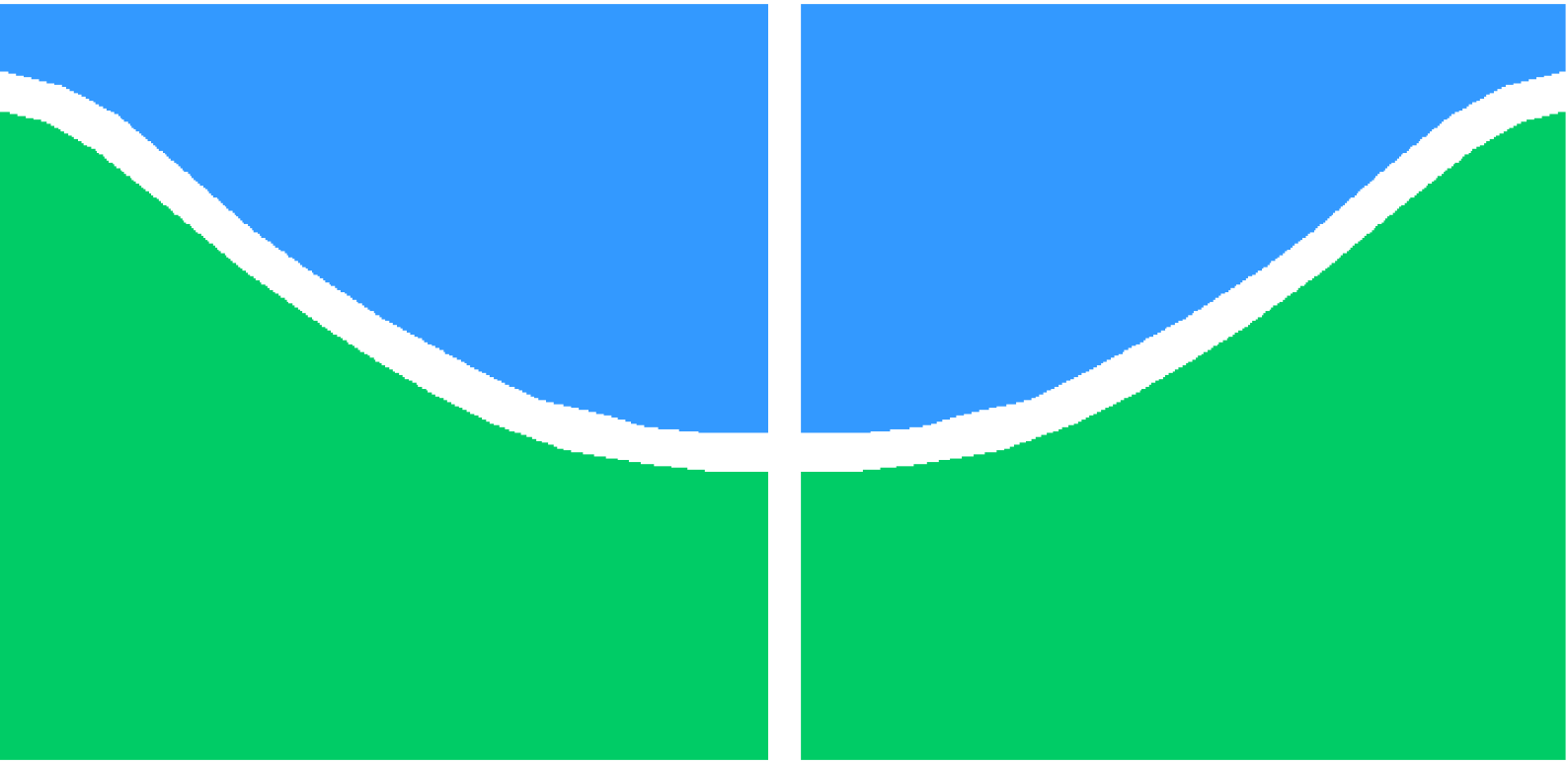


Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Resolvendo Puzznic através de planejamento automatizado

Autor: Samuel de Souza Buters Pereira

Orientador: Bruno César Ribas

Brasília, DF

2023

Samuel de Souza Buters Pereira

Resolvendo Puzznic através de planejamento automatizado

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Bruno César Ribas

Brasília, DF

2023

Samuel de Souza Buters Pereira
Resolvendo Puzznic através de planejamento automatizado/ Samuel de Souza Buters Pereira. – Brasília, DF, 2023-
62 p. : il. (algumas color.) ; 30 cm.

Orientador: Bruno César Ribas

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2023.

1. Planejamento automatizado. 2. jogo eletrônico. I. Bruno César Ribas. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Resolvendo Puzznic através de planejamento automatizado

CDU 02:141:005.6

Resumo

O planejamento automatizado consiste num ramo da inteligência artificial que busca estudar os modelos computacionais e métodos de criação, análise, administração e execução de planos. Essa área da inteligência artificial trata de problemas que possuem um estado inicial que desejamos transformar em um estado final através de um conjunto de ações. Este trabalho apresenta a modelagem parcial do jogo eletrônico Puzznic como um problema de planejamento automatizado, utilizando o PDDL como linguagem de descrição de problemas, bem como o planejador *ForwardPlanner* do pacote *SymbolicPlanners.jl* para a linguagem de programação Julia. Os experimentos realizados demonstram que os modelos desenvolvidos representam apropriadamente o Puzznic dentro do escopo planejado.

Palavras-chaves: planejamento automatizado. pddl. puzznic. jogo eletrônico.

Abstract

AI planning consists of a branch of artificial intelligence that seeks to study the computational models and methods to create, analyse, manage and execute plans. This area of artificial intelligence deals with problems that have an initial state that we wish to transform into a goal state through a set of actions. This thesis presents the partial modeling of the video game Puzznic as a AI planning problem, utilizing PDDL as the problem description language, as well as using the ForwardPlanner planner from the SymbolicPlanners.jl package for the programming language Julia. The experiments show that the developed models properly represent the planned scope for the Puzznic domain.

Key-words: ai planning. pddl. puzznic . video game

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A criação de planos é considerada um sinal de inteligência, sendo, portanto, uma área de estudo da inteligência artificial. Problemas que envolvem planejamento possuem algumas características em comum, como o fato de possuírem um estado inicial, um estado final que desejamos alcançar e um conjunto de ações que podem ser tomadas para que o estado final possa ser alcançado a partir do estado inicial (HASLUM et al., 2019). Esse tipo de problema é abordado e estudado no ramo da inteligência artificial chamado de planejamento automatizado.

Alguns jogos eletrônicos podem ser vistos como problemas de planejamento e, portanto, modelados e resolvidos através de técnicas e ferramentas de planejamento automatizado. Isso é interessante visto que aumenta o catálogo de problemas resolvidos através do planejamento automatizado, possivelmente atraindo a realização de outras modelagens semelhantes na área, as quais podem ser utilizadas na área de jogos para, por exemplo, testar a resolubilidade de mapas gerados proceduralmente. O jogo eletrônico Plotting, desenvolvido pela Taito (ESPASA; MIGUEL; VILLARET, 2022), é um exemplo abordado por Espasa, Miguel e Villaret (2022), consistindo em um jogo de quebra-cabeças de combinação de peças em que o jogador deve transformar o estado de mapas que possuem várias peças de diversos tipos em um estado em que só exista uma peça de cada tipo, através do arremesso de peças em direção a outras semelhantes.

Este trabalho tem como objetivo a modelagem do jogo eletrônico Puzznic, também desenvolvido pela Taito (TAITO, 1990), como um problema de planejamento automatizado, consistindo em um jogo de combinação de peças, onde o jogador deve combinar peças similares com o intuito de eliminar todas as peças presentes num determinado quebra-cabeça. Além disso, tem-se como objetivos específicos a criação de um *script* para auxiliar na descrição de problemas e a validação do modelo através do seu uso em um planejador, bem como mediante a criação de uma ferramenta para visualizar as soluções encontradas.

O texto está dividido em cinco capítulos. O Capítulo 2 explica uma série de conceitos básicos para a compreensão do modelo a ser apresentado no capítulo quatro, começando com aspectos gerais a respeito do planejamento automatizado e então adentrando em conceitos específicos acerca da *Planning Domain Definition Language*, consistindo na linguagem de descrição de problemas proposta a ser utilizada. O Capítulo 3 especifica os objetivos e regras do Puzznic, definindo o domínio do problema que será trabalhado. O Capítulo 4 apresenta o modelo desenvolvido, começando pelo arquivo de domínio, depois passando por todas as etapas envolvendo a geração de arquivos de problema e por fim

abordando a visualização das soluções. Por último, no Capítulo 5, são apresentados os experimentos realizados com o modelo.

2 Referencial teórico

Este capítulo está dividido em três partes. A primeira parte apresenta o conceito geral do planejamento automatizado. A segunda parte apresenta brevemente o conceito de linguagens de descrição de problemas, adentrando em detalhes acerca da *Planning Domain Definition Language*, sendo apresentadas as estruturas dos dois principais arquivos utilizados nessa linguagem. A terceira parte complementa o conceito de planejadores, o qual é brevemente mencionado em partes anteriores, seguindo com a apresentação do planejador Madagascar, mostrando um exemplo de entrada e saída do planejador.

2.1 Planejamento automatizado

O planejamento automatizado, conhecido também como *AI planning*, é um ramo da inteligência artificial que consiste no estudo de modelos computacionais e métodos de criação, análise, administração e execução de planos (HASLUM et al., 2019).

Tais planos geralmente consistem em problemas de transformação de estado, os quais possuem um estado inicial, um estado final desejado e um conjunto de ações que podem ser tomadas para mudar o estado. O plano, portanto, consiste em um conjunto de ações que podem ser tomadas a partir do estado inicial para se chegar ao estado final desejado. Esse conjunto de ações pode ser uma simples sequência de ações ou um cronograma onde ações são executadas em tempos determinados (HASLUM et al., 2019).

Um quebra-cabeça pode ser visto como um exemplo de problema de planejamento automatizado, onde as peças embaralhadas representam o estado inicial, as peças posicionadas de maneira que formem a imagem presente na caixa representam o estado final desejado e a ação de encaixar uma peça em outra consiste no conjunto de ações.

2.2 Linguagens de descrição de problemas

O planejamento automatizado, de um ponto de vista prático, insere uma camada intermediária entre o problema a ser resolvido e as técnicas de inteligência artificial utilizadas para resolver o problema: as linguagens de descrição de problemas (HASLUM et al., 2019).

2.2.1 *Planning Domain Definition Language*

Embora não seja a única, a linguagem de descrição de problemas mais difundida é a Planning Domain Definition Language, abreviada como PDDL (HASLUM et al.,

2019). Isso se dá em razão do PDDL ser a principal linguagem utilizada nas categorias da Competição Internacional de Planejamento, do inglês International Planning Competition (IPC), sendo a linguagem de descrição de problemas mais suportada por planejadores.

Desde a sua criação em 1998, diversas versões do PDDL surgiram. As versões subsequentes expandiram a linguagem com novas funcionalidades capazes de expressar tipos de problemas de planejamento diferentes, as quais, com o decorrer do tempo, foram removidas quando não conseguiram ganhar popularidade. As versões do PDDL publicadas até então são: PDDL 1; PDDL 1.2; PDDL 2.1; PDDL+; PDDL 2.2; PDDL 3.0; e PDDL 3.1 (HASLUM et al., 2019).

Os problemas descritos em PDDL são separados em duas partes: o arquivo de domínio e o arquivo de problema.

2.2.2 Arquivo de domínio

O arquivo de domínio busca definir o domínio do problema, envolvendo as definições das "físicas" de um dado problema, ou seja, definir quais são os predicados que existem nele, bem como as possíveis ações, a estrutura de ações compostas e os efeitos de tais ações (GHALLAB et al., 1998) (HASLUM et al., 2019).

Para explorar a estrutura básica de um arquivo de domínio, como um domínio pode ser modelado e a sintaxe do PDDL, o Código 2.1 apresenta um exemplo de modelagem do domínio do mundo dos blocos extraído da Conferência sobre Sistemas de Planejamento de Inteligência Artificial de 2000 (AIPS 2000)[1].

O mundo dos blocos consiste em um micromundo – um problema de domínio limitado que precisa de inteligência para ser resolvido – onde um conjunto de blocos são colocados sobre uma mesa e devem ser arranjados de uma forma específica desejada utilizando um braço robô que consegue pegar um bloco por vez (RUSSEL; NORVIG, 2010).

Observa-se no código que as expressões no PDDL ficam entre parênteses, além de utilizarem notação de prefixo, onde operadores e funções precedem os seus argumentos, semelhante à sintaxe da linguagem LISP.

O argumento `domain` consiste no nome do domínio. Os nomes de domínios não são *case-sensitive* e devem começar com uma letra e podem conter letras, dígitos, hifens e subtraços (GHALLAB et al., 1998). No Código 2.1, na linha 1, o nome é definido como `blocks`.

O argumento `:requirements` indica quais funcionalidades do PDDL o domínio usa e, por consequência, define o tipo de problema (HASLUM et al., 2019). Foi antecipado

[1] https://www.cs.colostate.edu/meps/repository/aips2000.html

Código 2.1 – Exemplo de definição do domínio do mundo de blocos

```
(define (domain blocks)
    (:requirements :strips)
    (:predicates (on ?x ?y)
        (ontable ?x)
        (clear ?x)
        (handempty)
        (holding ?x))

    (:action pick-up
        :parameters (?x)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect
            (and (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)))

    (:action put-down
        :parameters (?x)
        :precondition (holding ?x)
        :effect
            (and (not (holding ?x))
            (clear ?x)
            (handempty)
            (ontable ?x)))

    (:action stack
        :parameters (?x ?y)
        :precondition (and (holding ?x) (clear ?y))
        :effect
            (and (not (holding ?x))
            (not (clear ?y))
            (clear ?x)
            (handempty)
            (on ?x ?y)))

    (:action unstack
        :parameters (?x ?y)
        :precondition (and (on ?x ?y)
            (clear ?x)
            (handempty)
        )
        :effect
            (and (holding ?x)
            (clear ?y)
            (not (clear ?x))
            (not (handempty))
            (not (on ?x ?y)))))
```

Fonte: AIPS 2000

que apenas alguns planejadores – programas que resolvem problemas descritos em uma linguagem de descrição de problema – seriam capazes de lidar com todas as funcionalidades do PDDL e, portanto, a linguagem foi separada em um conjunto de requisitos (GHALLAB

et al., 1998). Dessa forma, o argumento `:requirements` formaliza o fato de que nem todos os planejadores conseguem resolver todos os problemas que podem ser descritos em PDDL, possibilitando ao planejador determinar rapidamente se ele será capaz ou não de resolver o problema modelado (GHALLAB et al., 1998). Caso nenhum requisito seja especificado, o domínio terá o requisito `:strips` por padrão (GHALLAB et al., 1998), sendo o tipo de domínio mais simples (HASLUM et al., 2019). O requisito `:strips` permite o uso de efeitos de adição e remoção básicos como especificado no resolvedor de problemas *Stanford Research Institute Problem Solver*, conhecido com STRIPS (FIKES; NILSSON, 1971). No Código 2.1, na linha 2, somente o `:strips` é definido como requisito.

