UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA ABORDAGEM PARA MAXIMIZAÇÃO DA PRODUÇÃO DE RECURSOS EM JOGOS RTS

THIAGO FRANÇA NAVES

Uberlândia - Minas Gerais
2012

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

THIAGO FRANÇA NAVES

UMA ABORDAGEM PARA MAXIMIZAÇÃO DA PRODUÇÃO DE RECURSOS EM JOGOS RTS

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Inteligência Artificial.

Orientador:
Prof. Dr. Carlos Roberto Lopes

Uberlândia, Minas Gerais
2012

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU

N323a Naves, Thiago França, 1988-
Uma abordagem para maximização da produção de recursos em jogos RTS / Thiago França Naves. - 2012.
144 f. : il.

Orientador: Carlos Roberto Lopes.

Dissertação (mestrado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Computação - Teses. 2. Jogos por computador - Teses. I. Lopes, Carlos Roberto, 1962- II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Uma Abordagem para Maximização da Produção de Recursos em Jogos RTS**" por **Thiago França Naves** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 20 de Julho de 2012

Orientador: _____
Prof. Dr. Carlos Roberto Lopes
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Prof. Dr. Luiz Chaimowicz
Universidade Federal de Minas Gerais (UFMG) - Minas Gerais

_____
Prof. Dr. Márcia Aparecida Fernandes
Universidade Federal de Uberlândia (UFU) - Minas Gerais

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Julho de 2012

Autor:	**Thiago França Naves**
Título:	**Uma Abordagem para Maximização da Produção de Recursos em Jogos RTS**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

_____

Autor

Dedicatória

Dedico este trabalho aos meus Pais e familiares, pois sempre estiveram ao meu lado acreditando e apoiando todas minhas ideias, ambições e decisões.

Agradecimentos

Agradeço ...

Aos meus amados pais José Batista Naves e Marli França Naves pela dedicação, amor incondicional, confiança, apoio e carinho em todos os momentos. Obrigado por serem os principais responsáveis por ter me tornado uma pessoa feliz, honesta, verdadeira e humilde. Com muito amor e dedicação em tudo o que faço eu agradeço a vocês por tudo que são na minha vida.

Ao meu orientador Carlos Roberto Lopes que sempre acreditou no meu trabalho, me ajudou e confiou em mim. O senhor sempre deu asas as minhas ideias e liberdade para minha criatividade ser o motor das nossas pesquisas. Muito obrigado por tudo.

Aos amigos Lucas Clemente Vella, Leonardo de Sá Alt e Rodrigo Queiroz Saramago que foram grandes amigos durante o mestrado contribuindo para que conversas e discussões sobre tecnologia fossem interessantes e divertidas nos corredores e laboratórios da universidade.

Ao meu amigo e primo Murilo Naves de Oliveira que foi um grande amigo quando me mudei para Uberlândia para realizar meu mestrado. Você contribuiu para que os dias de convívio fossem mais divertidos e animados quando as dificuldades e o cansaço do dia a dia pareciam maiores.

A professora Márcia Aparecida Fernandes que me ensinou o quanto computação e pesquisas científicas podem ser muito mais interessantes e belas do que eu já imaginava. A senhora é um exemplo de dedicação e espelho para a minha jornada após o mestrado.

A todos os outros professores que fizeram parte da minha vida e me ajudaram nesta jornada.

A todos os meus familiares que sempre torceram pelo meu sucesso.

A todos os colegas e funcionários da Faculdade de Computação da Universidade Federal de Uberlândia.

A todos que estiveram ao meu lado durante este trabalho.

"Quando você faz de modo inusitado as coisas comuns da vida, você controla a atenção do mundo" (George Washington Carver)

Resumo

Jogos de estratégia em tempo real (jogos RTS) são um importante campo de pesquisa em planejamento com inteligência artificial. Nesses jogos temos que lidar com características que representam desafios para o planejamento, tais como restrições de tempo, efeitos numéricos e ações com grande número de pré-condições.

