priority P3)))
                    (when (and (priority P3)) (and (not (priority P3))
(priority P4)))
                    (when (and (priority P4)) (and (not (priority P4))
(priority P5)))
                    (when (and (priority P5)) (and (not (priority P5))
(priority P0)))
             )
      )
```

- A ação *INCREASE_PRIORITY* é responsável por trocar a vez de desenvolvimento de cada prioridade.

```
      (:action CHANGE_TEAM
            :effect (and
                  (when (and (team team1)) (and (not (team team1)) (team
team1)))
                  (forall (?p - priorities)
                        (not (priority ?p))
                  )
                  (priority P1)
            )
      )
```

- A ação *CHANGE_TEAM* é responsável por trocar a vez de desenvolvimento de cada equipe. Ela está presente apenas no domínio com balanceamento.

```
      (:action RESET_CAPACITY
            :parameters (?t - team)
            :precondition (and (priority P0)
                  (team ?t)
            )
            :effect (and (assign (capacity ?t) 0)
                  (when (and (team team1)) (and (not (team team1)) (team
team2)))
                  (when (and (team team2)) (and (not (team team2)) (team
team3)))
                  …
                  (when (and (team teamX)) (and (not (team teamX)) (team
team1)))
                  (not (priority P0))
                  (priority P1)
            )
      )
```

- A ação *RESET_CAPACITY* é responsável por zerar a capacidade do time t, indicando que ?t não receberá mais requisitos para desenvolver na versão para a qual o plano está sendo gerado.

O arquivo pddl completo encontra-se no Apêndice B deste documento.

4.2.2.2 Arquivo de problema

Como foi relatado na Seção 2.3.1 deste documento, o arquivo de problema descreve a instância de um problema a ser resolvido. Dessa forma, é essencial que exista um arquivo diferente para cada problema e, portanto, foi necessário implementar a ferramenta de forma que, para cada nova instância a ser solucionada, um novo arquivo problem.pddl fosse gerado. Para isso, a modelagem deste arquivo é feita manualmente com base nas informações dos requisitos que se encontram salvos no vetor Cluster conforme exibido na Figura 25.

Conforme o que é apresentado na Seção 2.3.1, a definição do problema contém os objetos presentes na instância do problema, a descrição do estado inicial e o objetivo. Dessa forma, o arquivo problem.pddl foi modelado da seguinte maneira:

- Os objetos inicializados (a seção :*objects*) são os requisitos em si, declarados através dos mesmos nomes salvos no vetor Clusters, os times, ou seja, as equipes de desenvolvimento disponíveis e os níveis de prioridade referentes aos requisitos declarados.

```
1.         (:objects
2.         A B C D E F G - requirement
3.         team1 team2 - team
4.         P0 P1 P2 P3 P4 P5 - priorities
5.       )
6.
```

No caso deste exemplo são definidos nos requisitos A,B,C,D,E,F e G na linha 2, os times de desenvolvimento team1 e team2 na linha 3 e os níveis de prioridade de desenvolvimento P1, P2, P3, P4 e P5 na linha 4.

- A descrição do estado inicial (a seção: *init*) é simplesmente uma lista de todos os itens básicos que são verdadeiros no estado inicial.

```
7.
8.      (:init
9.          (= (sum) 0)
10.         (= (max_capacity) 12)
```

```
11.         (= (capacity team1) 12)
12.         (= (capacity team2) 12)
13.
14.         (priority P1)
15.
16.         (dependent B A)
17.         (dependent C B)
18.         (dependent E C)
19.         (dependent E D)
20.         (dependent F A)
21.         (dependent F B)
22.         (dependent F C)
23.         (dependent F D)
24.         (dependent F E)
25.         (dependent G A)
26.         (dependent G B)
27.         (dependent G C)
28.
29.         (= (size A) 3)
30.         (= (size B) 3)
31.         (= (size C) 6)
32.         (= (size D) 6)
33.         (= (size E) 5)
34.         (= (size F) 5)
35.         (= (size G) 9)
36.
37.         (r_priority A P1)
38.         (r_priority B P3)
39.         (r_priority C P2)
40.         (r_priority D P3)
41.         (r_priority E P3)
42.         (r_priority F P5)
43.         (r_priority G P2)
44.
45.         (= (weight P1) 10000)
46.         (= (weight P2) 1000)
47.         (= (weight P3) 100)
48.         (= (weight P4) 10)
49.         (= (weight P5) 1)
50.     )
```

Nesta seção, a linha 7 inicializa a variável “*sum*” (“soma”, em português), utilizada na busca por maximizar o valor da soma dos pesos atribuídos a cada prioridade nas linhas 43 a 47. Dessa forma as prioridades com peso maiores serão sempre priorizadas.

As linhas 9 e 10 inicializam as capacidades de trabalho dos times de desenvolvimento de modo que a soma do tamanho dos requisitos, definidos nas linhas 27 a 33, que serão elencados para cada equipe não ultrapassem

sua capacidade, ou seja, a quantidade de trabalho que elas conseguem realizar no período de uma *sprint*. A linha 8 define a capacidade máxima de cada plano por equipe, assim, quando a ação "nova versão" acontece, esse valor é passado para a capacidade de cada time.

Nas linhas 14 a 25 são definidas as dependências entre os requisitos sendo uma dependência por linha. A linha 14, por exemplo, afirma que o requisito B depende do requisito A.

Por fim, as linhas 35 a 41 deste exemplo relacionam cada requisito que será priorizado ao seu nível correspondente de prioridade sendo P1 a prioridade mais alta e P5 a prioridade mais baixa.

- A descrição do estado final (a seção: goal) é uma fórmula que define como um ou mais dos itens definidos no estado inicial e como eles devem se encontrar depois da execução do planejador.

```
51. (:goal
52.         (and (= (sum) 12301))
53. )
```

Neste caso, o estado objetivo é aquele onde a soma dos pesos das prioridades é igual a 12301.

O arquivo pddl completo gerado encontra-se no Apêndice B deste documento.

4.2.3 Processo de saída de dados na ferramenta

Como exposto na Figura 22, ao terminar a execução o planejador cria um arquivo novo com o resultado do planejamento efetuado. No caso do exemplo da Figura 24 o plano gerado pode ser encontrado no Apêndice B deste documento.

Uma vez gerado, este plano é lido pela ferramenta e as informações relevantes para o usuário são modeladas em um arquivo em extensão .DOT para que possa ser passado como entrada para a biblioteca Graphviz. O código abaixo mostra como ficou a modelagem do exemplo da Figura 23 em .DOT.