O argumento `:predicates` declara os predicados do domínio, consistindo em variáveis binárias que representam fatos que são verdadeiros ou falsos (GHALLAB et al., 1998) (HASLUM et al., 2019). No domínio do Código 2.1, da linha 3 à 7, os predicados são: `on`, correspondendo se um bloco `x` está em cima de um bloco `y` ou não; `ontable`, correspondendo se um dado bloco está sobre a mesa ou não; `clear`, correspondendo se um bloco possui ou não outro bloco sobre ele; `handempty`, correspondendo se o braço robô está segurando algo ou não; e `holding`, correspondendo se um bloco está sendo segurado pelo braço robô.

O argumento `:action` define transições entre estados (HASLUM et al., 2019), sendo tipicamente composto por um conjunto de argumentos, um conjunto de pré-condições e um conjunto de efeitos. O argumento `:parameters` define os objetos que serão utilizados nas pré-condições ou efeitos da ação. O argumento de `:precondition` é opcional e define as condições necessárias para que uma ação seja aplicada, sendo que, quando não especificado, a ação é sempre executável. Já o argumento `:effect` descreve os efeitos de uma ação, ou seja, o que acontece quando a ação é aplicada (GHALLAB et al., 1998) (HASLUM et al., 2019). No domínio do Código 2.1, da linha 9 à 48, as ações são: `pick-up`, correspondendo a ação do braço robô pegar um bloco sobre a mesa; `put-down`, correspondendo a ação do braço robô colocar um bloco sobre a mesa; `stack`, correspondendo a ação do braço robô empilhar um bloco `x` sobre um bloco `y`; e `unstack`, correspondendo a ação do braço robô desempilhar um bloco `x` sobre um bloco `y`.

Outro argumento comum, embora não presente no exemplo apresentado, é o `:types`. O argumento `:types` é usado para declarar uma lista de tipos (GHALLAB et al., 1998). Esses tipos se assemelham à classes em linguagens orientadas a objetos. Todo tipo é um subtipo de um outro tipo, o qual é especificado após um hífen depois de uma lista de tipos, sendo definido como subtipo do tipo `object` quando não especificado (GHALLAB et al., 1998). No trecho de Código 2.2, os tipos `person` e `boat` são do tipo `object`, enquanto os tipos `cannibal` e `missionary` são do tipo `person`, o qual foi especificado anteriormente.

Outros argumentos existem e podem ser utilizados na definição do domínio de um problema com o PDDL. Esses argumentos e diversas outras informações podem ser

Código 2.2 – Exemplo de tipos

```
(:types
    person boat - object
    cannibal missionary - person
)
```

Fonte: De autoria própria

encontradas na literatura utilizada como referência de Haslum et al. (2019) e Ghallab et al. (1998), bem como na wiki de planejamento automatizado e PDDL, disponível em planning.wiki.[2].

2.2.3 Arquivo de problema

O arquivo de problema busca especificar o estado inicial e o estado final desejado. Dessa forma, seguindo o exemplo do Código 2.1, o Código 2.3 exemplifica um arquivo de problema para o domínio discutido anteriormente.

Código 2.3 – Exemplo de definição de um problema em PDDL

```
1 (define (problem blocks-4-0)
2     (:domain blocks)
3     (:objects D B A C )
4     (:init (clear C) (clear A) (clear B)
5         (clear D) (ontable C) (ontable A)
6         (ontable B) (ontable D) (handempty))
7     (:goal (and (on D C) (on C B) (on B A))))
```

Fonte: AIPS 2000

Semelhante ao domínio, o problema também possui um nome, sendo observável no argumento `problem`. Além disso, o problema também possui uma referência ao nome do domínio, no argumento `:domain`, o que acaba por ser redundante logo que a maioria dos planejadores recebem ambos os arquivos de domínio e de problema como entrada (HASLUM et al., 2019). No Código 2.3, o nome é definido como `blocks-4-0` na linha 1.

O argumento `:objects` lista todos os objetos presentes na instância do problema (HASLUM et al., 2019). No Código 2.3, na linha 3, os objetos são `A`, `B`, `C` e `D`. Caso o domínio possuísse tipos, cada objeto também teria seu tipo especificado, opcionalmente caso seja do tipo padrão `object`.

O argumento `:init` define o estado inicial do problema, através da listagem de todos os fatos que são verdadeiros, ou seja, os predicados (HASLUM et al., 2019). Dessa forma, todos os predicados não listados são considerados como falsos. No Código 2.3, da linha 4 à 6, todo os blocos são definidos como estando sobre a mesa através do predicado `ontable`, além de não possuírem nenhum outro bloco sobre eles através do predicado

[2] https://planning.wiki

`clear`, enquanto é definido que o braço robô não está segurando bloco algum através do predicado `handempty`.

O argumento `:goal`, em sua forma mais simples, especifica as condições que devem ser satisfeitas ao fim de um plano válido, possuindo uma notação semelhante à pré-condição de uma ação (HASLUM et al., 2019). No Código 2.3, na linha 7, o estado final é definido como o bloco `B` sobre o `A`, o bloco `C` sobre o `B`, o bloco `C` sobre o `B` e o bloco `D` sobre o `C`.

2.3 Planejadores

Um sistema de planejamento automatizado, normalmente referido como planejador, é o responsável por receber o modelo de um problema, consistindo dos arquivos de domínio e de problema no PDDL, e encontrar uma solução. Para encontrar tais soluções, os planejadores utilizam técnicas de resolução de problemas, como buscas heurísticas e algoritmos de resolução de problemas de satisfatibilidade booleana (HASLUM et al., 2019).

Os planejadores não possuem conhecimento acerca da descrição formal do problema, podendo ser aplicados a qualquer problema que possa ser descrito na linguagem utilizada para a modelagem do problema, desde que esteja dentro das restrições específicas do planejador. Essa característica é chamada de independência de domínio (HASLUM et al., 2019).

Na escolha de um planejador, as restrições específicas do planejador, mencionadas anteriormente, devem ser levadas em conta. Tais restrições ocorrem por conta da maioria dos planejadores suportarem somente um subconjunto do PDDL (GHALLAB et al., 1998) (HASLUM et al., 2019). Dessa forma, a reformulação de um modelo para que se encaixe nas limitações de um determinado planejador é uma preocupação a ser considerada (HASLUM et al., 2019).

A busca por planejadores pode ser feita através dos arquivos das Competições Internacionais de Planejamento[3], organizadas pela Conferência Internacional sobre Planejamento Automatizado e Agendamento, abreviado em inglês como ICAPS, os quais mantém arquivados os planejadores participantes bem como os seus códigos-fonte. A planning.wiki[4] também possui um catálogo de planejadores e, portanto, também pode ser usada para auxiliar a atividade de busca.

Com o intuito de exemplificar e demonstrar o uso de um planejador, o planejador *ForwardPlanner* do pacote *SymbolicPlanners.jl*, o qual foi utilizado nos experimentos desse trabalho, será apresentado.

[3] https://www.icaps-conference.org/competitions/
[4] https://planning.wiki

2.3.0.1 *SymbolicPlanners.jl*

O *SymbolicPlanners.jl* consiste num pacote para a linguagem de programação Julia que contém um conjunto de planejadores que operam utilizando a interface providenciada pelo pacote *PDDL.jl*, o qual compreende um *parser*, um interpretador e um compilador (ZHI-XUAN, 2022). O código-fonte de ambos os pacotes se encontram disponíveis no Github[5,6].

Com os pacotes instalados, os arquivos referentes aos exemplos do Código 2.1 e do Código 2.3 foram utilizados como entrada no *script* do Código 2.4, o qual utiliza o *ForwardPlanner* para buscar a solução do problema.

Código 2.4 – Exemplo de utilização do *ForwardPlanner*

```
1 using PDDL, SymbolicPlanners
2
3 domain = load_domain("domain.pddl")
4 problem = load_problem("probBLOCKS-4-0.pddl")
5 state = initstate(domain, problem)
6 spec = MinStepsGoal(problem)
7
8 planner = ForwardPlanner()
9 sol = planner(domain, state, spec)
```

Fonte: De autoria própria

A variável `spec` presente na linha 6 consiste em uma especificação de problema utilizada pelo planejador para determinar que a solução deve possuir o menor número de passos.

Dessa forma, com a execução do *script* do Código 2.4, o plano encontrado pode ser encontrado na variável `sol`, mais especificamente em `sol.plan`. O plano encontrado segue como:

```
(pick-up b)
(stack b a)
(pick-up c)
(stack c b)
(pick-up d)
(stack d c)
```

2.4 Considerações

Para o entendimento do modelo que será apresentada no capítulo quatro, tem-se como necessária a compreensão dos conceitos básicos acerca do planejamento automati-

[5] https://github.com/JuliaPlanners/SymbolicPlanners.jl
[6] https://github.com/JuliaPlanners/PDDL.jl

zado, bem como a sintaxe do PDDL e a estrutura dos arquivos de domínio e problema utilizados por essa linguagem, os quais são conceitos apresentados neste capítulo. Além disso, os planejadores consistem numa parte essencial do planejamento automatizado, sendo eles as ferramentas que receberão os domínios e problemas que serão modelados e que apresentarão uma solução, sendo então importante o conhecimento de suas características e possíveis limitações.

3 Especificação do jogo Puzznic

Puzznic consiste em um jogo eletrônico de combinação de peças, semelhante a outros jogos como Brix, desenvolvido e lançado pela Taito para arcades em 1989 (SCOTT, 2014), o qual foi depois portado para outras plataformas, como o Nintendo Entertainment System (NES) (TAITO, 1990).

A versão que será usada como referência neste artigo será a versão norte-americana para o console NES.

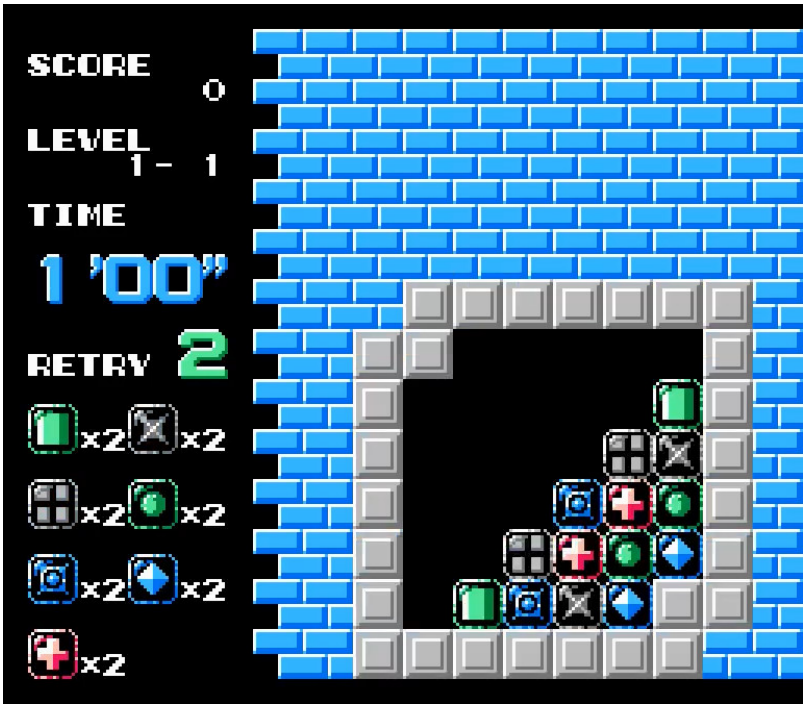
3.1 Objetivo

Em Puzznic, o objetivo do jogador é combinar peças que possuam cores e formatos correspondentes para que sejam removidas do jogo. O objetivo final do jogador é eliminar todas as peças presentes no quebra-cabeça dentro de um tempo limite (TAITO, 1990).

A Figura 1 apresenta o quebra-cabeça 1 da fase 1, referido como fase 1-1, do Puzznic em sua versão para o console NES. A fase em si pode ser visualizada principalmente à direita da figura, enquanto informações acerca da pontuação do jogador (score), o tempo da fase (time), as peças a serem combinadas e demais informações podem ser visualizadas à esquerda.

A pontuação pode ser considerada como um objetivo secundário do jogador, mas não interfere com a progressão do jogo em si.

Figura 1 – Fase 1-1 da versão de NES do jogo eletrônico Puzznic



Fonte: Taito

3.2 Regras

Como todo jogo eletrônico, o Puzznic possui um conjunto de regras que definem os critérios para vitória ou derrota, bem como as possíveis ações que o jogador pode tomar e as suas consequências. Tais regras serão exploradas a seguir.

3.2.1 Critérios para vitória

Para que um jogador complete uma fase no Puzznic, dois critérios devem ser atendidos:

- Todas as peças devem ser combinadas e, portanto, eliminadas da fase;
- O tempo limite, o qual é contado regressivamente, não deve chegar a zero.

Além disso, o jogador não conseguirá suceder em um quebra-cabeça caso possua somente uma única peça de uma cor e formato específico, ou seja, não conseguirá eliminar a peça pois não há outra correspondente. Essas situações podem ocorrer em quebra-cabeças como na Figura 2, em que o quebra-cabeça possui mais do que uma dupla de peças de um determinado padrão, onde o jogador pode chegar no estado da Figura 3. Todavia, tal situação não acarretará em uma derrota imediata, salvo quando o jogador não possuir tentativas restantes, o que ocasionará a exibição de uma mensagem, como na Figura 4, afirmando que não há como finalizar o quebra-cabeça.

Figura 2 – Fase 2-2 da versão de NES do jogo eletrônico Puzznic

Fonte: Taito

3.2.2 Fases

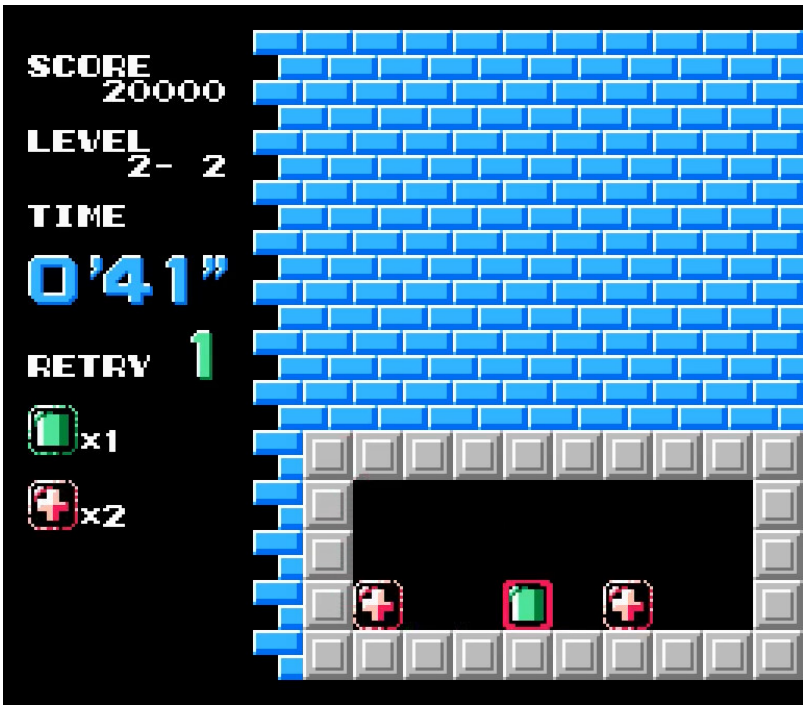
O jogo possui um total de dezesseis fases, cada uma contendo dez quebra-cabeças (TAITO, 1990). As fases seguem a nomenclatura fase X-Y, onde X e Y correspondem, respectivamente, à fase propriamente dita e o quebra-cabeça da fase. A fase atual pode ser vista na parte esquerda com as demais informações, como pode ser visto na Figura 1, onde o jogador está na fase 1 e em seu primeiro quebra-cabeça.