Jogos RTS são caracterizados por duas importantes fases. A primeira, onde é preciso reunir recursos e desenvolver um exército via produção de recursos. Na segunda, os recursos produzidos na primeira fase são utilizados em batalhas que envolvem defesa e ataque contra o inimigo. Deste modo, a primeira fase torna-se de vital importância para o sucesso do jogador dentro do jogo, e uma maximização na produção de recursos que eleve ao máximo o poder do exército desenvolvido, reflete diretamente nas chances de vitória. Essa pesquisa é focada na primeira fase do jogo.

Para maximizar a quantidade de recursos produzidos no jogo é necessário estipular metas que produzam recursos em tal escala, sendo consideradas metas com qualidade. Grande parte dos trabalhos na área de planejamento para jogos não consideram a busca por metas que tenham qualidade ou critérios que beneficiem o planejamento para serem utilizadas dentro do jogo. Com isso, para conseguir estipular tais metas é proposta uma abordagem que maximiza a produção de recursos utilizando *Simulated Annealing* (SA) junto com técnicas de planejamento. A abordagem utiliza busca estocástica para maximizar a produção de recursos gerando planos de ações que produzam tais recursos dentro do jogo e possam ser considerados metas a serem atingidas com qualidade para o planejamento.

Para o correto funcionamento deste trabalho, o SA foi adaptado para operar sobre o domínio de tempo real, e também foram desenvolvidos planejadores e verificadores de consistência para auxiliar em tal tarefa. Como resultado, a abordagem apresentou eficiência em seu uso dentro do ambiente de um jogo RTS, onde a produção de recursos foi capaz de equiparar-se com jogadores humanos durante os testes. Assim, essa pesquisa visa preencher uma lacuna presente nos trabalhos correlatos de planejamento para jogos RTS, em relação à produção de recursos baseada em critérios.

Palavras chave: busca estocástica; jogos; planejamento; recursos; tempo real.

Abstract

Real-time strategy Games (RTS) are an important field of research in artificial intelligence planning. In these games, we have to deal with characteristics that present challenges for the planning, such as time constraints, numerical effects and actions with many preconditions.

RTS games are characterized by two important phases. The first, where is necessary gather resources and build an army via resource production. In the second, the resources produced in the first phase are used in battles involving defense and attack against the enemy. Thus, the first phase becomes vitally important for the success of the player within the game, and maximize the production of resources to elevate the maximum power of the army developed, reflects directly on the chances of victory. This research is focused on the first phase of the game.

To maximize the amount of resources produced in the game is necessary to determine goals that produce resources such scale, considering quality goals. Large part of the works in planning for games not consider the search for goals that have quality or criteria that benefit planning for use within the game. Thus, to stipulate goals we propose an approach that maximizes the resource production using Simulated Annealing (SA) along with planning techniques. The approach uses stochastic search to maximize production of resources generating plans of action that produce such resources within the game and can be considered as goals with quality planning to be achieved.

For the correct operation of this work, the SA was adapted to operate on the domain of real time, and have also been developed planners and consistency checkers to assist it in this task. As a result, the approach was efficient in its use within the environment of an RTS game, where the resource production was able to match with human players during testing. Thus, this research aims to fill a gap present in related works of planning for RTS games, in relation to the production of resources based on criteria.

Keywords: games; planning; real-time; resources; stochastic search.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

ADL	Action Description Language
AG	Algoritmo Genético
BWAPI	Brood War API
IA	Inteligência Artificial
API	Application Program Interface
GPS	General Problem Solver
MEA	Means-end Analysis
PDDL	Planning Domain Definition Language
POP	Partial Order Planning
PSRC	Project Scheduling with Resource Constraints
RTS	Real-time Strategy
SA	Simulated Annealing
SGP	Sensory Graphplan
STRIPS	STanford Research Institute Problem Solver
UFU	Universidade Federal de Uberlândia

Capítulo 1

Introdução

Jogos de estratégia em tempo real (RTS) são uma das categorias mais populares de jogos de computador. Títulos recentes como os renomados *Warcraft 2* e *Stracraft 2* são exemplos dessa categoria. Esses jogos possuem diferentes tipos de classes de personagens e recursos, os quais são utilizados em batalhas. batalhas que podem ser disputadas entre jogadores humanos ou um agente jogador (software) controlado por um computador.