```
1.
2.  digraph NEXT_RELEASE_PROBLEM {
3.    node [style=filled];
4.    label="Plano";
5.    labelloc="t";
6.    subgraph cluster0 {
7.         T_10 -> A -> B -> C -> D -> E -> G -> H;
8.      label = "Release #1";
9.                T_10 [label="Time 1" shape=box style="solid"];
10.  }
11.   subgraph cluster1 {
12.        T_11 -> F -> I;
13.     label = "Release #2";
14.               T_11 [label="Time 1" shape=box style="solid"];
15.  }
16. }
```

Uma vez gerado o arquivo .DOT, a biblioteca Graphviz é chamada para modelar as informações lidas em um arquivo PDF que contém um conjunto de grafos que representam os requisitos priorizados conforme o que foi gerado pelo planejador. Essa saída mostra os requisitos separados no número de times informados pelo usuário via linha de comando e separados no número de versões necessárias para o desenvolvimento do projeto. A Figura 27 mostra a saída obtida com base no exemplo da Figura 24.

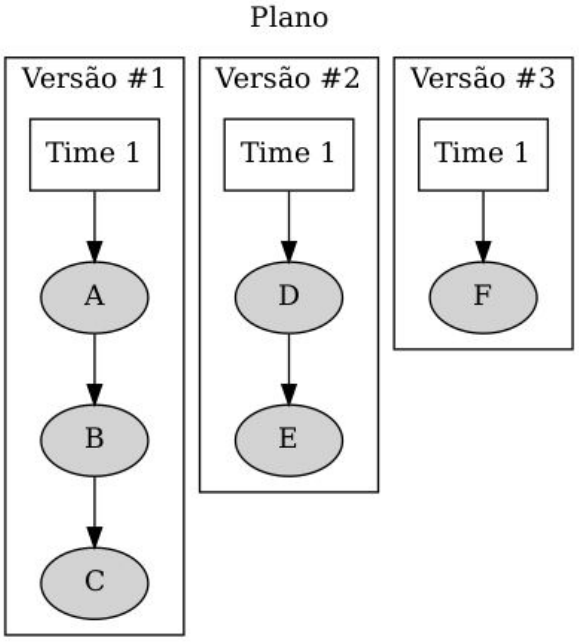

Figura 27. Exemplo de saída baseada na entrada da Figura 24.

4.3 Estudo de caso

A fim de verificar o comportamento da ferramenta quando aplicada em uma situação real, foi feito um estudo de caso baseado no trabalho de Carvalho & Dias (2019). O trabalho consiste em uma proposta de *software*, chamado Ambrosia, para auxiliar no gerenciamento de projetos por meio de soluções voltadas para planejamento de tarefas, atribuição dos recursos humanos e prazos, gerenciamento do cronograma geral e gerenciamento de impedimentos. Os requisitos funcionais do Ambrosia foram adaptados e usados como entrada da ferramenta (Tabela 1).

Nome	Descrição	Prioridade	Tamanho
[R01]	O Ambrosia deve fazer a identificação dos usuários: os usuários deverão fazer login como “gerente” ou “membro de equipe” antes de acessarem os recursos do sistema.	1	5
[R02]	O Ambrosia deve permitir que usuários logados como “gerente” cadastrem uma nova equipe.	2	4
[R03]	O Ambrosia deve permitir que usuários logados como “gerente” cadastrem uma nova tarefa.	4	4
[R04]	O Ambrosia deve permitir que usuários logados como “gerente” cadastrem um novo usuário.	2	4
[R05]	O Ambrosia deve permitir que usuários logados como “membro de equipe” criem um impedimento.	6	4

[R06]	O Ambrosia deve permitir que usuários logados como “gerente” alterem os membros de uma equipe.	3	3
[R07]	O Ambrosia deve permitir que ambos os tipos de usuários alterem uma tarefa.	4	3
[R08]	O Ambrosia deve permitir que usuários logados como “gerente” excluam uma tarefa.	4	2
[R09]	O Ambrosia deve permitir que usuários logados como “gerente” excluam um usuário.	2	2
[R10]	O Ambrosia deve permitir que usuários logados como “gerente” excluam uma equipe.	3	2
[R11]	O Ambrosia deve permitir que usuários logados como “gerente” excluam um impedimento.	6	2
[R12]	O Ambrosia deve permitir que ambos os tipos de usuários visualizem o cronograma do projeto.	5	3
[R13]	O Ambrosia deve permitir que ambos os tipos de usuários visualizem uma lista de tarefas.	4	3
[R14]	O Ambrosia deve permitir que usuários logados como “gerente” visualizem uma lista de usuários.	2	3

[R15]	O Ambrosia deve permitir que usuários logados como “gerente” visualizem uma lista de impedimentos.	6	3
[R16]	O Ambrosia deve permitir que usuários logados como “gerente” criem um cronograma.	5	4
[R17]	O Ambrosia deve permitir que usuários logados como “gerente” alterem o cronograma.	5	3

Tabela 1: Requisitos usados como caso de teste para a ferramenta.
Carvalho e Dias (2019).

Com relação à definição das prioridades de cada requisito, fixou-se prioridade 1 para o Requisito 01 ([R01]) sabendo que a maioria das funcionalidades do Ambrosia dependem do perfil do usuário logado no sistema. As demais prioridades foram alocadas por funções (usuários, equipes, tarefas e assim por diante):

- Requisitos relacionados ao gerenciamento de usuários receberam prioridade 2;
- Requisitos relacionados ao gerenciamento de equipes receberam prioridade 3;
- Requisitos relacionados ao gerenciamento de tarefas receberam prioridade 4;
- Requisitos relacionados ao gerenciamento do cronograma receberam prioridade 5;
- Requisitos relacionados ao gerenciamento de impedimentos receberam prioridade 6.

Alocar as prioridades dessa maneira tem como objetivo desenvolver uma função por vez, ou seja, priorizar que não sejam entregues pequenos pedaços de várias funcionalidades diferentes e sim uma funcionalidade inteira e funcional, aumentando o valor da entrega do incremento para o usuário final.