As peças são dispostas de maneira única em cada um dos quebra-cabeças. Além disso, o tempo limite varia de acordo com cada quebra-cabeça, sendo comum quebra-cabeças de uma mesma fase terem tempo limite iguais, embora isso não seja uma regra.

3.2.3 Ações

As principais ações disponíveis ao jogador para interagir com a fase e os objetos são a de mover uma peça ou mover o cursor. As peças podem ser movidas pelo jogador horizontalmente e caem quando são movidas para um espaço sem o suporte de uma outra peça ou um bloco delimitador dos limites do quebra-cabeça. Dessa forma, mover uma peça que serve de suporte para outra fará com que a peça posicionada acima caia. Uma peça pode ser movida para cima somente com o uso de blocos elevadores (TAITO, 1990). A movimentação de peças é feita a partir de um cursor que deve ser movido para cima da peça que se deseja mover. Quando uma peça é movida, o cursor segue a peça durante o

Figura 3 – Fase 2-2 sem possibilidade de vitória da versão de NES do jogo eletrônico Puzznic



Fonte: Taito

movimento, sendo verdadeiro também durante uma queda caso o jogador não movimente o cursor novamente.

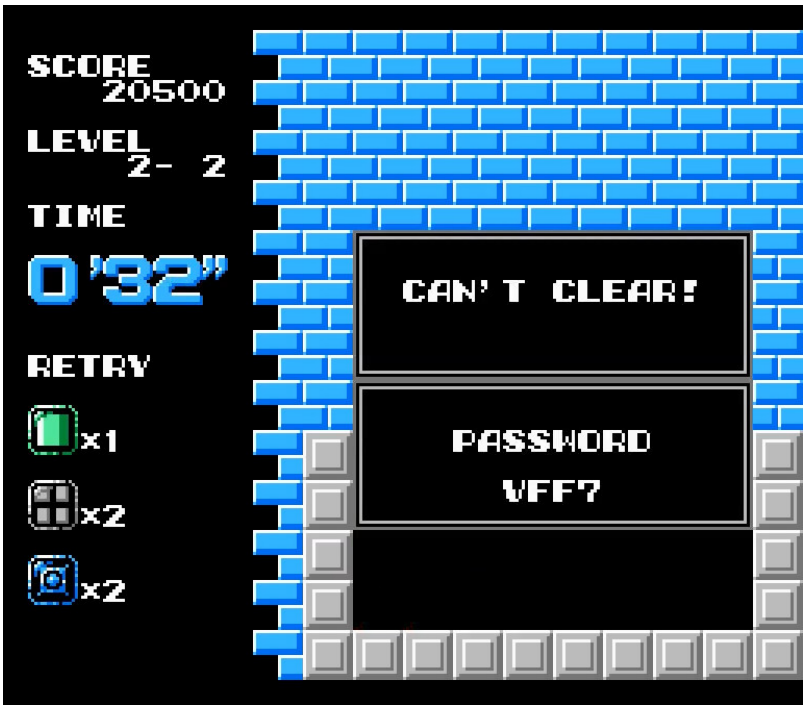
Além disso, o jogador pode escolher tentar um quebra-cabeça novamente. O jogador tem o direito a duas tentativas as quais não são devolvidas ao completar o quebra-cabeça. Tentar um quebra-cabeça novamente também não redefine o tempo limite, ou seja, o jogador não ganha tempo adicional (TAITO, 1990).

3.2.4 Pontuação

A pontuação do jogador é adquirida no decorrer do jogo e é acumulada de quebra-cabeça à quebra-cabeça. As formas com que o jogador obtém pontos são:

- A combinação de duas peças recompensa o jogador com 100 pontos;
- A combinação de três peças recompensa o jogador com 200 pontos;
- Caso combinações ocorram em cadeia em consequência de um único movimento, o jogador é recompensado pontos extras;
- Caso quatro ou mais peças sejam combinadas ao mesmo tempo, o jogador é recompensado pontos extras;

Figura 4 – Mensagem de impossibilidade de limpeza de fase da versão de NES do jogo eletrônico Puzznic



Fonte: Taito

- Ao finalizar um quebra-cabeça, o jogador é recompensado com pontos por cada segundo restante do tempo limite. O valor premiado equivale ao tempo restante multiplicado pelo produto da fase atual por cinquenta;
- Ao finalizar um quebra-cabeça, o jogador é recompensado com pontos bônus. Quanto maior for a fase, maior será esse valor.

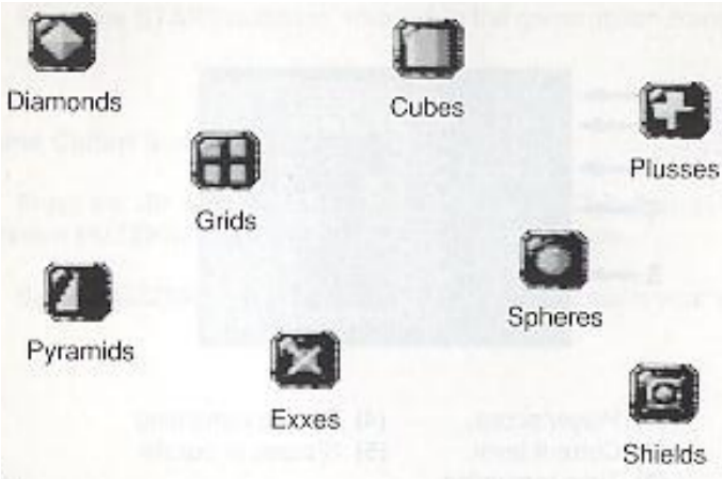
3.2.5 Objetos

O Puzznic possui três tipos de objetos, sendo eles as peças, os blocos elevadores e os blocos delimitadores. As peças são os principais objetos do Puzznic, visto que o objetivo do jogo é eliminá-las dos quebras-cabeça.

As peças são consideradas combinadas somente quando estiverem conectadas verticalmente ou horizontalmente e quando as suas cores e formatos forem iguais (TAITO, 1990). Os formatos das peças podem ser observados na Figura 5. Já em relação às cores, as peças podem assumir uma das seguintes cores: verde, vermelho, azul ou cinza.

Os blocos elevadores consistem em blocos que se movimentam continuamente e repetidamente em uma direção vertical ou horizontal. Embora o jogador não possa interagir diretamente com tais blocos, peças podem ser colocadas sobre esses blocos para movimentá-los e até mesmo combinar blocos, servindo também como obstáculos em certos

Figura 5 – Formatos das peças da versão de NES do jogo eletrônico Puzznic



Fonte: Taito (1990)

quebra-cabeças. Esses blocos possuem a mesma aparência dos blocos delimitadores.

Os blocos delimitadores são blocos cinzas que delimitam o espaço do quebra-cabeça e podem ser visualizados em todos os quebra-cabeças. Blocos delimitadores podem ser visualizados na Figura 3, sendo os blocos os quais as peças se mantém sobre.

4 Modelo

Neste capítulo, será apresentado o modelo do Puzznic em PDDL de maneira segmentada, tendo em vista que o modelo busca representar um sub-domínio do Puzznic, sem a existência de blocos elevadores. O primeiro tópico apresentará o domínio do Puzznic, especificado no arquivo de domínio, explicando todas as características do domínio e como elas implementam as físicas do Puzznic. O segundo tópico apresentará a estrutura dos arquivos de problema, abordando também como eles são gerados e construídos. O último tópico apresentará como a visualização das soluções é realizada, introduzindo a ferramenta desenvolvida para auxiliar nesse quesito.

O domínio possui duas versões: uma sem a implementação de um cursor e outra com a implementação de um cursor. O cursor é o meio pelo qual o jogador movimenta as peças, limitando o intervalo de tempo entre suas ações. Dessa forma, uma versão com um cursor foi implementada, visando restringir as ações que devem ser executadas dentro de uma janela de tempo, como as os movimentos das peças, visando a semelhança com as capacidades comuns de um ser humano. A separação do domínio em duas versões se dá em razão do fato de que embora o cursor seja uma adição relativamente simples, ela possui um grande impacto sob a performance do domínio. Como a segunda versão consta em uma adição ao conteúdo da primeira, as explicações serão focadas no conteúdo da primeira versão, com um subtópico para explicar onde existirem diferenças no arquivo de domínio.

Todos os códigos e arquivos mencionados neste capítulo estão disponíveis no Github[1].

4.1 Arquivo de domínio

Iniciaremos com o arquivo de domínio, que conterá - como explicado anteriormente - a especificação das "físicas" do problema, ou seja, o arquivo do domínio do Puzznic será onde o comportamento e as regras do jogo serão especificados, bem como as ações do jogador e suas consequências.

4.1.1 Especificações iniciais

O Código 4.1 apresenta as especificações do nome, os requisitos e os tipos do domínio. O nome do domínio, especificado na linha 1, é definido, de maneira objetiva, como `puzznic`.

[1] https://github.com/UnB-SAT/tcc-samuel-buters

Nas linhas 2 à 3, estão especificados os requisitos do domínio, sendo eles: `:strips`, que permite o uso de efeitos de adição e remoção básicos como especificado no STRIPS; `:typing`, que permite a utilização de tipos; `:negative-preconditions`, que permite a utilização de negação em pré-condições; `:conditional-effects`, permite a utilização de `when` para expressar efeitos condicionais; e `:numeric-fluents`, permite a utilização do bloco `:function`, possibilitando a representação de variáveis numéricas em um domínio além de operações sobre elas.

Por último, nas linhas 4 à 8, estão especificados os tipos do domínio. O tipo `block` corresponde a um bloco genérico, ou seja, pode corresponder a qualquer tipo de bloco, como uma peça combinável ou bloco delimitador. Os tipos `movable-block` e `immovable-block` correspondem a sub-tipos de `block`, separando os blocos em blocos móveis e imóveis. Os blocos imóveis correspondem aos blocos delimitadores enquanto os móveis correspondem às peças combináveis e blocos vazios. O tipo `matching-block` corresponde a um sub-tipo do `movable-block`, específico para as peças combináveis. Esses sub-tipos são utilizados para otimizar o número de blocos a serem explorados como parâmetros em algumas ações, como nas ações de permutação em que o primeiro parâmetro sempre será uma peça e o segundo sempre será um bloco vazio.

Código 4.1 – Especificações iniciais

```
1 (domain puzznic)
2 (:requirements :strips :typing :negative-preconditions :conditional-effects
3 :numeric-fluents)
4 (:types
5     block - object
6     movable-block immovable-block - block
7     matching-block - movable-block
8 )
```

Fonte: De autoria própria

4.1.2 Funções e predicados

O Código 4.2 apresenta o bloco de funções e predicados do domínio. Nas linhas 1 à 3 está especificada a única função do domínio, denominada `num-steps`, correspondendo ao número de ações de permutação realizadas. Essa função, através de sua minimização, é utilizada como uma métrica para encontrar planos com o menor número de ações de permutação. Ela é necessária visto que o interesse está em encontrar planos com o menor número de ações do jogador e não o menor número de ações do domínio, logo que as ações presentes no domínio simulam as físicas do Puzznic, bem como as ações do jogador.

Nas linhas 4 à 25 estão especificados os predicados do domínio. A estrutura geral para representar o Puzznic consiste na utilização do posicionamento relativo das peças entre si - semelhante à forma com que o Pipe Mania foi trabalhado por Banci (2022) -

para definir os quebras-cabeças, onde cada peça possui a informação acerca de quais peças estão ao seu redor. Os predicados responsáveis por prover tal característica ao domínio podem ser vistos da linha 15 à linha 18, sendo eles os predicados `left`, `right`, `up` e `down`. Por exemplo, o predicado `(left ?b1 ?b2)` corresponde dizer que o bloco `b1` possui o bloco `b2` à sua esquerda. A mesma lógica é utilizada nos demais predicados.

Tendo em vista que a estrutura geral consiste num posicionamento relativo entre as peças, não é possível a existência de espaços sem blocos. Logo, para a representação de espaços vazios, serão utilizados blocos vazios, semelhante a estratégias implementadas em alguns jogos eletrônicos como o Minecraft (Minecraft Wiki, 2023). Como é necessário a eventual transformação de uma peça combinatória em um bloco vazio, logo que o objetivo final é eliminar todas as peças combinatórias, esses blocos vazios são determinados através de um predicado em vez de um tipo, sendo esse predicado o `is-empty`, presente na linha 5.

O predicado da linha 6 especifica se um bloco é imóvel, ou seja, se é um bloco delimitador. Esse predicado é necessário pois em algumas ações, onde não é possível restringir o parâmetro para `immovable-block`, é necessário saber se um dos parâmetros corresponde a um bloco delimitador.

Os predicados das linhas 8 à 9 especificam propriedades acerca da queda das peças combinatórias. Caso a peça esteja em queda, o predicado `is-falling` será verdadeiro para a peça. Já o predicado `is-on-falling-cooldown` é utilizado no ciclo de queda das peças - o qual será explicado mais detalhadamente no tópico de ações - especificando que a peça já sofreu queda na fase do ciclo atual.

Os predicados das linhas 11 à 13 especificam propriedades acerca da combinação de peças. Caso a peça esteja combinando o predicado `is-matching` será verdadeiro para a peça. O predicado `is-marked` se refere a propriedade de uma peça estar marcada para verificar por combinações com peças ao seu redor. Caso uma peça possa vir a cair em razão de um combinação, o predicado `might-fall` será verdadeiro para a peça. O processo de combinação será explicado mais detalhadamente no tópico de ações.

O predicado `same-color` presente na linha 20 especifica que duas peças, `b1` e `b2`, possuem a mesma cor e serve como critério para a combinação de peças. Esse predicado é especificado de maneira bidirecional, ou seja, é declarado em ambas direções nos arquivos de problemas.