É possível identificar duas fases distintas em um jogo de RTS. A primeira, onde cada jogador começa a partida com algumas unidades militares e recursos e procura desenvolver esses ao máximo através da produção de recursos. Na fase seguinte, são travadas batalhas entre os jogadores e suas respectivas classes utilizando os recursos produzidos na fase anterior. Essas batalhas envolvem ataques ao inimigo e defesa da própria base. Dessa forma, uma eficiente produção de recursos durante a primeira fase torna-se de vital importância para o sucesso do jogador dentro do jogo.

Recursos em um jogo RTS são todos os tipos de recursos naturais (ouro, madeira e gás), construções base, unidades militares e civilização. Para obter um desejado conjunto de recursos é necessário executar ações que consigam produzir tais recursos dentro do mundo do jogo. Em geral, existem três tipos de ações relatadas à produção de recursos: ações que produzem recursos, ações que consomem recursos e ações que coletam recursos.

Uma vez que temos um conjunto de ações (plano) que produz um desejado conjunto de recursos (meta), é preciso executá-las para conduzir o jogo de um estado inicial de recursos em que ele encontra-se, a um estado final no qual o novo conjunto de recursos é atingido. Num jogo RTS é necessário maximizar a quantidade de recursos a serem produzidos, com o objetivo de aumentar a força do exército que será gerado pelo jogador durante a primeira fase de desenvolvimento. Para isto é necessário especificar metas adequadas para produção de recursos. É preciso evitar que sejam especificadas metas sem critério ou sem garantia de qualidade para o planejamento, algo comum na maioria dos trabalhos que envolvem planejamento para jogos RTS. Nessa dissertação de mestrado, será descrita uma abordagem que possibilita a determinação de metas que maximizam a produção de recursos. Determinar uma meta significa estipular os recursos a serem

produzidos juntamente com um plano de ações que gera tais recursos. Essa abordagem é a principal contribuição deste trabalho.

Esta pesquisa teve como objetivo o desenvolvimento de uma abordagem para resolver o problema de determinar metas com qualidade a serem alcançadas durante partidas em jogos RTS. Inspirado em parte pelos resultados obtidos por [Fayard 2005] e nas lacunas presentes nos trabalhos de [Chan et al. 2007], [Chan et al. 2008] e [Branquinho et al. 2011b, Branquinho et al. 2011a], foi elaborada uma abordagem que maximiza a produção de recursos utilizando o algoritmo de busca estocástica *Simulated Annealing* (SA) para que seja encontrada uma meta com qualidade. Além de especificar uma meta, o método proposto retorna ao fim da busca o plano contendo todas as ações e informações necessárias para atingi-la, desenvolvendo assim um planejamento completo. Para o correto funcionamento desta proposta, foram desenvolvidos algoritmos para ajustar, validar e gerenciar a maximização da produção de recursos, que é o mecanismo de especificação das metas. Tais algoritmos são baseados em técnicas de planejamento apoiado em inteligência artificial (IA).

Além de propor uma solução para o problema de escolha de metas, esta pesquisa apresenta uma nova abordagem para a produção de recursos em jogos RTS. Essa utiliza o SA como algoritmo de busca, onde são gerados planos de ações com intuito de maximizar a produção de recursos. Para iniciar o SA é passado um plano inicial de ações obtido por um dos planejadores desenvolvidos no trabalho. Sempre que é preciso gerar um novo plano de ações é usado um dos verificadores de consistência construídos nesta pesquisa. No decorrer do trabalho, sempre que a escolha de uma meta for proposta, significa que será executada uma busca que estipulará uma produção de recursos através de planejamento que maximiza essa produção através de um critério. Tal critério objetiva fortalecer ao máximo a força do exército produzido.