Com relação às dependências existentes entre os requisitos, seguiu-se o princípio das prioridades de forma que todos os requisitos dependem do primeiro ([R01]) e, para os demais, tem-se que requisitos de exclusão, alteração e visualização dependem dos requisitos de criação. A Figura 28 apresenta como ficaram as dependências entre todos os requisitos definidos na Tabela 1.

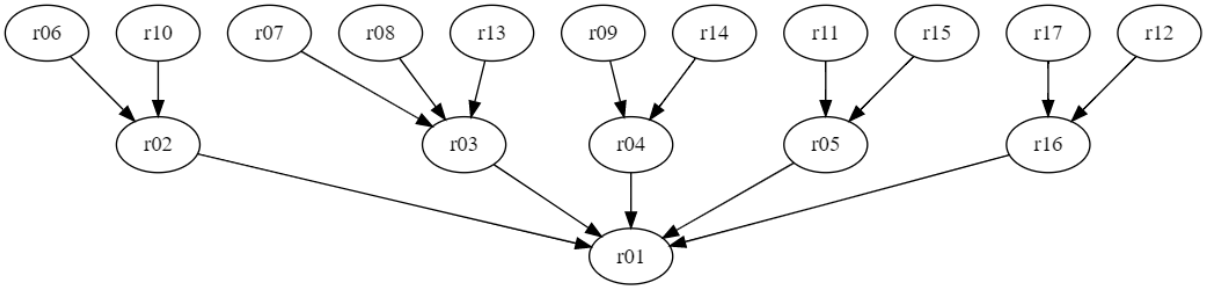

Figura 28. Relações de dependência entre os requisitos do sistema Ambrosia.

Com relação aos tamanhos utilizados para este teste, optou-se por utilizar uma versão de tamanho 10(duas semanas), que é um prazo bastante comum para projetos que utilizam o desenvolvimento incremental como metodologia. Os tamanhos dos requisitos foram definidos com base na quantidade de dias que seriam necessários para desenvolver cada funcionalidade. Assim, os tamanhos ficaram definidos da seguinte forma:

- Tamanho 5 para o primeiro requisito;
- Tamanho 4 para funcionalidades de criação;
- Tamanho 3 para funcionalidades de alteração e visualização;
- Tamanho 2 para funcionalidades de exclusão.

4.3.1 Resultados do estudo de caso

O estudo de caso foi feito de três maneiras: com uma equipe, com duas equipes e opção de balanceamento ligada, e com duas equipes e opção de balanceamento desligada. Todos os testes foram realizados em uma máquina virtual com Intel i5-9300H CPU @ 2.40GHz × 4, 4GB de memória RAM e sistema operacional Ubuntu 20.04.LTS. No Apêndice C são apresentados os domínio e

problema modelados em arquivos PDDL (*domain.pddl* e *problem.pddl)*, respectivamente.

O plano gerado para o caso com uma equipe é apresentado na Figura 29.

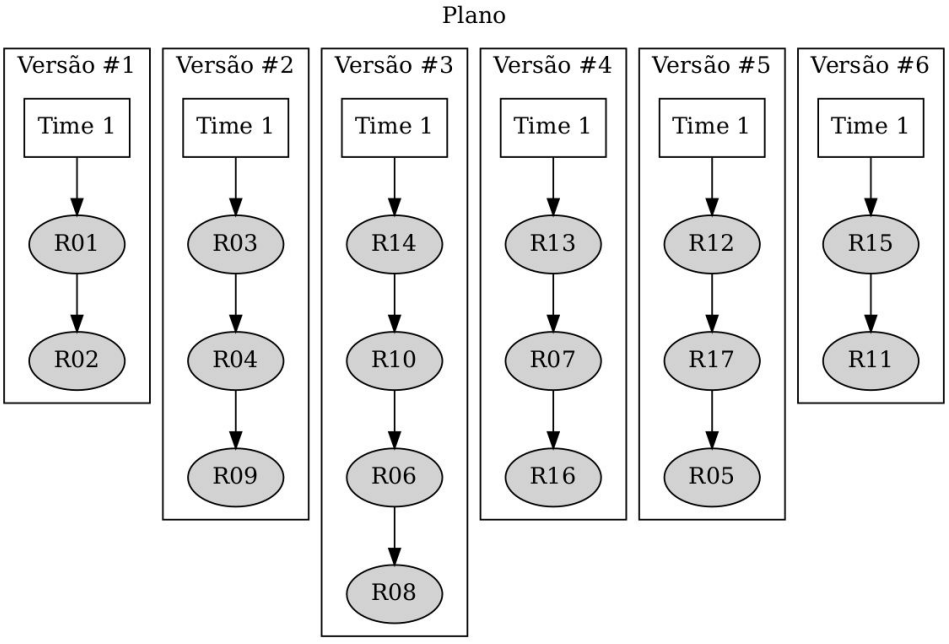

Figura 29. Resultado para o caso de estudo com uma equipe.

A ferramenta conseguiu gerar um plano de maneira satisfatória, com um tempo de execução de 83ms. Todas as dependências, prioridades e tamanho de versão foram respeitadas. Considerando que tamanho 10 equivale a duas semanas, nota-se que o planejador gerou como resultado um plano de projeto que terá como tempo de desenvolvimento cerca de 12 semanas, ou seja, aproximadamente 3 (três) meses.

A Figura 30 mostra o plano gerado para o caso com duas equipes e opção de balanceamento ligada. O tempo de execução foi de 1,8s. Assim como no caso do plano gerado para uma única equipe todos os parâmetros foram respeitados. Neste caso, o plano gerado mostra que o time 2 ficou sem requisitos atribuídos à ele na primeira versão. Isso acontece pois todos os requisitos do projeto dependem do primeiro ([R01]), conforme o que foi apresentado na seção anterior deste documento. Assim, nenhum requisito poderia estar sendo desenvolvido enquanto o

primeiro não estivesse pronto. Portanto, conclui-se que a ferramenta gerou o plano de maneira satisfatória.

Neste caso, com duas equipes disponíveis o tempo de desenvolvimento do projeto cairia para 8(oito) semanas, ou seja, cerca de 2 meses.

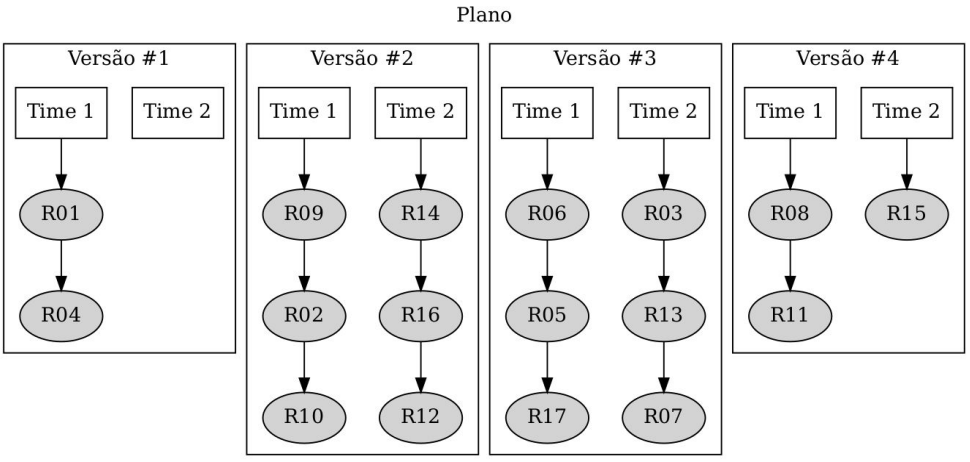

Figura 30. Resultado para o caso de estudo com duas equipes e opção de balanceamento ligada (quantidade de requisitos atribuídos a cada equipe é equilibrada quando possível).

Por fim, a Figura 31 mostra o plano gerado para o caso com duas equipes e opção de balanceamento desligada. O tempo de execução nesse caso foi de 2,5s.

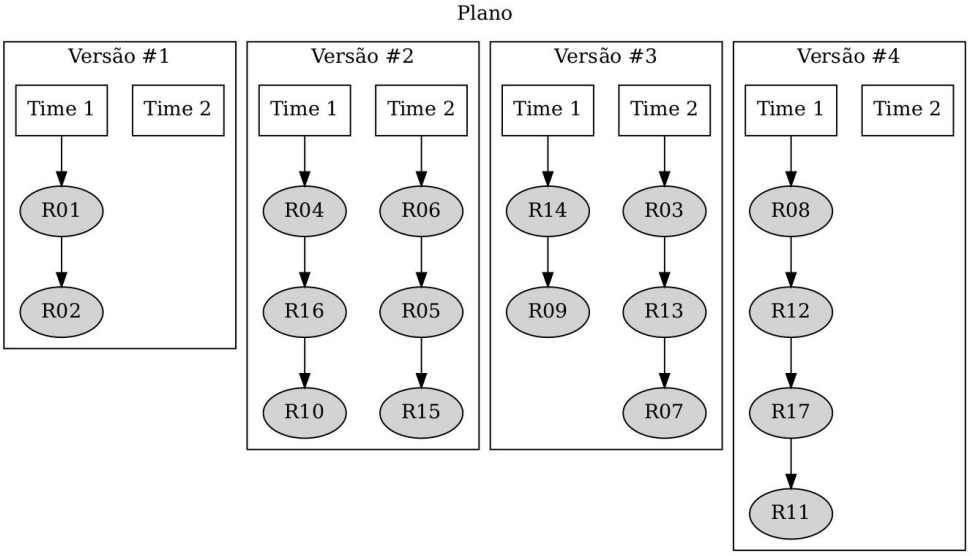

Figura 31. Resultado para o caso de estudo com duas equipes e opção de balanceamento desligada.

Assim como nos casos anteriores, todos os parâmetros foram tratados corretamente. Entretanto, apesar do tempo de desenvolvimento do projeto ter permanecido igual ao caso onde o balanceamento estava ligado é possível verificar que, neste caso, a quantidade de requisitos que o time 1 desenvolve é muito maior que do time 2 que, inclusive, fica com nenhum requisito atribuído a ele em duas versões diferentes.

4.4 Considerações finais do capítulo

Este capítulo apresentou a arquitetura do método proposto e as principais rotinas implementadas na ferramenta desenvolvida. As etapas de processamento foram discutidas citando elementos de entrada, rotinas de implementação e saídas desejadas, considerando um exemplo apresentado na Figura 24.

O capítulo também abordou o experimento realizado para validar o protótipo desenvolvido. A contextualização desses experimentos assim como os resultados obtidos foram discutidos na seção anterior.

CAPÍTULO 5

CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Este trabalho tem como objetivo propor uma ferramenta para auxiliar na solução automatizada do problema da próxima versão, o qual consiste em selecionar um subconjunto de requisitos de *software* para serem desenvolvidos e entregues no próximo incremento do produto.