Os predicados das linhas 22 à 26 correspondem a *flags*. Os predicados `blocks-are-falling` e `at-end-of-falling-cycle` utilizados no processo de queda, indicando se o processo de queda está ocorrendo e se o estado atual está no fim do ciclo de queda, respectivamente. O predicado `has-time-swapped` é utilizado durante o ciclo de queda para restringir ações de permutação. Por último, os predicados `blocks-are-matchin`

`g`, `blocks-are-priority-matching` e `has-match-caused-falling` são *flags* utilizadas no processo de combinação, indicando se o processo de combinação está ocorrendo, se o processo de combinação prioritária está ocorrendo e se o processo de combinação ocasionou queda de blocos, respectivamente. No domínio, a queda de um bloco tem prioridade sobre a combinação de blocos, com exceção das combinações prioritárias.

Código 4.2 – Funções e predicados

```
(:functions
    (num-steps)
)
(:predicates
    (is-empty ?b - block)
    (is-immovable ?b - block)

    (is-falling ?b - matching-block)
    (is-on-falling-cooldown ?b - matching-block)

    (is-marked ?b - matching-block)
    (is-matching ?b - matching-block)
    (might-fall ?b - matching-block)

    (left ?b1 - block ?b2 - block)
    (right ?b1 - block ?b2 - block)
    (up ?b1 - block ?b2 - block)
    (down ?b1 - block ?b2 - block)

    (same-color ?b1 - matching-block ?b2 - matching-block)

    (has-time-swapped)

    (blocks-are-falling)
    (blocks-are-matching)
    (blocks-are-priority-matching)
    (at-the-end-of-falling-cycle)
    (has-match-caused-falling))
```

Fonte: De autoria própria

4.1.3 Ações

As ações de movimentações de peça pelo jogador são implementadas como ações de permutação de posições entre peças combinatórias e blocos vazios. Existem dois tipos de permutações, a permutação padrão e a permutação dentro de uma janela de tempo, ambas possuindo ações para a permutação à direita e à esquerda.

A permutação padrão corresponde a uma permutação realizada em um momento onde a queda de blocos não está ocorrendo. Como as ações para a esquerda e direita funcionam de maneira análoga, a ação de permutação à esquerda será utilizada como exemplo.

O Código 4.3 apresenta os parâmetros e pré-condições da ação de permutação para esquerda.

Nas linhas 2 à 3 estão especificados os parâmetros da ação. O parâmetro `?b1` corresponde à peça combinatória que será movida enquanto o parâmetro `?b2` corresponde ao bloco vazio para o qual a peça será movida. Os outros parâmetros correspondem aos blocos ao redor da peça combinatória e do bloco vazio, sendo necessários para a atualização dos posicionamentos relativos das peças ao final da ação, bem como outros efeitos.

Nas linhas 4 à 22 estão especificadas as pré-condições da ação. Os predicados da linha 5 à 6 especificam que a ação não é executável caso blocos estejam caindo ou combinando. Os predicados das linhas 8 à 9 especificam que o parâmetro `?b1` não deve ser vazio enquanto o `?b2` deve ser vazio. O predicado da linha 11 especifica que o bloco `?b1` não deve estar caindo. Os predicados das linhas 13 à 21 especificam o posicionamento relativo dos blocos, como, por exemplo, na linha 13 é especificado que o bloco do parâmetro `?b2` deve estar a esquerda da peça combinatória do parâmetro `?b1`.

Código 4.3 – Parâmetros e pré-condições da ação de permutação

```
1  (:action SWAP-BLOCK-LEFT
2      :parameters (?b1 - matching-block ?b2 - movable-block ?b1up ?b1down
3      ?b1right ?b2up ?b2down ?b2left - block)
4      :precondition (and
5                      (not (blocks-are-matching))
6                      (not (blocks-are-falling))
7
8                      (not (is-empty ?b1))
9                      (is-empty ?b2)
10
11                     (not (is-falling ?b1))
12
13                     (left ?b1 ?b2)
14
15                     (right ?b1 ?b1right)
16                     (up ?b1 ?b1up)
17                     (down ?b1 ?b1down)
18
19                     (left ?b2 ?b2left)
20                     (up ?b2 ?b2up)
21                     (down ?b2 ?b2down))
```

Fonte: De autoria própria

O Código 4.4 especifica os efeitos não condicionais da ação.

Na linha 2 é especificado o incremento da função `num-steps` que, como explicado anteriormente, corresponde ao número de ações de permutação realizadas. Os predicados das linhas 4 à 23 especificam as posições relativas atualizadas entre os blocos, enquanto os predicados das linhas 25 à 39 negam as posições antigas.

Na Figura 6 é possível visualizar a ação de permutação à esquerda, onde a peça

Código 4.4 – Efeitos não condicionais da ação de permutação

```
1  :effect (and
2              (increase (num-steps) 1)
3
4              (left ?b2 ?b1)
5              (right ?b1 ?b2)
6
7              (left ?b1 ?b2left)
8              (right ?b2left ?b1)
9
10             (down ?b1 ?b2down)
11             (up ?b2down ?b1)
12
13             (up ?b1 ?b2up)
14             (down ?b2up ?b1)
15
16             (right ?b2 ?b1right)
17             (left ?b1right ?b2)
18
19             (down ?b2 ?b1down)
20             (up ?b1down ?b2)
21
22             (up ?b2 ?b1up)
23             (down ?b1up ?b2)
24
25             (not (right ?b2 ?b1))
26             (not (left ?b1right ?b1))
27             (not (down ?b1up ?b1))
28             (not (up ?b1down ?b1))
29             (not (right ?b2left ?b2))
30             (not (down ?b2up ?b2))
31             (not (up ?b2down ?b2))
32
33             (not (left ?b1 ?b2))
34             (not (right ?b1 ?b1right))
35             (not (up ?b1 ?b1up))
36             (not (down ?b1 ?b1down))
37             (not (left ?b2 ?b2left))
38             (not (up ?b2 ?b2up))
39             (not (down ?b2 ?b2down))
```

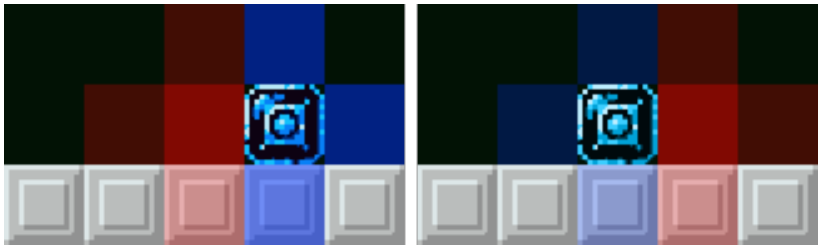
Fonte: De autoria própria

azul à direita troca de posição com o bloco vazio à esquerda. Os blocos ao redor da peça estão destacados como azul enquanto os blocos ao redor do bloco vazio, bem como o próprio bloco vazio, estão destacados em vermelho, sendo assim possível visualizar a atualização do posicionamento relativo dos blocos envolvidos.

O Código 4.5 especifica os efeitos condicionais da ação.

O efeito condicional da linha 1 à 22 ocorre quando algum bloco ao redor da nova posição da peça combinatória possui a mesma cor, não seja vazia e nem esteja caindo. Esse efeito condicional trata da possibilidade em que a peça combinatória é movida para uma posição adjacente a uma peça de mesma cor, ocasionando a sua combinação, desde que a

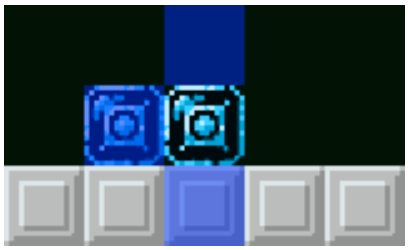
Figura 6 – Ação de permutação



Fonte: De autoria própria

peça não tenha sido movida para uma posição sem suporte. Essa combinação também é considerada prioritária, ou seja, ela tem prioridade sobre qualquer queda de bloco causada pela ação de permutação, especificado através da *flag* na linha 20. Na Figura 7 é possível visualizar um exemplo em que o efeito ocorrerá com uma peça azul, a qual checa a sua volta por peças candidatas – destacado em azul – e encontra uma outra peça azul à sua esquerda.

Figura 7 – Efeito condicional de combinação



Fonte: De autoria própria

O efeito condicional da linha 24 à 29 ocorre quando `?b2down` é um bloco vazio, ou seja, quando o bloco abaixo da nova posição da peça combinatória `?b1` é vazio. Esse efeito condicional trata da possibilidade em que a peça combinatória é movida para um espaço sem suporte, ocasionando a sua queda. Na Figura 8 é possível visualizar um exemplo em que o efeito ocorrerá a uma peça que foi permutada para esquerda.

Figura 8 – Efeito condicional de queda

Fonte: De autoria própria

O efeito condicional da linha 31 à 38 ocorre quando `?b1up` não é um bloco vazio e não é um bloco delimitador, ou seja, caso o bloco acima da posição inicial da peça

combinatória `?b1` também for uma peça combinatória. Esse efeito condicional trata da possibilidade em que uma peça combinatória esteja acima da que moveremos, causando a sua queda. Na Figura 9 é possível visualizar um exemplo em que o efeito ocorrerá por conta da peça azul ter sido permutada para esquerda, fazendo com a peça verde acima da peça azul caia.

Figura 9 – Efeito condicional de queda de peça acima do bloco permutado

Fonte: De autoria própria

A permutação dentro de uma janela de tempo funciona de maneira similar à ação padrão, mas pode ser executada somente no fim do ciclo da queda de peças. No Código 4.6 é possível visualizar a diferença que permite com que a ação seja executada durante o fim do ciclo. Em vez de possuir a condição `(not (blocks-are-falling))`, que restringiria a sua execução durante a queda de blocos, essa ação possui a pré-condição `(at-end-of-falling-cycle)`. Além disso, a *flag* `(has-time-swapped)` é utilizada para restringir o uso de uma única ação de permutação durante a janela de tempo, a qual pode ser encontrada tanto nas pré-condições e efeitos da ação. Ambas as ações foram separadas visando restringir a execução desses efeitos e pré-condições somente para esse tipo de permutação.

Seguindo para a próxima ação que lida com movimentação de peças, embora não mais controlada pelo jogador, temos a ação de queda. Essa ação funciona em conjunto com múltiplas ações que controlam o ciclo de queda, funcionando da seguinte maneira:

1. Cada peça combinatória em queda deve sofrer a ação de queda individualmente, ocasionando com que as peças entrem em tempo de recarga;
2. Caso não haja mais peças em queda, as *flags* que especificam a queda de blocos e fim do ciclo são desativadas e o ciclo de queda é finalizado; caso o contrário, a *flag* de fim de ciclo é ativada;
3. Durante o fim do ciclo, uma única ação de permutação de peças dentro de uma janela de tempo ou ações de combinação de peças podem ser executadas, não exclusivamente.
4. A *flag* de fim de ciclo é desativada e todos os blocos em queda têm seus tempo de recarga removidos, voltando à etapa 1.

Código 4.5 – Efeitos condicionais da ação de permutação

```
1  (when
2      (and
3          (not (is-empty ?b2down))
4          (or
5              (and
6                  (not (is-empty ?b2up))
7                  (not (is-falling ?b2up))
8                  (same-color ?b1 ?b2up))
9              (and
10                 (not (is-empty ?b2down))
11                 (not (is-falling ?b2down))
12                 (same-color ?b1 ?b2down))
13             (and
14                 (not (is-empty ?b2left))
15                 (not (is-falling ?b2left))
16                 (same-color ?b1 ?b2left))))
17
18     (and
19         (blocks-are-matching)
20         (blocks-are-priority-matching)
21         (is-matching ?b1)
22         (is-marked ?b1)))
23
24 (when
25     (is-empty ?b2down)
26     (and
27         (is-falling ?b1)
28         (not (is-on-falling-cooldown ?b1))
29         (blocks-are-falling)))
30
31 (when
32     (and
33         (not (is-empty ?b1up))
34         (not (is-immovable ?b1up)))
35     (and
36         (is-falling ?b1up)
37         (not (is-on-falling-cooldown ?b1up))
38         (blocks-are-falling)))
```

Fonte: De autoria própria

A abordagem do ciclo foi escolhida de tal forma que force os planejadores a causarem a queda de todas as peças especificadas como em queda – não possibilitando a busca por planos que tomem vantagem de quedas parciais – além de funcionar para um número arbitrário de peças, sem a utilização de fluentes numéricos, também permitindo com que ações de jogador sejam executadas durante a queda, semelhante ao funcionamento no jogo Puzznic.

O Código 4.7 apresenta os parâmetros e pré-condições da ação de queda de um bloco. Similar à ação de permutação, os parâmetros correspondem à peça combinatória que cairá, o bloco abaixo da peça o qual permutará de posição com a peça e os blocos posicionados ao redor de ambos os blocos, respectivamente. Os predicados da linha 5

Código 4.6 – Diferença nas pré-condições da ação de permutação dentro de uma janela de tempo

```
(:action TIMED-SWAP-BLOCK-LEFT
    :parameters (?b1 - matching-block ?b2 - movable-block ?b1up ?b1down
    ?b1right ?b2up ?b2down ?b2left - block)
    :precondition (and
                    (not (blocks-are-matching))
                    (at-the-end-of-falling-cycle)
                    (not (has-time-swapped))
                    [...]
    :effect (and
                [...]
                (has-time-swapped)
```

Fonte: De autoria própria

à 7 especificam que a ação só pode ser executada enquanto a *flag* de queda de blocos está ativa, o estado não esteja no fim do ciclo e não esteja ocorrendo uma combinação prioritária de peças. Os predicados das linhas 9 à 15 especificam que a peça `?b1` deve estar caindo, que não deve estar em tempo de recarga e não deve ser vazia, enquanto também especifica que o bloco abaixo da peça deve estar vazio e não deve estar caindo. Os predicados da linha 17 à 25 especificam o posicionamento relativos dos parâmetros.

Código 4.7 – Parâmetros e pré-condições da ação de queda

```
1  (:action FALL-BLOCK
2      :parameters (?b1 - matching-block ?b2 - movable-block ?b1left
3      ?b1right ?b1up ?b2left ?b2right ?b2down - block)
4      :precondition (and
5          (not (at-the-end-of-falling-cycle))
6          (blocks-are-falling)
7          (not (blocks-are-priority-matching))
8
9          (not (is-on-falling-cooldown ?b1))
10
11         (is-falling ?b1)
12         (not (is-falling ?b2))
13
14         (not (is-empty ?b1))
15         (is-empty ?b2)
16
17         (down ?b1 ?b2)
18
19         (left ?b1 ?b1left)
20         (right ?b1 ?b1right)
21         (up ?b1 ?b1up)
22
23         (left ?b2 ?b2left)
24         (right ?b2 ?b2right)
25         (down ?b2 ?b2down))
```

Fonte: De autoria própria

O Código 4.8 apresenta os efeitos não condicionais da ação de queda. Também

semelhante à ação de permutação, a maioria dos efeitos não condicionais são relacionados à atualização do posicionamento relativo dos parâmetros, como pode ser visto da linha 2 à 37, tendo como único efeito não referente ao posicionamento o da linha 39, o qual especifica que a peça combinatória entra em tempo de recarga ao cair.