Esse trabalho apresenta relevância para o campo de planejamento em IA, pois lida com diversos desafios e problemas tais como concorrência entre atividades, restrições entre recursos, efeitos numéricos e exigências de tempo real. Além disso, essa abordagem pode ser utilizada por um agente jogador em partidas do jogo ou pode funcionar como interface para auxiliar um jogador humano a escolher metas dentro do jogo. Dessa forma, esse trabalho vai em direção a uma nova abordagem para produção de recursos, podendo ser estendido para outras áreas e aplicações. Para validar os resultados aqui apresentados, foram feitos testes utilizando a abordagem proposta dentro do ambiente de um jogo RTS.

1.1 Contribuição

As principais contribuições deste trabalho são:

1. Adaptação do algoritmo **Simulated Annealing** (SA) através de técnicas que per-

mitem que ele consiga operar em um ambiente de tempo real. Tais técnicas modificam a forma que o algoritmo trabalha, permitindo que ele retorne resultados compatíveis com o domínio de Jogos RTS.

2. Desenvolvimento de um planejador sequencial que constrói um plano de ações inicial para o SA sem utilizar uma meta pré estabelecida.
3. Desenvolvimento de um verificador de consistência sequencial que opera gerenciando e validando as operações do SA sobre os planos gerados. O verificador é responsável por garantir que o SA quando finalizar sua execução, além de estipular uma meta com qualidade, também retornará o plano de ações para atingi-la.
4. Desenvolvimento de um planejador de ordem parcial **POPlan**, que elabora um plano e escalona suas ações ao mesmo tempo. POPlan também não utiliza uma meta pré estabelecida e gera um plano baseado em um tempo limite para a execução de suas ações. Também foi proposto um algoritmo de escalonamento.
5. Desenvolvimento de um verificador de consistência com escalonamento **SHELR-Checker**, que gerencia as mudanças nos planos com ações paralelizadas. SHELR-Checker ao gerenciar as operações do SA garante que o plano mantenha suas ações paralelizadas para atingir a meta que será determinada pelo algoritmo.

1.2 Organização da Dissertação

A dissertação apresenta a seguinte ordem e estrutura de capítulos. No Capítulo 2 são apresentados os principais conceitos e técnicas de planejamento. Já o capítulo 3 detalha os principais aspectos da produção de recursos em Jogos RTS e os trabalhos correlatos que mais auxiliaram no desenvolvimento dessa dissertação. O Capítulo 4 mostra as adaptações e técnicas que foram feitas para o *Simulated Annealing* operar sob as condições e restrições de jogos RTS. Os Capítulos 5 e 6 apresentam respectivamente as arquiteturas sequencial e com escalonamento a serem usadas no SA. Já o Capítulo 7 descreve os experimentos realizados e resultados obtidos para essa dissertação. Por fim, o Capítulo 8 apresenta a conclusão e trabalhos futuros.

Capítulo 2

Fundamentos Teóricos

Neste capítulo serão descritos e discutidos os fundamentos teóricos mais importantes visando uma melhor compreensão desta pesquisa de mestrado. Este capítulo está estruturado da seguinte forma: Na Seção 2.1 são apresentados algoritmos de busca local juntamente com funcionamento e uso dos mesmos. Já na Seção 2.2 são abordadas as principais técnicas e conceitos de planejamento.

2.1 Algoritmos de Busca Local

Os algoritmos de busca local constituem uma das mais importantes classes de algoritmos de busca em IA. Esses algoritmos possuem vantagens sobre algoritmos clássicos de busca como Busca em Largura ou Busca de Primeira Ordem, quando é preciso encontrar soluções para problemas de tempo real com restrições e planejamento. Nesses cenários, o caminho percorrido para encontrar uma solução é irrelevante e devido ao tamanho do espaço de estados (soluções) e as restrições de tempo, retornar uma solução pode ser algo muito dispendioso em tempo e em processamento computacional. Assim, os algoritmos de busca local conseguem retornar soluções de qualidade, sem infringir as restrições presentes no problema e com um menor custo computacional. Algoritmos de busca clássica esbarram em sistemáticas como não retornar uma solução parcial quando as restrições são atingidas e grande consumo de recursos computacionais e tempo durante a busca [Hoos e Stützle 2012].