No Capítulo 2 foram introduzidos os principais conceitos teóricos e definições técnicas usados como base para compreender o contexto onde a ferramenta proposta é aplicada. Foram abordados temas como o desenvolvimento incremental de *softwares*, o problema da próxima versão, conceitos de planejamento em IA, a linguagem PDDL e o uso de planejadores. O Capítulo 3 apresentou os métodos e técnicas utilizados para consolidar a ferramenta de modo que o planejamento em IA fosse aplicado de forma que a arquitetura final gerasse resultados satisfatórios. Por fim, o Capítulo 4 apresentou como foi feita a construção da ferramenta, como ficou sua arquitetura e trouxe informações sobre seu funcionamento. Neste capítulo também foram apresentados os resultados obtidos através da aplicação da ferramenta em alguns estudos de caso.

Como resultados, a ferramenta conseguiu gerar planos de forma automatizada de maneira satisfatória tanto nos casos de teste, apresentados no Capítulo 3, quanto quando foi usada para priorizar os requisitos de um projeto real, como mostrado no Capítulo 4. Em ambos os casos todas as dependências, prioridades, tamanho de versão e número de equipes disponibilizados que foram passados como entrada foram respeitados. Sendo assim, conclui-se que os objetivos específicos desta pesquisa foram alcançados e que a ferramenta, apesar de depender de fatores externos como o projeto em que está sendo aplicada e o nível de experiência profissional de quem está definindo seus parâmetros de entrada, ela pode contribuir para o planejamento das atividades e do cronograma dos projetos.

As principais limitações identificadas nesta pesquisa referem-se à subjetividade relacionada aos valores usados como prioridade e tamanho dos requisitos e, até mesmo, às dependências apresentadas. Apesar do planejador ser

objetivo na hora de gerar um plano, os valores que lhe são passados como entrada dependem de diversos fatores como, por exemplo, do projeto em questão ou da experiência de quem definirá esses valores. Essas e outras variáveis impactam diretamente na qualidade do resultado gerado pela ferramenta.

Além disso, também é possível citar como limitação que o balanceamento realizado pelo algoritmo da ferramenta para distribuir os requisitos nas equipes deveria ser feito considerando o tamanho de cada requisito e não a quantidade. Foram realizados testes durante a construção da ferramenta para que isso fosse tratado, mas a implementação foi inconclusiva. Diante disso, identifica-se como trabalho futuro a possibilidade de alterar o modelo para que ele passe a considerar o tamanho dos requisitos na hora de dividi-los entre as equipes.

Como trabalho futuro também sugere-se o desenvolvimento de uma interface gráfica que facilite a utilização da ferramenta tornando seu uso mais fácil e intuitivo.

Finalmente, conclui-se que, fundamentando-se no desenvolvimento feito pela equipe durante todas as etapas do projeto, os objetivos propostos foram alcançados e que a criação de planos para priorizar requisitos e a utilização da ferramenta desenvolvida pode auxiliar desenvolvedores na tarefa de priorização. Ainda que não forneça resultados exatos, os experimentos mostram que é possível alcançar resultados aproximados. Sendo assim, a utilização do método proposto em processos de *software* que contam com um grande conjunto de requisitos diminuiria a quantidade de tempo/esforço empregados nessas atividades.

REFERÊNCIAS

[1] ALMUTAIRI, Abeer & Qureshi, M. Rizwan. The Proposal of Scaling the Roles in Scrum of Scrums for Distributed Large Projects. Journal of Information Technology and Computer Science (IJITCS). 7. 68-74. 10.5815/ijitcs.2015.08.10. 2015.

[2] AMARAL, Aruan Galves Nascimento et al. Uma abordagem baseada em riscos de software para seleção de requisitos. 2017.

[3] BAGNALL, Anthony J.. ; RAYWARD-SMITH, Victor J.. ; WHITTLEY, Ian M. The next release problem. **Information and software technology**, v. 43, n. 14, p. 883-890, 2001.

[4] BENTON, J., Amanda Coles, and Andrew Coles. "Temporal planning with preferences and time-dependent continuous costs." Proceedings of the International Conference on Automated Planning and Scheduling. Vol. 22. No. 1. 2012.

[5] CANTONI, Luiz Fernando. Avaliação do Uso da Linguagem PDDL no Planejamento de Missões para Robôs Aéreos. 2010.

[6] CARVALHO, Bárbara & Dias, Daffyne. Ambrosia: Solução para Gerenciamento de Projetos. 2019.

[7] CASHMORE, Michael, et al. "A compilation of the full PDDL+ language into SMT." Proceedings of the International Conference on Automated Planning and Scheduling. Vol. 26. No. 1. 2016.