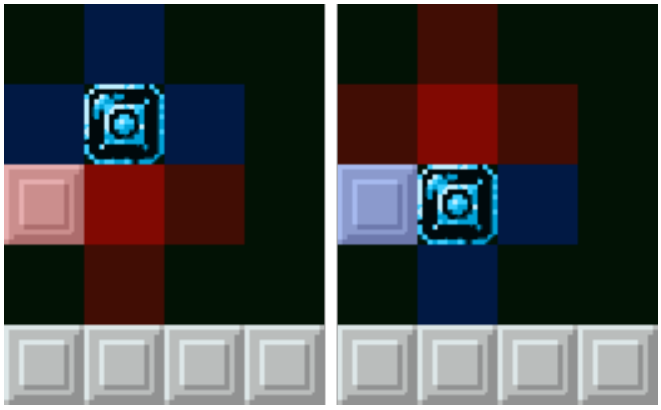
Código 4.8 – Efeitos não condicionais da ação de queda

```
1  :effect (and
2      (up ?b1 ?b2)
3      (down ?b2 ?b1)
4
5      (left ?b1 ?b2left)
6      (right ?b2left ?b1)
7
8      (right ?b1 ?b2right)
9      (left ?b2right ?b1)
10
11     (down ?b1 ?b2down)
12     (up ?b2down ?b1)
13
14     (left ?b2 ?b1left)
15     (right ?b1left ?b2)
16
17     (right ?b2 ?b1right)
18     (left ?b1right ?b2)
19
20     (up ?b2 ?b1up)
21     (down ?b1up ?b2)
22
23     (not (left ?b1 ?b1left))
24     (not (right ?b1 ?b1right))
25     (not (up ?b1 ?b1up))
26     (not (down ?b1 ?b2))
27     (not (left ?b2 ?b2left))
28     (not (right ?b2 ?b2right))
29     (not (down ?b2 ?b2down))
30
31     (not (right ?b1left ?b1))
32     (not (left ?b1right ?b1))
33     (not (down ?b1up ?b1))
34     (not (up ?b2 ?b1))
35     (not (right ?b2left ?b2))
36     (not (left ?b2right ?b2))
37     (not (up ?b2down ?b2))
38
39     (is-on-falling-cooldown ?b1)
```

Fonte: De autoria própria

Na Figura 10 é possível visualizar a ação de queda, onde a peça azul troca de posição com o bloco vazio logo abaixo dela. Os blocos ao redor da peça estão destacados como azul enquanto os blocos ao redor do bloco vazio, bem como o próprio bloco vazio, estão destacados em vermelho, sendo assim possível visualizar a atualização do posicionamento relativo dos blocos envolvidos.

Figura 10 – Ação de queda



Fonte: De autoria própria

O Código 4.9 apresenta os efeitos condicionais da ação de queda.

O primeiro efeito condicional, da linha 1 à 25, ocorre quando o bloco abaixo da nova posição da peça combinatória não é vazio e não está caindo. Esse efeito trata da situação em qual a peça deve parar de cair, logo que encontrou suporte em um bloco sólido. Além disso, esse efeito possui um outro efeito condicional, como pode ser visto da linha 9 à 25, ocorrendo quando um dos blocos ao redor da nova posição da peça seja uma peça da mesma cor e não seja vazia, ocasionando a sua combinação. Na Figura 11 é possível visualizar um exemplo em que o efeito de fim de queda ocorrerá, onde a peça azul da Figura 10 caiu até se encontrar com um bloco delimitador. Já a Figura 12 exemplifica uma situação em que a peça azul cai sobre uma outra peça azul, ocasionando o efeito de combinação.

Figura 11 – Efeito condicional de fim de queda

Fonte: De autoria própria

O segundo efeito condicional, da linha 27 à 33, ocorre quando temos uma peça combinatória acima da peça que está caindo. Esse efeito trata da situação da queda de peças empilhadas. Na Figura 13 é possível visualizar um exemplo em que o efeito ocorrerá

Figura 12 – Efeito condicional de fim de queda com combinação

Fonte: De autoria própria

logo que a peça verde foi permutada para direita, ocasionando a queda da peça azul logo acima dela que ocasionará a queda da peça vermelha acima da peça azul.

Figura 13 – Efeito condicional de queda de peças empilhadas

Fonte: De autoria própria

O Código 4.10 apresenta a ação de término de queda. Essa ação é responsável por desativar a *flag* `blocks-are-falling`. As suas pré-condições, presentes na linha 2 à 5, especificam que a ação pode ser executada quando a *flag* está ativada e quando todas as peças do problema não estejam caindo. Essa ação também desativa a *flag* `at-the-end-of-falling-cycle` para prevenir que um novo ciclo de queda comece no seu fim.

O Código 4.11 apresenta a ação de ativação da *flag* de fim de ciclo. Essa ação se torna disponível quando todos as peças em queda estiverem em tempo de recarga, sendo a única ação executável no problema durante esse estado.

O Código 4.12 apresenta a ação de desativação da *flag* de fim de ciclo. Essa ação pode ser tomada enquanto a *flag* de fim de ciclo está ativa, sendo a única forma de desativá-la.

Código 4.9 – Efeitos condicionais da ação de queda

```
(when
    (and
        (not (is-empty ?b2down))
        (not (is-falling ?b2down)))

    (and
        (not (is-falling ?b1))

        (when
            (or
                (and
                    (not (is-empty ?b2left))
                    (same-color ?b1 ?b2left))
                (and
                    (not (is-empty ?b2right))
                    (same-color ?b1 ?b2right))
                (and
                    (not (is-empty ?b2down))
                    (same-color ?b1 ?b2down))
            )

            (and
                (is-matching ?b1)
                (is-marked ?b1)
                (blocks-are-matching)))))

(when
    (not (is-empty ?b1up))

    (and
        (is-falling ?b1up)
        (not (is-on-falling-cooldown ?b1up))
        (blocks-are-falling)))
```

Fonte: De autoria própria

Código 4.10 – Ação de término de queda

```
(:action END-FALLING
    :precondition (and
        (blocks-are-falling)
        (forall (?b - matching-block)
            (not (is-falling ?b))))
    :effect (and
        (not (blocks-are-falling))
        (not (at-the-end-of-falling-cycle))))
```

Fonte: De autoria própria

O último conjunto de ações é referente às ações de combinação de peças, possuindo uma abordagem com intuito semelhante à abordagem do ciclo de queda, visando forçar os planejadores à combinar todas as peças possíveis em um determinado estado. A combinação de blocos funciona através das operações de marcar, espalhar e remover marcas. Quando uma peça combinatória é movida para uma posição adjacente à outra peça de

Código 4.11 – Ação de ativação da *flag* de fim de ciclo

```
(:action ACTIVATE-AT-END-OF-FALLING-CYCLE
    :precondition (and
        (blocks-are-falling)
        (not (at-the-end-of-falling-cycle))
        (exists (?b - matching-block) (is-falling ?b))
        (forall (?b - matching-block)
            (or
                (not (is-falling ?b))
                (is-on-falling-cooldown ?b)
                (and
                    (is-falling ?b)
                    (is-on-falling-cooldown ?b)))))
    :effect (at-the-end-of-falling-cycle)
)
```

Fonte: De autoria própria

Código 4.12 – Ação de desativação de *flag* de fim de ciclo

```
(:action DEACTIVATE-AT-END-OF-FALLING-CYCLE
    :precondition (and
        (at-the-end-of-falling-cycle)
        (not (blocks-are-matching)))
    :effect (and
        (not (at-the-end-of-falling-cycle))
        (forall (?b - matching-block)
            (not (is-on-falling-cooldown ?b)))))
```

Fonte: De autoria própria

mesma cor, a peça movida é especificada como combinando e marcada através dos predicados `is-matching` e `is-marked`, respectivamente. Uma peça marcada terá os seus blocos adjacentes checados por peças da mesma cor, especificando-as como combinando e marcadas caso existam. Quando não houver mais peças candidatas, a marcação é removida da peça, deixando-a somente como combinando. Quando não houver mais peças marcadas, a combinação ocorre, transformando todas as peças combinando em blocos vazios.

Semelhante às demais ações, as ações de espalhamento de marca são análogas em todas as direções, dessa forma sendo escolhida a ação de espalhamento de marca à esquerda, como exemplo, apresentada no Código 4.13. Os predicados das linhas 4 à 8 consistem nas *flags* de restrição da ação, onde é possível visualizar que a queda toma prioridade sobre a ação de combinação, a não ser quando a combinação é prioritária ou quando o estado está no fim do ciclo de queda. Os predicados das linhas 10 à 16 especificam características e restrições da peça origem da marca e da peça alvo. Os predicados das linhas 18 à 19 especificam o posicionamento das peças. Os efeitos da linha 21 à 22 é onde o espalhamento da marca ocorre. O efeito condicional das linhas 25 à 35 ocorre quando existe uma peça acima da peça que será marcada a qual não possui a mesma cor e não está combinando. Esse efeito condicional trata da possibilidade de uma combinação poder

causar a queda de peças. É importante perceber que esse efeito marca peças que têm a possibilidade cair, mas não que irão necessariamente cair.

Na Figura 14 é possível visualizar a ação de espalhamento de marca, onde a peça azul, especificada como marcada pelo símbolo de coração preenchido, espalha a marca para a outra peça azul à sua esquerda. A Figura 15 apresenta um exemplo do efeito condicional onde a peça amarela sobre a peça azul pode vir a cair caso ela não faça parte de alguma combinação.

Figura 14 – Ação de espalhamento de marca

Fonte: De autoria própria

Figura 15 – Efeito condicional de possível queda

Fonte: De autoria própria

O Código 4.14 apresenta a ação de remoção de marca. Os predicados das linhas 4 à 8 consistem nas *flags* de restrição da ação, similar à ação anterior. O predicado da linha 10 especifica que a peça alvo, `?b1`, deve estar marcada. Os predicados das linhas 12 à 15 especificam o posicionamento dos blocos adjacentes à peça alvo. Os predicados das linhas 17 à 21 especificam as condições para que um bloco não seja elegível para espalhamento, sendo elas caso a peça seja vazia, já esteja combinando, esteja caindo ou não seja da mesma cor. Essas condições são checadas para os demais blocos adjacentes. O efeito da linha 40 portanto remove a marca da peça `?b1`, deixando-a somente como combinando.

Na Figura 16 é possível visualizar a ação de remoção de marca, onde a peça azul à direita não possui peças candidatas adjacentes e portanto remove a sua marca. Todavia, a peça ainda especificada como combinando através do símbolo do coração vazio.

O Código 4.15 apresenta as pré-condições da ação de combinação de blocos. Os predicados das linhas 3 à 7 consistem nas *flags* de restrição da ação, similar às duas ações anteriores. A condição da linha 8 à 9 especifica que para toda peça, nenhuma deve estar

Figura 16 – Ação de remoção de marca

Fonte: De autoria própria

marcada. Essa ação portanto só é executável quando não existirem mais peças marcadas, ou seja, quando não haver mais como espalhar a combinação.

O Código 4.16 apresenta os efeitos da ação de combinação de blocos. Os efeitos das linhas 2 à 3 desativam as *flags* de combinação, enquanto o efeito da linha 4 à 10 define todas as peças que estejam combinando como vazias, além de especificá-las como não combinando. O efeito condicional da linha 12 à 26 ocorre quando a *flag* `has-match-caused-falling` é verdadeira. Esse efeito trata do caso mencionado anteriormente quando a combinação pode ter causado a queda de peças, ativando a *flag* de queda de blocos e desativando a *flag* de possibilidade de queda por combinação, além de checar todas as peças que foram marcadas como sujeitas a uma possível queda e especificando-as como em queda caso não estejam em processo de combinação.

Na Figura 17 é possível visualizar a ação de combinação, onde todas as peças especificadas como combinando são especificadas como vazias. A Figura 18 apresenta um exemplo do efeito condicional onde a peça amarela sobre a peça azul, que estava marcada com risco de queda, irá de fato cair por não fazer parte da combinação.

Figura 17 – Ação de combinação

Fonte: De autoria própria

Figura 18 – Efeito condicional da ação combinação

Fonte: De autoria própria

Código 4.13 – Ação de espalhamento de marca à esquerda

```
(:action SPREAD-MATCH-LEFT
    :parameters (?b1 ?b2 - matching-block ?b2up - block)
    :precondition (and
        (blocks-are-matching)
        (or
            (blocks-are-priority-matching)
            (not (blocks-are-falling))
            (at-the-end-of-falling-cycle))

        (not (is-empty ?b1))
        (not (is-empty ?b2))

        (not (is-matching ?b2))
        (not (is-falling ?b2))

        (same-color ?b1 ?b2)

        (left ?b1 ?b2)
        (up ?b2 ?b2up))
    :effect (and
        (is-matching ?b2)
        (is-marked ?b2)
        (not (might-fall ?b2))

        (when
            (and
                (not (is-empty ?b2up))
                (not (is-matching ?b2up))
                (not (same-color ?b2 ?b2up)))

            (and
                (might-fall ?b2up)
                (not (is-on-falling-cooldown ?b2up))
                (has-match-caused-falling))))
)
```

Fonte: De autoria própria

Código 4.14 – Ação de remoção de marca

```
(:action REMOVE-MARK
    :parameters (?b1 - matching-block ?b1left ?b1up ?b1right ?b1down - block)
    :precondition (and
        (blocks-are-matching)
        (or
            (blocks-are-priority-matching)
            (not (blocks-are-falling))
            (at-the-end-of-falling-cycle))

        (is-marked ?b1)

        (left ?b1 ?b1left)
        (up ?b1 ?b1up)
        (right ?b1 ?b1right)
        (down ?b1 ?b1down)

        (or
            (is-empty ?b1left)
            (is-matching ?b1left)
            (is-falling ?b1left)
            (not (same-color ?b1 ?b1left)))

        (or
            (is-empty ?b1up)
            (is-matching ?b1up)
            (is-falling ?b1up)
            (not (same-color ?b1 ?b1up)))

        (or
            (is-empty ?b1right)
            (is-matching ?b1right)
            (is-falling ?b1right)
            (not (same-color ?b1 ?b1right)))

        (or
            (is-empty ?b1down)
            (is-matching ?b1down)
            (is-falling ?b1down)
            (not (same-color ?b1 ?b1down))))
    :effect (not (is-marked ?b1)))
```

Fonte: De autoria própria

Código 4.15 – Pré-condições da ação de combinação de blocos

```
(:action MATCH-BLOCKS
    :precondition (and
        (blocks-are-matching)
        (or
            (blocks-are-priority-matching)
            (not (blocks-are-falling))
            (at-the-end-of-falling-cycle))
        (forall (?b - matching-block)
            (not (is-marked ?b))))
```

Fonte: De autoria própria

Código 4.16 – Efeitos da ação de combinação de blocos

```
:effect (and
    (not (blocks-are-matching))
    (not (blocks-are-priority-matching))
    (forall (?b - matching-block)
        (when
            (is-matching ?b)

            (and
                (is-empty ?b)
                (not (is-matching ?b)))))

    (when
        (has-match-caused-falling)

        (and
            (blocks-are-falling)
            (not (has-match-caused-falling))
            (forall (?b - matching-block)
                (when
                    (and
                        (might-fall ?b)
                        (not (is-matching ?b)))

                    (and
                        (is-falling ?b)
                        (not (might-fall ?b))))))))
```

Fonte: De autoria própria

4.1.4 Diferenças do domínio com cursor

Nos predicados do domínio, o domínio com cursor possui um predicado adicional, apresentado no Código 4.17. O predicado `cursor-at` especifica o bloco o qual o cursor está presente sobre.

Código 4.17 – Predicado *cursor-at*

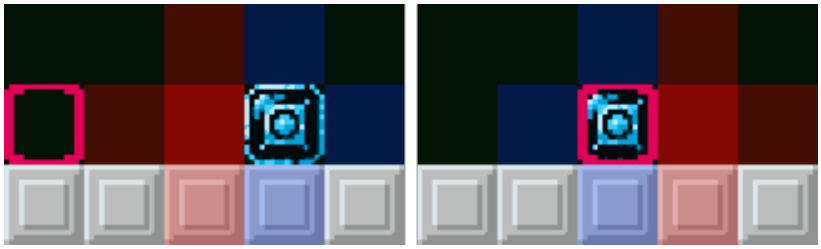
```
1 (cursor-at ?b - block)
```

Fonte: De autoria própria

Nas ações do domínio, o domínio com cursor apresenta diferenças nas ações de permutação e na ação de queda, além de incluir um novo conjunto de ações e possuir uma diferença indireta em relação a ação de desativação da *flag* de fim de ciclo.

Nas ações de permutação, mantendo o exemplo da permutação à esquerda, é possível visualizar, no Código 4.18, que a ação padrão de permutação possui um parâmetro extra, `?cursorblock`. Esse parâmetro corresponde ao bloco que contém o cursor sobre ele, especificado pelo predicado da linha 8, sendo necessário para a atualização da posição do cursor ao fim da ação, como pode ser visto nas linhas 12 à 13. Isso simula o jogador movendo a peça combinatória com o seu cursor, o qual segue a posição da peça. Na Figura 19 é possível visualizar a posição do cursor sendo atualizada para a posição da peça movida.

Figura 19 – Ação de permutação no domínio com cursor

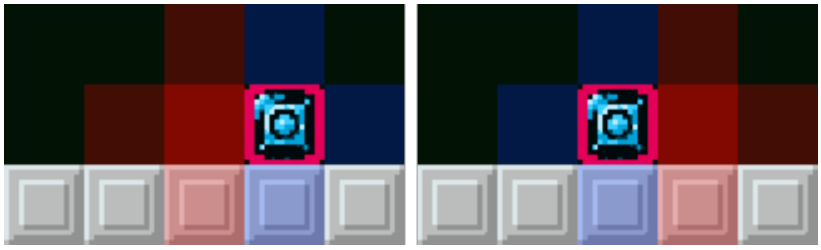


Fonte: De autoria própria

Já a permutação dentro de uma janela de tempo possui uma mudança com um impacto maior, tornando a ação mais restrita. No Código 4.19 observa-se que a ação possui um predicado de pré-condição extra, na linha 7, possibilitando a execução da ação somente caso o cursor esteja sobre a peça combinatória `?b1`. Nessa ação não há a necessidade de atualizar a posição do cursor pois ele já está sobre a peça `?b1` que será movida. Na Figura 20 é possível visualizar a ação de permutação em janela de tempo no domínio do cursor, necessitando a presença do cursor sobre a peça azul sendo permutada.

Dessa forma, para possibilitar com que a ação de permutação dentro de uma janela de tempo tenha uma pré-condição alcançável, o domínio implementa a movimentação do

Figura 20 – Ação de permutação em janela de tempo no domínio com cursor



Fonte: De autoria própria

Código 4.18 – Diferença na ação de permutação na versão do domínio com cursor

```
1  (:action SWAP-BLOCK-LEFT
2      :parameters (?b1 - matching-block ?b2 - movable-block ?b1up ?b1down
3      ?b1right ?b2up ?b2down ?b2left ?cursorblock - block)
4      :precondition (and
5                      (not (blocks-are-matching))
6                      (not (blocks-are-falling))
7                      [...]
8                      (cursor-at ?cursorblock))
9      :effect (and
10                 (increase (num-steps) 1)
11                 [...]
12                 (cursor-at ?b1)
13                 (not (cursor-at ?cursorblock))
14                 [...]
```

Fonte: De autoria própria

Código 4.19 – Diferença na ação de permutação dentro de uma janela de tempo na versão do domínio com cursor

```
1  (:action TIMED-SWAP-BLOCK-LEFT
2      [...]
3      :precondition (and
4                      (not (blocks-are-matching))
5                      (at-the-end-of-falling-cycle)
6
7                      (cursor-at ?b1)
8                      [...]
```

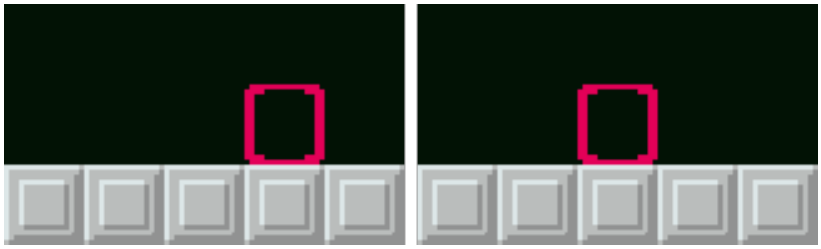
Fonte: De autoria própria

cursor para as quatro direções através de ações. Da mesma forma que as ações de permutação, essas ações funcionam de maneira análoga, sendo necessária somente a apresentação de uma delas, sendo escolhida a movimentação do cursor para a esquerda como exemplo.

O Código 4.20 apresenta a ação de movimentação do cursor para a esquerda. Essa ação é relativamente simples, utilizando somente dois parâmetros, presentes na linha 2, correspondendo ao bloco onde o cursor está localizado e ao bloco a sua esquerda, respectivamente. Essa ação, tal como a ação de permutação dentro de uma janela de tempo, só pode ser executada quando o estado atual estiver no fim do ciclo de queda e

caso não esteja ocorrendo combinação de peças. A *flag* `has-time-swapped` está presente nessa ação para que garantir que ela não seja executada após uma ação de permutação. Os efeitos podem ser visualizados nas linhas 10 à 16, correspondendo simplesmente à atualização da posição do cursor e os efeitos de restrição para que somente uma dessas ações possa ser executada durante o fim do ciclo. Na Figura 21 é possível visualizar a ação de movimentação de cursor para a esquerda.

Figura 21 – Ação de movimentação de cursor



Fonte: De autoria própria

As ações de movimentação cursor adicionam uma diferença indireta em relação à ação de desativação de *flag* de fim de ciclo, logo que essas ações também conseguem desativar a *flag* mencionada.

Código 4.20 – Ação de movimentação do cursor para a esquerda

```
1  (:action MOVE-CURSOR-LEFT
2      :parameters (?b1 ?b2 - block)
3      :precondition (and
4          (not (blocks-are-matching))
5          (at-the-end-of-falling-cycle)
6          (not (has-time-swapped))
7
8          (cursor-at ?b1)
9          (left ?b1 ?b2))
10     :effect (and
11         (cursor-at ?b2)
12         (not (cursor-at ?b1))
13
14         (not (at-the-end-of-falling-cycle))
15         (forall (?b - matching-block)
16             (not (is-on-falling-cooldown ?b)))))
```

Fonte: De autoria própria

Na ação de queda, a diferença está num efeito condicional extra que visa garantir com que o cursor não seja permutado caso ele esteja posicionado sobre o bloco com que a peça em queda irá permutar de lugar. Dessa forma, o cursor é posicionado sobre a peça em queda, como pode ser visto no Código 4.21.

Código 4.21 – Diferença na ação de queda na versão do domínio com cursor

```
(when
    (cursor-at ?b2)

    (and
        (cursor-at ?b1)
        (not (cursor-at ?b2))))
```

Fonte: De autoria própria

4.2 Arquivo de problema

Os arquivos de problemas do domínio do Puzznic devem especificar, para determinado problema, todos os blocos existentes e seus tipos, bem como o posicionamento relativos de todos os blocos e quais peças possuem a mesma cor entre si. Isso faz com que até mesmo os problemas mais simples contenham centenas linhas. Dessa forma, os arquivos de problemas são gerados através de um *script* com o auxílio de mapas construídos utilizando a ferramenta de código-aberto Tiled.

4.2.1 Criação de mapas

O primeiro passo para a geração de um arquivo de problema é a criação do mapa através da ferramenta Tiled. O Tiled consiste em uma ferramenta para criação de mapas para jogos eletrônicos baseados em ladrilhos, disponível no Github[2].

A primeira etapa consiste em criar um conjunto de blocos no Tiled utilizando um mapa de ladrilhos, de dimensão 32px de largura por 32px de altura, que possui os blocos que estarão presentes nos problemas, ou seja, os blocos delimitadores e todas as variações de cores das peças. O conjunto de blocos utilizado foi gerado a partir do mapa de ladrilhos da Figura 22 e está disponível no Github, não precisando ser criado novamente.

A próxima etapa consiste em criar o mapa propriamente dito. Durante a criação do mapa, as seguintes configurações devem ser utilizadas, não havendo restrições às dimensões de um mapa:

- Orientação: Ortogonal;
- Formato de camada de blocos: CSV;
- Ordem de renderizar dos blocos: Direita Abaixo;
- Tamanho do mapa: Fixo;
- Largura do bloco: 32px;

[2] https://github.com/mapeditor/tiled

Figura 22 – Mapa de ladrilhos utilizado

Fonte: De autoria própria

- Altura do bloco: 32px.

Com o mapa criado na ferramenta, pode-se desenhar o mapa referente ao problema, utilizando o conjunto de blocos criado anteriormente. Os únicos blocos que devem ser inseridos são os blocos delimitadores e as peças, utilizando somente uma camada. A Figura 23 apresenta um dos mapas criados e utilizados durante os testes dentro da ferramenta.

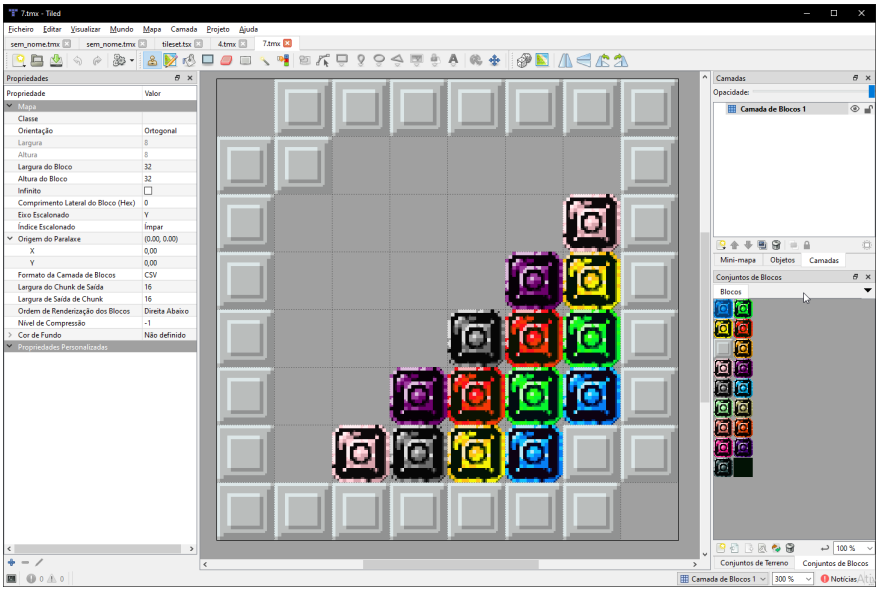
A última etapa consiste em exportar o mapa para que ele possa ser utilizado pelo *script* que irá gerar o arquivo de problema. Para isso, o mapa deve ser exportado como um arquivo JSON. Opcionalmente, o arquivo do mapa pode ser salvo para alterações e consultas futuras.

4.2.2 Geração do arquivo de problema

Para a geração dos arquivos de problema, um *script* na linguagem Python é utilizado. O *script* recebe o nome do problema como argumento. É esperado que uma pasta nomeada com o nome do problema esteja na pasta raíz do *script*. Essa pasta deve conter o arquivo JSON do mapa, também nomeado com o nome do problema seguido pela extensão ".json". Como exemplo, um problema nomeado "8" teria o seu arquivo de mapa exportado da pasta "8" com o nome "8.json".

Além disso, o *script* espera por um arquivo "config.json" que contenha o nome de

Figura 23 – Mapa do problema 7



Fonte: De autoria própria

todos os blocos presentes no mapa de ladrilhos, da esquerda para direita, de cima para baixo. O nome dos blocos são arbitrários, com exceção do bloco delimitador que deve ser nomeado como “stage”. O arquivo de configuração utilizado, correspondendo ao mapa de ladrilhos da figura 22, pode ser visualizado no Código 4.22.

Código 4.22 – Efeitos da ação de combinação de blocos

```
{
        "tilesetColors": [
                "blue",
                "green",
                "yellow",
                "red",
                "stage",
                "orange",
                "pink",
                "purple",
                "gray",
                "deepskyblue",
                "palegreen",
                "darkkhakiyellow",
                "samonred",
                "orangeredorange",
                "deeppink",
                "indigopurple",
                "darkslategray"
        ]
}
```

Fonte: De autoria própria

Com o arquivo de configuração pronto, bem como os mapas dispostos como espe-

cificado anteriormente, o *script* é executado da seguinte forma, utilizando como exemplo com mapa nomeado "exemplo":

```
$ python level2problem.py exemplo
```

A geração do arquivo de problema ocorre em duas etapas principais: a etapa de *parsing* e a etapa de geração de saída.

A etapa de *parsing* consiste no *parsing* do arquivo de mapa em um dicionário que contém uma lista de todos os blocos e um dicionário de todas as cores. O Código 4.23 apresenta um exemplo da estrutura desse dicionário, não necessariamente representando um mapa válido. A lista de blocos contém blocos representados como um dicionário que contempla informações acerca de quais blocos são adjacentes a ele, bem como se o bloco em questão é vazio ou móvel. Um exemplo de como os blocos são representados pode ser visto nas linhas 2 à 7 do Código 4.23. O dicionário de cores possui uma entrada para cada cor especificada no arquivo de configuração. Cada entrada possui uma lista de todos os blocos pertencentes à cor em questão. Um exemplo do dicionário de cores pode ser visto nas linhas 8 à 25 do Código 4.23.