[8] COLES, Amanda, et al. "Forward-chaining partial-order planning." Proceedings of the International Conference on Automated Planning and Scheduling. Vol. 20. No. 1. 2010.

[9] COOPER, Clark, et al. EXPAT. Disponível em: <https://libexpat.github.io/>. Acesso em 13 jan. 2021.

[10] DENENBERG, Elad & Coles, Amanda. Automated Planning in Non-Linear Domains for Aerospace Applications. 2018.

[11] DEL SAGRADO, José; ÁGUILA, Isabel M.; ORELLANA, Francisco J. Requirements interaction in the next release problem. In: **Proceedings of the 13th annual conference companion on Genetic and evolutionary computation**. ACM, 2011. p. 241-242.

[12] GEFFNER, Hector; BONET, Blai. A concise introduction to models and methods for automated planning. **Synthesis Lectures on Artificial Intelligence and Machine Learning**, v. 8, n. 1, p. 1-141, 2013.

[13] GHALLAB, Malik & Knoblock, Craig & Wilkins, David & Barrett, Anthony & Christianson, Dave & Friedman, Marc & Kwok, Chung & Golden, Keith & Penberthy, Scott & Smith, David & Sun, Ying & Weld, Daniel. PDDL - The Planning Domain Definition Language. 1998.

[14] GRAPHVIZ. Disponível em: <https://graphviz.org/>. Acesso em 13 jan. 2021.

[15] HELMET, Malte. "The fast downward planning system." Journal of Artificial Intelligence Research 26 (2006): 191-246.

[16] HOFFMANN, Joerg. METRIC-FF: Top Performer in the Numeric Track of the 3rd International Planning Competition. Disponível em: <https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html>. Acesso em 02 jan. 2021.

[17] International Conference on Automated Planning and Scheduling (ICAPS). Disponível em: <https://www.icaps-conference.org/>. Acesso em 15 jan. 2021.

[18] KARP, Richard M. Reducibility among combinatorial problems. In: **Complexity of computer computations**. Springer, Boston, MA, 1972. p. 85-103.

[19] MACHADO, Leticia *et al*. Task allocation for crowdsourcing using AI planning. In: **Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering**. ACM, 2016. p. 36-40.

[20] MARELLA, Andrea. "What automated planning can do for business process management." International Conference on Business Process Management. Springer, Cham, 2017.

[21] MUJUMDAR, Ashwini & Masiwal, GAYATRI & CHAWAN, Pramila. (2012). **Analysis of various Software Process Models**. International Journal of Engineering Research and Applications (IJERA). 2. 2015-2021.

[22] OLARONKE, Iroju; RHODA, Ikono; ISHAYA, Gambo. An Appraisal of Software Requirement Prioritization Techniques. **Asian Journal of Research in Computer Science**, p. 1-16, 2018.

[23] PAASIVAARA, MARIA & LASSENIUS, Casper. Scaling Scrum in a Large Distributed Project. International Symposium on Empirical Software Engineering and Measurement. 363-367. 10.1109/ESEM.2011.49. 2011.

[24] PRESSMAN, Roger. Software Engineering: A Practitioner's Approach, 8th Edition, McGraw-Hill Publication. 2015.

[25] POLIVAEV, Dimitry, et al. FREEPLANE. Disponível em: <https://www.freeplane.org/wiki/index.php/Home>. Acesso em 14 jan. 2021.

[26] SOMMERVILLE, I. Software engineering. 5th. ed. Addison-Wesley, 1995.

APÊNDICE A

Estudo dos planejadores

Com intuito de decidir qual planejador seria usado no desenvolvimento da ferramenta relativa e este documento foram testados alguns planejadores (Tabela 2) entre os quais dois se sobressaíram: o Metric-FF e o SMTPlan. O primeiro oferece seis algoritmos de busca: subida de encosta seguido de busca “melhor primeiro”, “melhor primeiro” (com e sem poda), subida de encosta, A* e subida de encosta seguido por A*. Já o segundo, usa um modelo baseado em acontecimentos para gerar o plano, que consiste em uma maneira de capturar mudanças nos estados em um determinado instante.

Todos os testes foram realizados em uma máquina Intel core i5-9300H, 8GB de RAM, sistema operacional Ubuntu 18.04.3.LTS.

Planejador	**Compilação**	**Execução**	**Descrição**
Metric-FF	✓	✓	Aviso sobre não executar minimização em domínios com pré-condições numéricas.
SMTplan	✓	✓	Não executou minimização, apesar de não imprimir nenhum aviso.
Fast-downward	✓	✓	Não aceita o símbolo de menor ou igual (“<=”) na pré-condição.
OPTIC	X	X	Chamou uma função não definida
POPF	X	X	Chamou uma função não definida

Tabela 2: Planejadores testados. ✓ representa etapas compridas corretamente, X etapas não compridas.

A seguir são apresentados os resultados de três casos de teste conduzidos usando os dois planejadores cujas características e desempenho se sobressaíram aos demais:o Metric-FF e o SMTPlan.

Caso de teste 1

O primeiro caso de teste procurou verificar a situação na qual existem requisitos de alta prioridade dependendo de requisitos de baixa prioridade.