A etapa de geração de saída utiliza os dicionários de blocos e cores da etapa anterior para gerar o arquivo de problema. A função responsável por esta etapa utiliza três variáveis principais que armazenam, como strings, os objetos, os predicados do argumento `:init` e os predicados do argumento `:goal`. Primeiramente, o algoritmo irá iterar sobre todos os blocos da lista de blocos, adicionando-o na variável de objetos junto ao seu tipo, bem como adicionando as suas características como predicados na variável de estado inicial. Em seguida, o algoritmo itera por todos os blocos de cada cor do dicionário de cores, especificando todos os blocos que possuem a mesma cor na variável de inicialização, além de definir cada bloco iterado como vazio na variável de meta. Por fim, o algoritmo formata as variáveis populadas dentro de uma *string* que recebe cada variável em seus respectivos lugares. Essa *string* possui toda a estrutura padrão do problema, incluindo a especificação da minimização da métrica `num-steps`, bem como a localização padrão do cursor, caso o domínio com cursor seja utilizado.

O *script* assume que os problemas não iniciarão com blocos caindo. Embora ele consiga gerar problemas a partir de mapas que constam com essas características, os problemas não serão válidos, logo que o *script* não inicializa os predicados referentes à queda.

Código 4.23 – Exemplo da estrutura de um dicionário de mapa

```
{
    "blocks": [
        {"surroundingBlocks": {"right": 1, "down": 2}, "isMatchingBlock": True},
        {"surroundingBlocks": {"left": 0, "right": 2, "down": 3}, "isMatchingBlock":
        ↪  False, "empty": True},
        {"surroundingBlocks": {"left": 1, "up": 0, "right": 3}, "isMatchingBlock":
        ↪  True},
        {"surroundingBlocks": {"left": 2, "up": 1}, "isMatchingBlock": False,
        ↪  "immovable": True}
    ],
    "colors": {
        "blue": [0],
        "green": [],
        "yellow": [],
        "red": [2],
        "orange": [],
        "pink": [],
        "purple": [],
        "gray": [],
        "deepskyblue": [],
        "palegreen": [],
        "darkkhakiyellow": [],
        "samonred": [],
        "orangeredorange": [],
        "deeppink": [],
        "indigopurple": [],
        "darkslategray": []
    }
}

```

Fonte: De autoria própria

4.3 Visualização dos planos

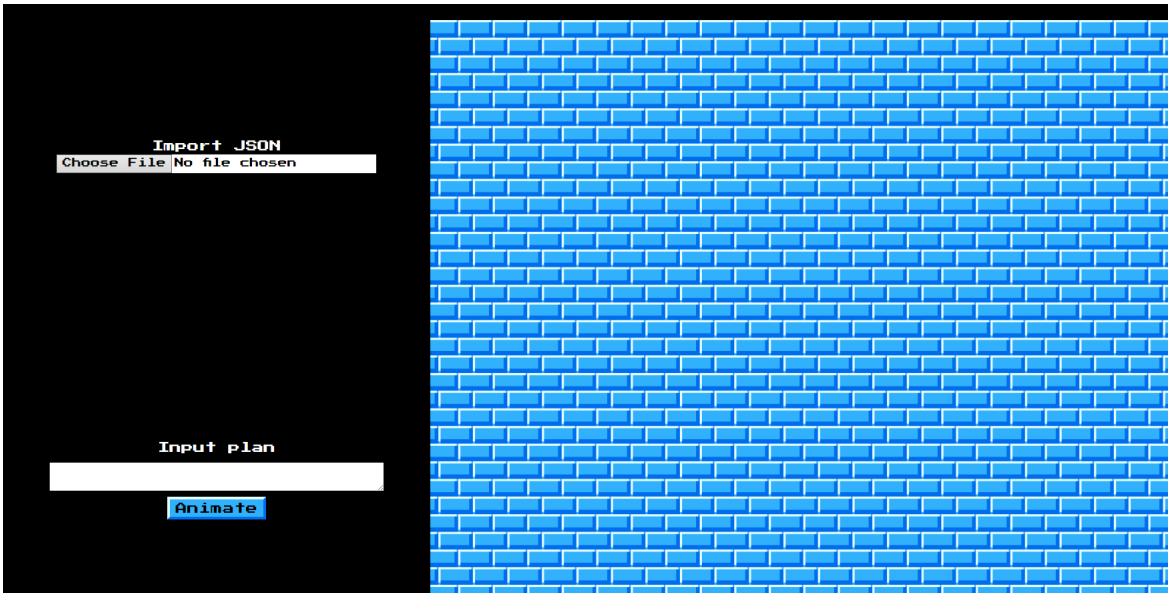
Para a visualização dos planos, uma aplicação web foi desenvolvida utilizando o *framework* Vue[3] em vez da utilização do Planimation, o qual foi considerado previamente. Isso se deu em conta da maior praticidade e controle no processo de representação e animação dos planos. A aplicação foi desenvolvida em paralelo com o modelo, auxiliando na validação dos planos encontrados, bem como no descobrimento de *bugs* no modelo.

Na Figura 24 é possível visualizar a interface inicial do visualizador. Nessa interface, primeiramente deve ser selecionado, na entrada "Import JSON", o arquivo JSON do mapa referente ao problema ao qual se deseja ser visualizado. Em seguida, a solução deve ser inserida na entrada "Input plan". Com isso, o usuário pode pressionar o botão "Animate" para gerar as animações e visualizar o plano, caso nenhum erro ocorra.

A próxima interface consiste na de visualização do plano em si, presente na Figura 25. A interface possui à esquerda os controles da animação do plano, possibilitando o

[3] https://vuejs.org

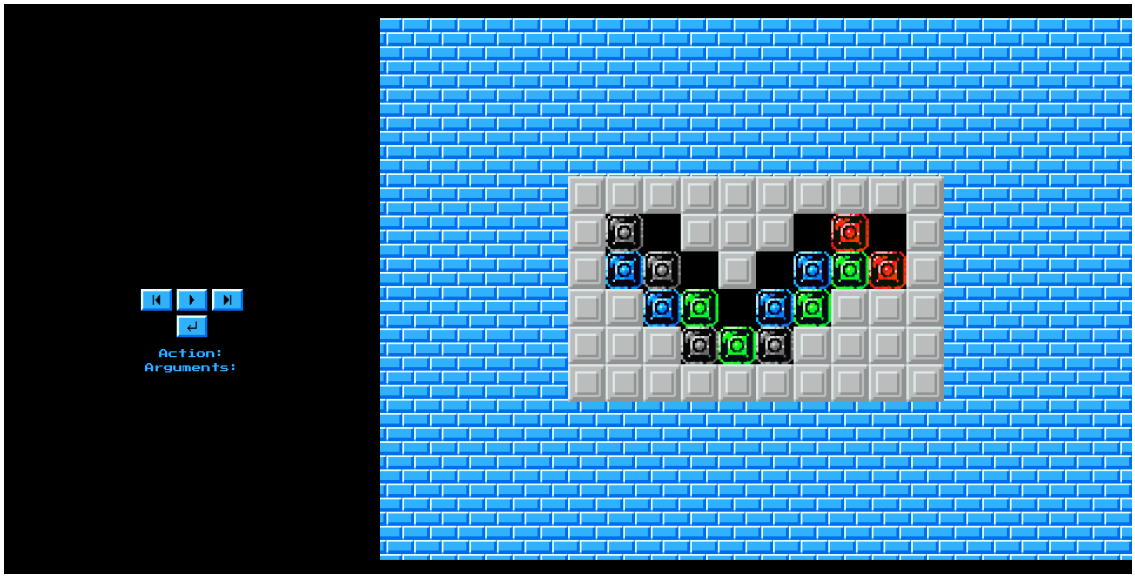
Figura 24 – Interface de entrada



Fonte: De autoria própria

usuário a reproduzir e pausar a animação, bem como ir ao passo anterior ou seguinte da animação. O botão abaixo dos de controle permite voltar à interface anterior para visualizar a solução de outro problema. Abaixo dos botões é possível visualizar a ação atual e argumentos da ação, sendo visível somente quando os passos da animação estão sendo controlados manualmente através dos botões de avançar e retroceder. À direita é onde a solução do problema pode ser visualizada.

Figura 25 – Interface de visualização



Fonte: De autoria própria

O funcionamento do visualizador pode ser explicado em três principais etapas:

1. Conversão do mapa: Esta etapa consiste na conversão do arquivo JSON do mapa alvo em uma grade de blocos. Essa conversão utiliza as informações de largura, altura e dados da camada de blocos, presente no arquivo JSON exportado. A informação de dados da camada de blocos é referente ao conjunto de blocos utilizados e portanto o visualizador necessita do mapa de ladrilhos utilizado para a geração do conjunto de blocos, como o da Figura 22, o qual é usado por padrão.

2. *Parsing* da solução: Esta etapa consiste na conversão da *string* fornecida como entrada em um *array* de passos do plano que será utilizado na geração das animações. O *parser* implementado funciona com o padrão de saída do planejador *ForwardPlanner* do *SymbolicPlanners.jl*, o qual foi utilizado.

3. Geração das animações: Esta etapa consiste na criação das animações do plano. A partir do *array* de passos gerados no passo anterior, as animações são geradas utilizando o nome das ações e seus argumentos. Essas animações são adicionadas à uma linha do tempo, ambos sendo objetos criados através da biblioteca anime.js[4]. Além disso, um novo *array* é criado o qual adiciona o tempo de início da animação ao passo do plano, sendo utilizado para controle manual da animação.

[4] https://github.com/juliangarnier/anime/

5 Experimentos

Com o intuito de avaliar o funcionamento e desempenho do modelo, foram criados diversos mapas, alguns provenientes do jogo Puzznic e outros criados com o intuito de testar as funcionalidades do domínio. Os experimentos foram realizados em um computador contendo um processador Ryzen 7 2700 com clock base de 3.2 GHz e 32 GBs de RAM.

O planejador utilizado foi o *ForwardPlanner*, o qual foi apresentado no capítulo dois. As configurações padrões do planejador foram utilizadas, logo que os melhores resultados foram obtidos com a sua utilização. Dessa forma, a heurística padrão foi utilizada, sendo ela a `GoalCount`, a qual conta o número de metas não satisfeitas. A especificação do problema utilizada é a `MinMetricGoal` que especifica que a métrica do problema, `num-steps`, deve ser minimizada. O Código 5.1 apresenta um exemplo de como um problema é solucionado, sendo possível visualizar as configurações mencionadas.

Código 5.1 – Exemplo de execução do problema 7

```
1 using PDDL, SymbolicPlanners
2
3 domain = load_domain("puzznic_domain.pddl")
4 problem = load_problem("problem-7.pddl")
5 state = initstate(domain, problem)
6 spec = MinMetricGoal(problem)
7
8 planner = ForwardPlanner()
9 sol = planner(domain, state, spec)
```

Fonte: De autoria própria

O modelo foi desenvolvido de forma incremental, buscando aumentar a sua sofisticação de forma gradual. Com isso, um total de 22 mapas foram criados para testar a capacidade do modelo em cada etapa do seu desenvolvimento. Os problemas foram nomeados numericamente em ordem cronológica, dessa forma problemas de números menores tendem a ser menos sofisticados e necessitam de menos funcionalidades para sua solução. Os problemas criados especificamente para testar as funcionalidades do domínio são:

- Problema 1: Testa a permutação de peças, bem como a combinação dupla horizontal.
- Problema 2: Semelhante ao problema 1, todavia utiliza duas cores.
- Problema 3: Testa a combinação dupla vertical.
- Problema 4: Testa a queda de uma única peça.

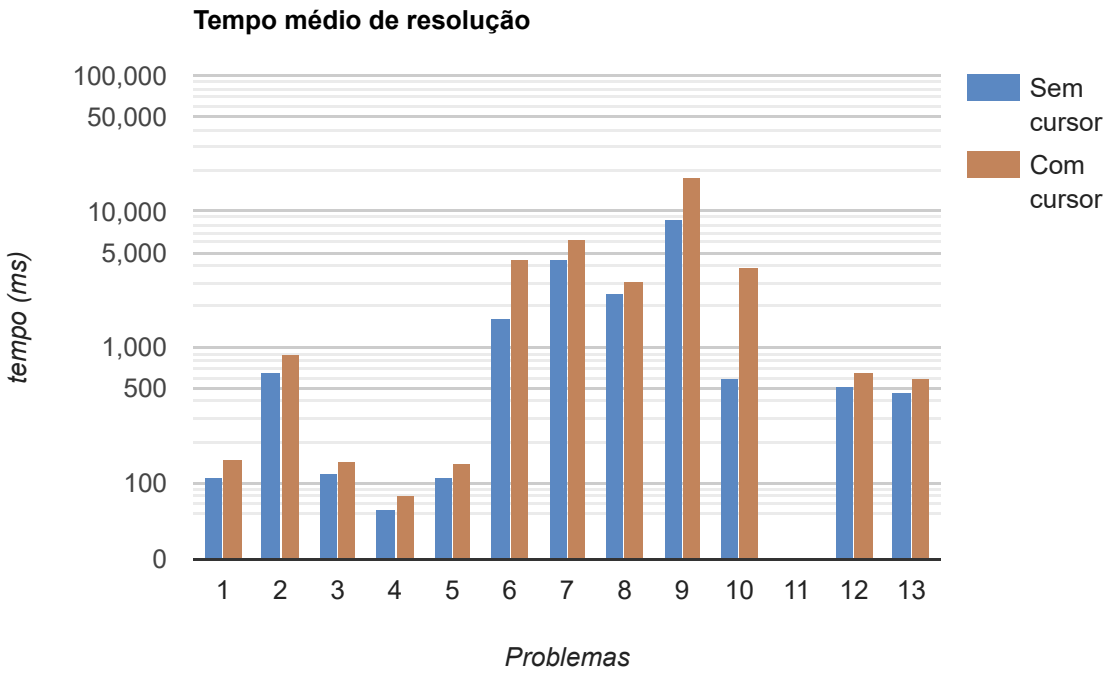
- Problema 5: Semelhante ao problema 4, todavia utiliza duas cores.
- Problema 6: Semelhante ao problema 5, todavia é possível a criação de um estado sem solução caso as peças forem combinadas em ordem errada.
- Problema 10: Testa a combinação de três ou mais peças simultaneamente.
- Problema 12: Testa a queda de múltiplas peças, a queda de peças ocasionadas por permutação de peças e a combinação de peças de cores diferentes ao mesmo tempo, todavia a utilização dessas características não é obrigatória.
- Problema 13: Semelhante ao problema 12, todavia a solução utilizada as características mencionadas obrigatoriamente.

O tempo médio de resolução das duas versões do domínio podem ser visualizados na Figura 26. É possível visualizar que o domínio com cursor obteve tempos maiores em todos os problemas em comparação com o domínio sem cursor, todavia a diferença de tempo das versões atuais não são tão grandes como em versões anteriores, onde a maior diferença de performance pode ser vista no problema 10, em que o domínio com cursor possui uma performance cerca de 7 vezes mais lenta que o domínio sem cursor. Entretanto, essas diferenças podem se tornar maiores de acordo com o aumento da sofisticação dos problemas ou até mesmo dependendo de características específicas do problema, como foi observado em versões anteriores, sendo observável também nas versões atuais com o problema 10, o qual consiste em um problema pouco sofisticado.