Considere o conjunto de requisitos R={A, B, C, D, E} com tamanhos 2, 1, 1, 1, 1 e prioridades 5, 1, 1, 3, 2, respectivamente. Sendo que o requisito C depende de B, e este, por sua vez, depende de A, conforme Figura 32. Acrescenta-se ainda que, o tamanho máximo de plano foi configurado para 4 (quatro). Os resultados são mostrados na Tabela 3.

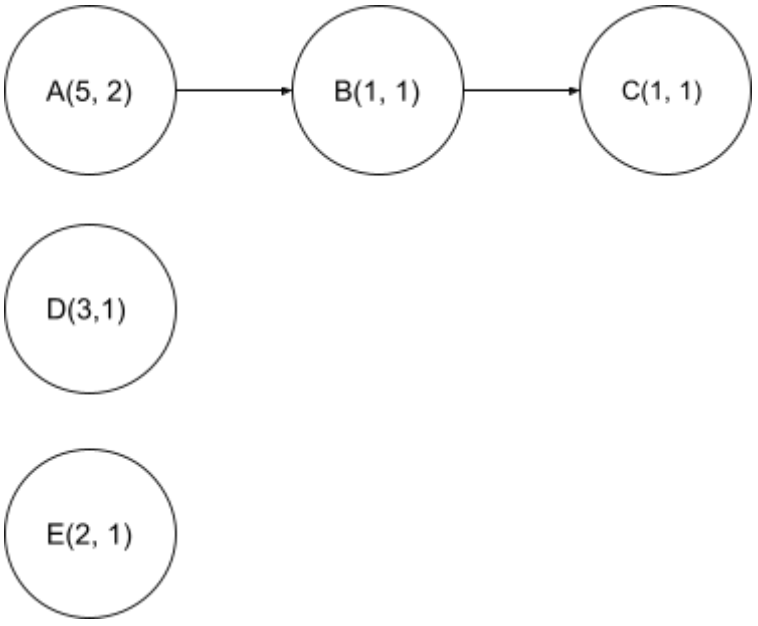

Figura 32: Caso de teste 1. Cada vértice representa um requisito. Uma aresta E(X,Y) denota dependência do requisito Y por X. Os vértices possuem o formato “nome(prioridade, tamanho)”.

Método	Planejador	Algoritmo	Resultado	Tempo de execução
Planejamento	Metric-FF	Subida de encosta seguido de busca “melhor primeiro”	A, E, D	0.004s
Planejamento	Metric-FF	Busca “melhor primeiro”	A, B, C	0.005s
Planejamento	Metric-FF	Melhor primeiro” com poda	D, E, A	0.005s
Planejamento	Metric-FF	A*	A, B, C	0.005s
Planejamento	Metric-FF	Subida de encosta	E, D, A	0.005s
Planejamento	Metric-FF	Subida de encosta seguido de A*	A, E, D	0.005s
Planejamento	SMTPlan	Padrão	E, D, A	0.005s

Tabela 3: Resultados do caso de teste 1.

A partir do problema exposto acima, o plano ideal para este caso é {A,B,C}. Os únicos algoritmos que alcançaram esse resultado foram o algoritmo de busca “melhor primeiro” e o A*, ambos do planejador Metric-FF. Os planos que utilizaram o algoritmo de subida de encosta não conseguiram minimizar a métrica de prioridade, assim como o planejador SMTPlan.

Caso de teste 2

O objetivo deste teste é verificar o caso no qual um requisito de baixa prioridade preenche grande parte do tamanho do subconjunto selecionado. Por exemplo, um subconjunto de tamanho dez pode ser formado por dez requisitos de prioridade 5 (cinco) e de tamanho 1 um ou por um requisito de prioridade 1 (um) e de tamanho dez.

Considere o conjunto de requisitos R={A, B, C, D, E, F, G} com tamanhos 2, 2, 3, 4, 2, 1, 7 e prioridades 1, 4, 4, 2, 3, 3, 5 respectivamente. Sendo que o requisito C depende de B, e este, por sua vez, depende de A, e o requisito F depende de D e E, conforme Figura 33. Acrescenta-se ainda que o tamanho máximo de plano foi configurado para 9 (nove). Os resultados são mostrados na Tabela 4.

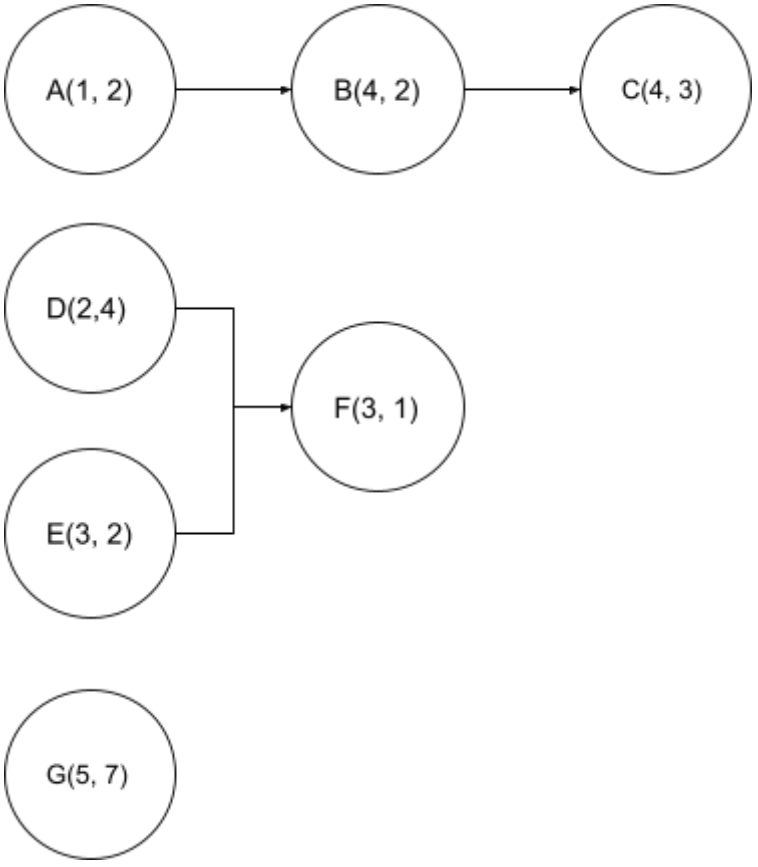

Figura 33: Caso de teste 2. Cada vértice representa um requisito. Uma aresta E(X,Y) denota dependência do requisito Y por X. Os vértices possuem o formato “nome(prioridade, tamanho)”.

Método	Planejador	Algoritmo	Resultado	Tempo de execução
Planejamento	Metric-FF	Subida de encosta seguido de busca “melhor primeiro”	G, E	0.004s
Planejamento	Metric-FF	Busca “melhor primeiro”	A, G	0.005s
Planejamento	Metric-FF	Melhor primeiro” com poda	A, G	0.005s
Planejamento	Metric-FF	A*	G, A	0.005s
Planejamento	Metric-FF	Subida de encosta	E, G	0.005s
Planejamento	Metric-FF	Subida de encosta seguido de A*	G, E	0.005s
Planejamento	SMTPlan	Padrão	G, E	0.005s

Tabela 4: Resultados do caso de teste 2.

Neste caso, o plano ideal é {A, D, E, F}, entretanto nenhum planejador conseguiu alcançar ele.

Caso de teste 3

Este caso é semelhante ao segundo, entretanto o tamanho do requisito G foi alterado para 1, conforme Figura 34, com o intuito de gerar um problema simples, sem induzir a um caso complexo. Os planos gerados são mostrados na Tabela 5.

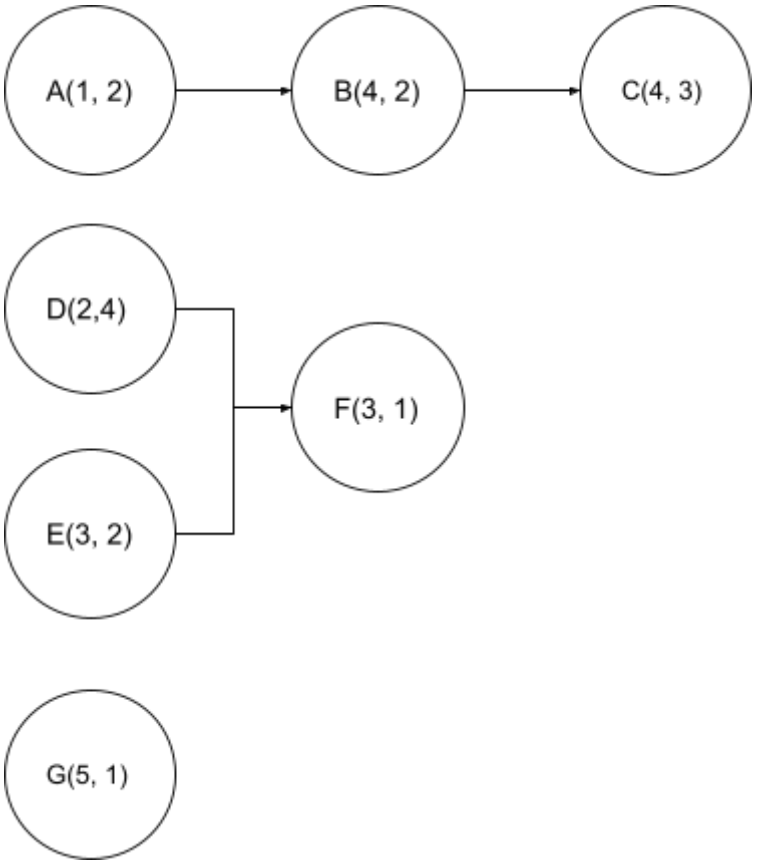

Figura 34: Caso de teste 33. Cada vértice representa um requisito. Uma aresta E(X,Y) denota dependência do requisito Y por X. Os vértices possuem o formato “nome(prioridade, tamanho)”.

Método	**Planejador**	**Algoritmo**	**Resultado**	**Tempo de execução**
Planejamento	Metric-FF	Subida de encosta seguido de busca “melhor primeiro”	D, E, A, G	0.005s
Planejamento	Metric-FF	Busca “melhor primeiro”	A, B, C, E	0.005s

Planejamento	Metric-FF	Melhor primeiro” com poda	G, A, B, D	0.005s
Planejamento	Metric-FF	A*	G, E, D, A	0.005s
Planejamento	Metric-FF	Subida de encosta	E, D, F, A	0.005s
Planejamento	Metric-FF	Subida de encosta seguido de A*	D, E, A, G	0.005s
Planejamento	SMTPlan	Padrão	G, E, D, A	0.005s

Tabela 5: Resultados do caso de teste 3.

Neste caso, o plano ideal ainda é {A, D, E, F}. Somente o algoritmo “subida de encosta”, do planejador Metric-FF, chegou ao resultado ótimo.

Com base nos resultados obtidos nos casos de teste, o planejador escolhido para ser utilizado no desenvolvimento da ferramenta relacionada a este documento foi o planejador Metric-FF, desenvolvido por Joerg Hoffmann.

APÊNDICE B

Arquivo de domínio em PDDL (domain.pddl) para o exemplo apresentado no Capítulo 4, ilustrado nas Figuras 24, 25 e 26.

```
(define (domain nrl)
      (:requirements :typing :fluents :negative-preconditions :disjunctive-preconditions)
      (:types requirement team priorities)

  (:predicates
            (done_all ?r - requirement)
            (done_team ?r - requirement ?t - team)
            (dependent ?r ?req - requirement)
            (r_priority ?r - requirement ?p - priorities)
            (priority ?p - priorities)
            (team ?t - team)
     )

     (:functions
            (max_capacity)
            (capacity ?t - team)
            (size ?r - requirement)
            (sum)
            (weight ?p - priorities)
     )

     (:action DEV_REQ
            :parameters (?r - requirement ?t - team ?p - priorities)
            :precondition (and
                  (priority ?p)
                  (team ?t)
                  (r_priority ?r ?p)
                  (>= (capacity ?t) (size ?r))
                  (forall (?team - team)
                        (not (done_team ?r ?team))
                  )
                  (forall (?req - requirement)
                        (imply (dependent ?r ?req) (or (done_team ?req ?t) (done_all ?req)))
                  )
```

```
35.           )
36.           :effect (and
37.                 (done_team ?r ?t)
38.                 (decrease (capacity ?t) (size ?r))
39.                 (increase (sum) (weight ?p))
40.                 (not (priority ?p))
41.                 (priority P1)
42.                 (when (and (team team1)) (and (not (team
   team1)) (team team2)))
43.                 (when (and (team team2)) (and (not (team
   team2)) (team team1)))
44.           )
45.     )
46.     (:action NEW_SPRINT
47.           :precondition (and
48.                 (forall (?t - team)
49.                       (= (capacity ?t) 0)
50.                 )
51.           )
52.           :effect (and
53.                 (forall (?r - requirement ?t - team)
54.                       (when (and (done_team ?r ?t)) (and
   (done_all ?r)))
55.                 )
56.                 (forall (?t - team)
57.                       (and (assign (capacity ?t)
   (max_capacity))
58.                             (not (team ?t))
59.                       )
60.                 )
61.                 (team team1)
62.                 (forall (?p - priorities)
63.                       (not (priority ?p))
64.                 )
65.                 (priority P1)
66.           )
67.     )
68.     (:action INCREASE_PRIORITY
69.           :effect (and
70.                 (when (and (priority P1)) (and (not (priority
   P1)) (priority P2)))
71.                 (when (and (priority P2)) (and (not (priority
   P2)) (priority P3)))
```

```
                (when (and (priority P3)) (and (not (priority P3)) (priority P4)))
                (when (and (priority P4)) (and (not (priority P4)) (priority P5)))
                (when (and (priority P5)) (and (not (priority P5)) (priority P0)))
            )
     )
     (:action CHANGE_TEAM
            :effect (and
                (when (and (team team1)) (and (not (team team1)) (team team2)))
                (when (and (team team2)) (and (not (team team2)) (team team1)))
                (forall (?p - priorities)
                      (not (priority ?p))
                )
                (priority P1)
            )
     )
     (:action RESET_CAPACITY
            :parameters (?t - team)
            :precondition (and (priority P0) (team ?t))
            :effect (and (assign (capacity ?t) 0)
                (when (and (team team1)) (and (not (team team1)) (team team2)))
                (when (and (team team2)) (and (not (team team2)) (team team1)))
                (not (priority P0))
                (priority P1)
            )
     )
  )
```

Arquivo de problema em PDDL (problem.pddl) para o exemplo apresentado no Capítulo 4, ilustrado nas Figuras 24, 25 e 26.

```
(define (problem nrl)
    (:domain nrl)
    (:objects
        A C G D E B F - requirement
        team1 team2 - team
        P0 P1 P2 P3 P4 P5 - priorities
    )

    (:init
            (= (sum) 0)

            (= (max_capacity) 12)
            (= (capacity team1) 12)
            (= (capacity team2) 12)

            (priority P1)

            (team team1)

            (dependent C A)
            (dependent C B)
            (dependent G A)
            (dependent G B)
            (dependent G C)
            (dependent E C)
            (dependent E D)
            (dependent B A)
            (dependent F A)
            (dependent F B)
            (dependent F C)
            (dependent F D)
            (dependent F E)

            (= (size A) 3)
            (= (size C) 6)
            (= (size G) 9)
            (= (size D) 4)
            (= (size E) 5)
            (= (size B) 3)
```

```
40.          (= (size F) 5)
41.
42.
43.          (r_priority A P1)
44.          (r_priority C P2)
45.          (r_priority G P2)
46.          (r_priority D P3)
47.          (r_priority E P3)
48.          (r_priority B P4)
49.          (r_priority F P5)
50.
51.          (= (weight P5) 1)
52.          (= (weight P4) 10)
53.          (= (weight P3) 100)
54.          (= (weight P2) 1000)
55.          (= (weight P1) 10000)
56.      )
57.
58.     (:goal
59.          (and (= (sum) 12211))
60.      )
61.  )
```

Arquivo do plano gerado pela ferramenta Metric-FF, para o exemplo apresentado no Capítulo 4.

```
1. ff: parsing domain file
2. domain 'NRL' defined
3.  ... done.
4. ff: parsing problem file
5. problem 'NRL' defined
6.  ... done.
7.
8.
9. warning: empty con/disjunction in domain definition. simplifying.
10.
11.
12.   translating negated cond for predicate DONE_TEAM
13.   no metric specified.
14.
15.   task contains conditional effects. turning off state
   domination.
16.
17.
18.
19.   ff: search configuration is Enforced Hill-Climbing, if that
   fails then best-first search.
20.   Metric is plan length.
21.   NO COST MINIMIZATION (and no cost-minimizing relaxed plans).
22.
23.   Cueing down from goal distance:    9 into depth [1]
24.                                      8            [1]
25.                                      7            [1]
26.                                      6            [1]
27.                                      5            [1]
28.                                      4            [1]
29.                                      3            [1][2]
30.                                      2            [1]
31.                                      1            [1]
32.                                      0
33.
34.   ff: found legal plan as follows
35.   step    0: DEV_REQ A TEAM1 P1
36.           1: DEV_REQ B TEAM1 P1
37.           2: DEV_REQ C TEAM1 P1
```