Além disso, é possível visualizar que o problema 11 não possui um valor definido para ambas as versões, logo que não foi possível encontrar uma solução para tal problema antes do esgotamento de recursos da máquina. Esse caso aconteceu nos demais problemas não presentes no gráfico, ou seja, os problemas 14 ao 22. Têm-se como hipótese que isso se deu simplesmente pelo fato dos problemas serem sofisticados, seja pela quantidade de blocos, peças e cores, bem como a característica das soluções desses problemas potencialmente exigirem estratégias específicas para que sejam alcançadas.

Ambos os problemas 11 e 20 são exemplos, logo que suas soluções foram alcançadas em versões anteriores do domínio que possuíam erros que ocasionavam comportamentos inesperados os quais permitiam a realização de ações inválidas, possibilitando soluções mais simples que ignoravam características sofisticadas de suas soluções. No problema 11 é esperado que o jogador crie uma ponte utilizando um empilhamento vertical das peças, como demonstrado na Figura 27, para permitir a combinação das peças verdes. Uma das falhas no domínio permitia com que o problema fosse resolvido em cerca de 4 minutos e 30 segundos, isso ocorrendo em razão de uma falha permitir a permutação de uma peça em queda, possibilitando com que uma solução fosse encontrada sem o empilhamento de peças, tornando a busca pela solução bem mais simples.

Figura 26 – Gráfico de tempo médio de resolução dos dois domínios

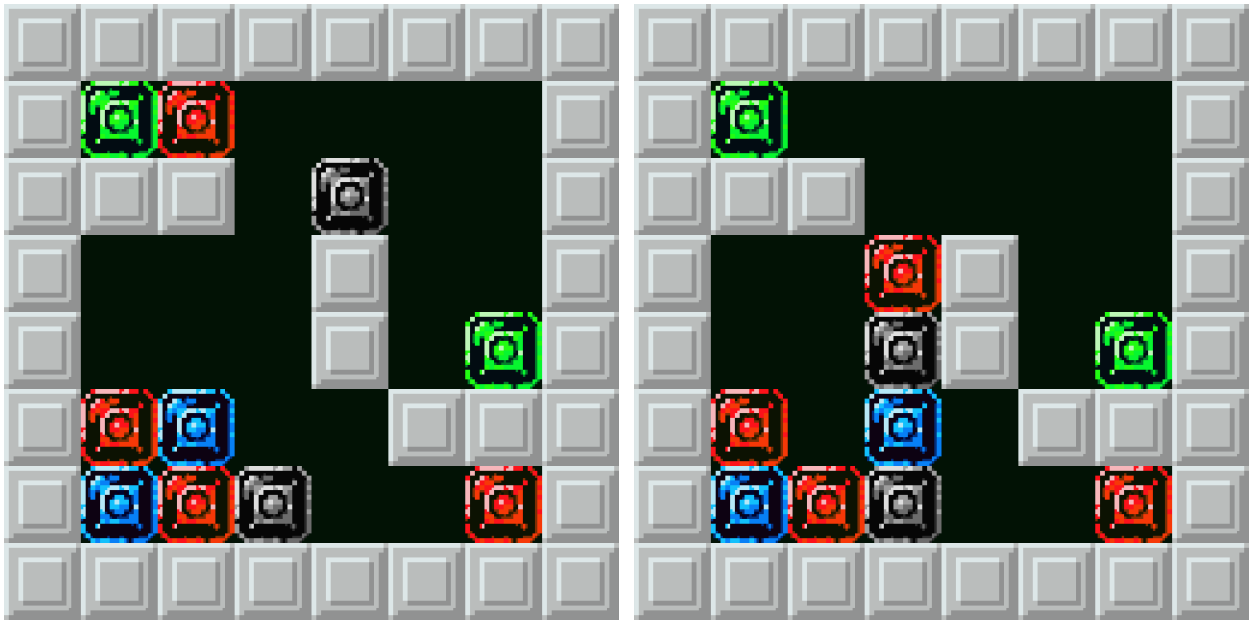


Fonte: De autoria própria

Dentro das otimizações realizadas, a definição de uma métrica e a sua utilização através da especificação de minimização de métrica foi a otimização com o impacto mais significativo. Inicialmente, a especificação de problema `MinStepsGoal` foi utilizada, especificando que a solução deve conter o menor número de passos. Todavia, isso acaba otimizando todas as ações do domínio, incluindo as ações referentes às físicas do Puzznic. Dessa forma, a função `num-steps` foi criada para ser utilizada como uma métrica que sofre incrementos nas ações de permutação, ou seja, as ações referentes às ações do jogador. Dessa forma, a métrica é utilizada para minimizar o número de ações de jogador realizadas através da especificação de problema `MinMetricGoal`.

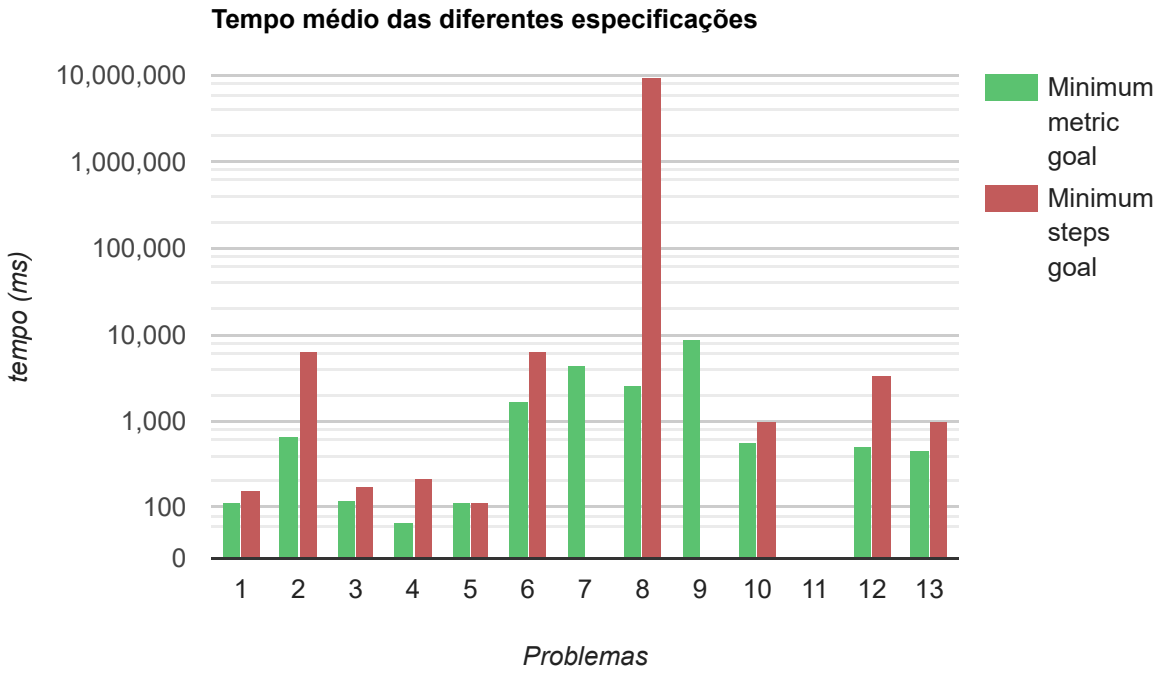
Na Figura 28 é possível visualizar a diferença dos tempos de execução usando ambas especificações nos domínios sem cursor. É observável que a utilização da especificação `MinMetricGoal` apresenta uma performance superior na maioria dos casos, com exceção do problema 5 em que os tempos são aproximadamente iguais. Além disso, a procura por planos utilizando a especificação `MinStepsGoal` não consegue achar soluções para problemas com certo grau de sofisticação sem esgotar os recursos da máquina ou sem utilizar tempos exorbitantes, como no caso do problema 8, que utilizou 9.258.335 milissegundos, cerca de 2 horas e 34 minutos, para encontrar uma solução, aproximadamente 367.103% do tempo de execução utilizando a `MinMetricGoal`.

Figura 27 – Utilização de empilhamento vertical no problema 11



Fonte: De autoria própria

Figura 28 – Gráfico de tempo médio de resolução das especificações utilizando o domínio sem cursor



Fonte: De autoria própria

Na Tabela 1 consta os resultados das buscas por soluções dos problemas 1 ao 13 utilizando ambas especificações nos dois domínios, possuindo os dados utilizados pelos

gráficos das Figuras 26 e 28. A sigla RE, significando *Runtime Error*, foi utilizada para os casos em que os recursos da máquina foram esgotados antes que uma solução pudesse ser encontrada.

Tabela 1 – Tabela dos tempos médios de execução em milissegundos dos domínios utilizando ambas especificações

	MinMetricGoal		MinStepsGoal	
Problema	Sem cursor	Com cursor	Sem cursor	Com cursor
1	112	148	153	172
2	660	902	6251	9682
3	118	147	172	217
4	65	81	214	332
5	112	139	115	147
6	1629	4490	6282	12764
7	4376	6213	RE	RE
8	2522	3053	9258335	11780291
9	8910	18040	RE	RE
10	582	3942	996	4026
11	RE	RE	RE	RE
12	522	648	3347	4609
13	464	582	952	1240

Fonte: De autoria própria

Para a resolução dos problemas, o interpretador do *PDDL.jl* foi utilizado. Dessa forma, caso a resolução dos problemas fosse feita utilizando o compilador em vez do interpretador, existe a possibilidade dos tempos de resolução serem significativamente menores, potencialmente resolvendo os problemas 10 vezes mais rapidamente (ZHI-XUAN, 2022). Entretanto, não foi possível utilizar o compilador com os domínios do Puzznic por conta de um problema indeterminado que ocasiona em buscas que não retornam qualquer resultado mesmo com horas de execução em todos os problemas. Em relação ao momento de confecção deste texto, uma *issue*[1] está aberta para discussão do problema em questão.

[1] https://github.com/JuliaPlanners/PDDL.jl/issues/19

6 Conclusão

O objetivo deste trabalho consistiu em desenvolver um modelo do jogo Puzznic em PDDL, com capacidade de solucionar problemas que não possuam a utilização de blocos elevadores. Foram desenvolvidos dois modelos: um com a presença de um cursor e outro sem. Além disso, foram desenvolvidos um *script* e uma aplicação para auxiliar na criação dos arquivos de problema e na validação dos planos encontrados, respectivamente.

Dentre os dois modelos, o modelo com o cursor consiste em uma representação mais fiel do jogo Puzznic, logo que o cursor limita as ações do planejador, buscando fazer com que os seus movimentos durante janelas pequenas de tempo se assemelhem às capacidades comuns de um ser humano. Em contraste, o domínio sem cursor permite com que os planejadores possam executar ações acima das capacidades humanas, como mover duas peças em lados opostos de um quebra-cabeça dentro de um período curto de tempo.

Tanto o processo de criação de arquivos de problemas e o visualizador de planos foram essenciais para o desenvolvimento dos modelos. Com o número de predicados necessários para especificar o estado de um problema, a especificação manual de problemas é inviável. Portanto, o desenvolvimento de um *script* junto com a utilização da aplicação Tiled para a criação de mapas e arquivos de problemas tornou possível a confecção de quebra-cabeças de maneira rápida e intuitiva, independente de suas sofisticações. O visualizador de planos se provou fundamental para validar os planos encontrados e depurar os modelos. Através de sua utilização, foi possível encontrar comportamentos indesejados em razão de *bugs* nos modelos, bem como comprovar se a modelagem estava correta.

Durante o processo de modelagem, vários desafios foram encontrados, todavia os mais proeminentes foram modelar a gravidade e a combinação de peças, ambas sendo ações que devem ocorrer automaticamente, sem a intervenção direta do jogador, simulando as físicas do jogo. O principal desafio em modelar essas ações foi garantir que ambas funcionem para um número arbitrário de peças, garantindo que todas as peças que devem ser combinadas sejam combinadas e todas as peças que devem cair caiam, sendo especialmente desafiador no caso da gravidade, em razão do número de parâmetros envolvidos. Esses desafios foram superados através de abordagens que definem estados específicos que restringem as ações que podem ser executadas, forçando os planejadores a realizarem a combinação ou queda de todas as peças envolvidas, funcionando como um ciclo no caso da queda.

Diante dos resultados obtidos, é observado que os modelos desenvolvidos representam apropriadamente o domínio do Puzznic dentro do escopo planejado, todavia a sua utilização, junto ao planejador escolhido, para a resolução de problemas apresentou

algumas dificuldades e limitações durante os experimentos. A resolução de problemas de alta sofisticação – envolvendo diversas peças e cores, bem como estratégias específicas de resolução – só foi possível em versões dos domínios que apresentavam falhas que possibilitaram a simplificação do problema, enquanto nas versões finais, com tais falhas corrigidas, as soluções não foram encontradas por conta da exaustão de recursos das máquinas.

Portanto, é proposto para trabalhos futuros a procura por mais otimizações ao domínio, a expansão de seu escopo - buscando incluir os blocos elevadores - e a realização de experimentos utilizando outros planejadores, tal como o *Fast Downward*.

Referências

BANCI, G. *Resolvendo Pipe Mania com Planejamento*. Bacharelado em Engenharia de Software — Universidade de Brasília, 2022. Citado na página 26.

ESPASA, J.; MIGUEL, I.; VILLARET, M. Plotting: A Planning Problem with Complex Transitions. In: SOLNON, C. (Ed.). *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. (Leibniz International Proceedings in Informatics (LIPIcs), v. 235), p. 22:1–22:17. ISBN 978-3-95977-240-2. ISSN 1868-8969. Disponível em: <https://drops.dagstuhl.de/opus/volltexte/2022/16651>. Citado na página 9.

FIKES, R. E.; NILSSON, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, v. 2, n. 3, p. 189–208, 1971. ISSN 0004-3702. Disponível em: <http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/strips.pdf>. Citado na página 14.

GHALLAB, M. et al. *PDDL | The Planning Domain Definition Language*. 1998. Disponível em: <https://planning.wiki/_citedpapers/pddl1998.pdf>. Citado 4 vezes nas páginas 12, 14, 15 e 16.

HASLUM, P. et al. *An Introduction to the Planning Domain Definition Language*. 1. ed. [S.l.: s.n.], 2019. Citado 6 vezes nas páginas 9, 11, 12, 14, 15 e 16.

Minecraft Wiki. *Air*. 2023. Disponível em: <https://minecraft.fandom.com/wiki/Air>. Citado na página 27.

RUSSEL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3. ed. [S.l.]: Prentice Hall, 2010. ISBN 978-0-13-604259-4. Citado na página 12.

SCOTT, J. *Puzznic*. 2014. Disponível em: <https://archive.org/details/arcade_puzznic>. Citado na página 19.

TAITO. *Puzznic: How to play*. [S.l.], 1990. 29 p. Citado 6 vezes nas páginas 9, 19, 21, 22, 23 e 24.

ZHI-XUAN, T. *PDDL.jl: An Extensible Interpreter and Compiler Interface for Fast and Flexible AI Planning*. Dissertação (Mestrado) — Massachusetts Institute of Technology, fev. 2022. Citado 2 vezes nas páginas 17 e 59.