```
          3: DEV_REQ D TEAM1 P1
          4: DEV_REQ E TEAM1 P1
          5: DEV_REQ G TEAM1 P1
          6: DEV_REQ H TEAM1 P1
          7: NEW_SPRINT
          8: DEV_REQ F TEAM1 P1
          9: DEV_REQ I TEAM1 P1

  time spent:      0.00 seconds instantiating 12 easy, 0 hard action templates
                   0.00 seconds reachability analysis, yielding 33 facts and 12 actions
                    0.00 seconds creating final representation with 33 relevant facts, 4 relevant fluents
                 0.00 seconds computing LNF
                 0.00 seconds building connectivity graph
                   0.00 seconds searching, evaluating 13 states, to a max depth of 2
                 0.00 seconds total time


```

APÊNDICE C

Arquivo PDDL de domínio (domain.pddl) do Ambrosia

```
1.  (define (domain nrl)
2.        (:requirements :typing :fluents :negative-preconditions
    :disjunctive-preconditions)
3.        (:types requirement team priorities)
4.
5.        (:predicates
6.              (done_all ?r - requirement)
7.              (done_team ?r - requirement ?t - team)
8.              (dependent ?r ?req - requirement)
9.              (r_priority ?r - requirement ?p - priorities)
10.             (priority ?p - priorities)
11.             (team ?t - team)
12.       )
13.
14.       (:functions
15.             (max_capacity)
16.             (capacity ?t - team)
17.             (size ?r - requirement)
18.             (sum)
19.             (weight ?p - priorities)
20.       )
21.
22.       (:action DEV_REQ
23.             :parameters (?r - requirement ?t - team ?p - priorities)
24.             :precondition (and
25.                   (priority ?p)
26.                   (team ?t)
27.                   (r_priority ?r ?p)
28.                   (>= (capacity ?t) (size ?r))
29.                   (forall (?team - team)
30.                         (not (done_team ?r ?team))
31.                   )
32.                   (forall (?req  - requirement)
33.                         (imply (dependent ?r ?req) (or (done_team
    ?req ?t) (done_all ?req)))
34.                   )
35.             )
36.             :effect (and
37.                   (done_team ?r ?t)
38.                   (decrease (capacity ?t) (size ?r))
39.                   (increase (sum) (weight ?p))
40.                   (not (priority ?p))
41.                   (priority P1)
42.                   (when (and (team team1)) (and (not (team team1))
    (team team2)))
43.                   (when (and (team team2)) (and (not (team team2))
    (team team1)))
44.             )
45.       )
```

```
46.       (:action NEW_SPRINT
47.             :parameters ()
48.             :precondition (and
49.                   (forall (?t - team)
50.                         (= (capacity ?t) 0)
51.                   )
52.             )
53.             :effect (and
54.                   (forall (?r - requirement ?t - team)
55.                         (when (and (done_team ?r ?t)) (and
   (done_all ?r)))
56.                   )
57.                   (forall (?t - team)
58.                         (and (assign (capacity ?t) (max_capacity))
59.                               (not (team ?t))
60.                         )
61.                   )
62.                   (team team1)
63.                   (forall (?p - priorities)
64.                         (not (priority ?p))
65.                   )
66.                   (priority P1)
67.             )
68.       )
69.       (:action INCREASE_PRIORITY
70.             :parameters ()
71.             :precondition (and )
72.             :effect (and
73.                   (when (and (priority P1)) (and (not (priority P1))
   (priority P2)))
74.                   (when (and (priority P2)) (and (not (priority P2))
   (priority P3)))
75.                   (when (and (priority P3)) (and (not (priority P3))
   (priority P4)))
76.                   (when (and (priority P4)) (and (not (priority P4))
   (priority P5)))
77.                   (when (and (priority P5)) (and (not (priority P5))
   (priority P6)))
78.                   (when (and (priority P6)) (and (not (priority P6))
   (priority P0)))
79.             )
80.       )
81.       (:action CHANGE_TEAM
82.             :parameters ()
83.             :precondition (and )
84.             :effect (and
85.                   (when (and (team team1)) (and (not (team team1))
   (team team2)))
86.                   (when (and (team team2)) (and (not (team team2))
   (team team1)))
87.                   (forall (?p - priorities)
88.                         (not (priority ?p))
89.                   )
90.                   (priority P1)
91.             )
```

```
92.       )
93.       (:action RESET_CAPACITY
94.             :parameters (?t - team)
95.             :precondition (and (priority P0)
96.                   (team ?t)
97.             )
98.             :effect (and (assign (capacity ?t) 0)
99.                   (when (and (team team1)) (and (not (team team1))
   (team team2)))
100.                  (when (and (team team2)) (and (not (team team2))
   (team team1)))
101.                  (not (priority P0))
102.                  (priority P1)
103.            )
104.      )
105.  )
106.
107.
```

Arquivo PDDL de problema (problem.pddl) do Ambrosia

```
(define (problem nrl)
    (:domain nrl)
    (:objects
        R01 R02 R04 R09 R14 R06 R10 R03 R07 R08 R13 R12 R16 R17 R05
  R11 R15 - requirement
        team2 team1 - team
        P0 P1 P2 P3 P4 P5 P6 - priorities
    )

    (:init
        (= (sum) 0)

        (= (max_capacity) 10)
        (= (capacity team1) 10)
        (= (capacity team2) 10)

        (priority P1)

        (team team1)

        (dependent R02 R01)
        (dependent R04 R01)
        (dependent R09 R04)
        (dependent R14 R04)
        (dependent R06 R02)
        (dependent R10 R02)
        (dependent R03 R01)
        (dependent R07 R03)
        (dependent R08 R03)
        (dependent R13 R03)
        (dependent R12 R16)
        (dependent R16 R01)
        (dependent R17 R16)
        (dependent R05 R01)
        (dependent R11 R05)
        (dependent R15 R05)

        (= (size R01) 5)
        (= (size R02) 4)
        (= (size R04) 4)
        (= (size R09) 2)
        (= (size R14) 3)
        (= (size R06) 3)
        (= (size R10) 2)
        (= (size R03) 4)
        (= (size R07) 3)
        (= (size R08) 2)
        (= (size R13) 3)
        (= (size R12) 3)
        (= (size R16) 4)
        (= (size R17) 3)
        (= (size R05) 4)
```

```
52.         (= (size R11) 2)
53.         (= (size R15) 3)
54.
55.         (r_priority R01 P1)
56.         (r_priority R02 P2)
57.         (r_priority R04 P2)
58.         (r_priority R09 P2)
59.         (r_priority R14 P2)
60.         (r_priority R06 P3)
61.         (r_priority R10 P3)
62.         (r_priority R03 P4)
63.         (r_priority R07 P4)
64.         (r_priority R08 P4)
65.         (r_priority R13 P4)
66.         (r_priority R12 P5)
67.         (r_priority R16 P5)
68.         (r_priority R17 P5)
69.         (r_priority R05 P6)
70.         (r_priority R11 P6)
71.         (r_priority R15 P6)
72.
73.         (= (weight P6) 1)
74.         (= (weight P5) 10)
75.         (= (weight P4) 100)
76.         (= (weight P3) 1000)
77.         (= (weight P2) 10000)
78.         (= (weight P1) 100000)
79.
80.     )
81.     (:goal
82.         (and (= (sum) 142433))
83.     )
84.)
85.
```