Algoritmos de busca local em especial os de busca local estocástica, utilizam de mecanismos de escolha aleatória para gerar soluções candidatas em problemas de análise combinatória e otimização. Entre os primeiros algoritmos de sucesso temos o algoritmo de *Lin-Kernighan* [Hart et al. 1968] para o problema do caixeiro viajante, os diversos métodos baseados em algoritmos evolucionários [Back 1996] como algoritmos genéticos [Holland 1975], Algoritmo de Metropolis [Lecheda 2004] e *Simulated Annealing*, sendo o último utilizado neste trabalho [Aarts e Korst 1989].

Na Seção 2.1.1 será descrito o algoritmo genético e um resumo do seu funcionamento,

pois ele é um algoritmo que poderia ser utilizado no problema tratado neste trabalho. Na Seção 2.1.2 o Simulated Annealing é apresentado juntamente com suas principais características e funcionamento.

2.1.1 Algoritmos Genéticos

Algoritmos Genéticos (AG) são modelos computacionais/matemáticos utilizados em problemas de otimização, busca combinatória, classificação de padrões e em outros que necessitem de busca em um espaço de soluções [Lecheda 2004]. Inspirado na maneira como a teoria da evolução das espécies de Darwin foi construída e nas leis de Gregor Mendel, Jonh Holland [Holland 1975] em meados dos anos 70 desenvolveu juntamente com sua equipe de pesquisa um modelo computacional capaz de reproduzir o comportamento evolutivo das espécies em estruturas de dados e modelos matemáticos. Ele dividiu esse modelo nas seguintes etapas: inicialização, avaliação, seleção, cruzamento, mutação, atualização e finalização.

Um AG trabalha basicamente criando uma população de possíveis respostas para o problema e submete essas ao processo de evolução. Durante esse processo, os indivíduos (possíveis soluções) presentes em uma população combinam-se gerando novos indivíduos. Entre esses, são selecionados os que forem considerados mais aptos para resolver o problema. Sendo assim, os demais indivíduos são descartados. Esses melhores indivíduos são novamente submetidos ao processo de evolução até que uma solução seja encontrada ou alguma condição de parada seja invocada.

Durante as últimas décadas, diversas áreas adotaram os algoritmos genéticos como método para resolver os problemas de otimização. Entre essas podemos citar: Simulação de modelos biológicos, codificação de DNA, indução e otimização de bases de regras e até composição musical. Para cada uma dessas áreas o algoritmo sofre algumas alterações para adaptar-se ao problema em questão. A Figura 2.1 mostra a estrutura básica de funcionamento de um AG.

Em seu funcionamento os algoritmos genéticos iniciam suas operações e geram a população de indíviduos que são possíveis soluções. A geração desses normalmente é feita por um mecanismo aleatório que visa gerar indíduos com maior biodiversidade [1] [Michael Bowman e Labiche 2010]. Os indivíduos gerados são submetidos a um processo de avaliação, onde recebem uma nota ou um índice que indica o quão aptos eles são para combinarem-se a outras soluções formando novos indivíduos. A função de avaliação deve ser construída de acordo com o problema que está sendo tratado, fornecendo uma nota adequada a cada um dos indivíduos. Após gerar a população inicial e avaliar seus indivíduos o algoritmo termina sua primeira etapa. Na próxima etapa, funções como *crossover* e *mutação* são executadas. O Algoritmo1 descreve uma representação das funções em um pseudocódigo

[1] Biodiversidade se refere ao termo utilizado em computação bio-inspirada, onde o significado está ligado a geração de dados de entrada distintos que serão usados em um algoritmo.

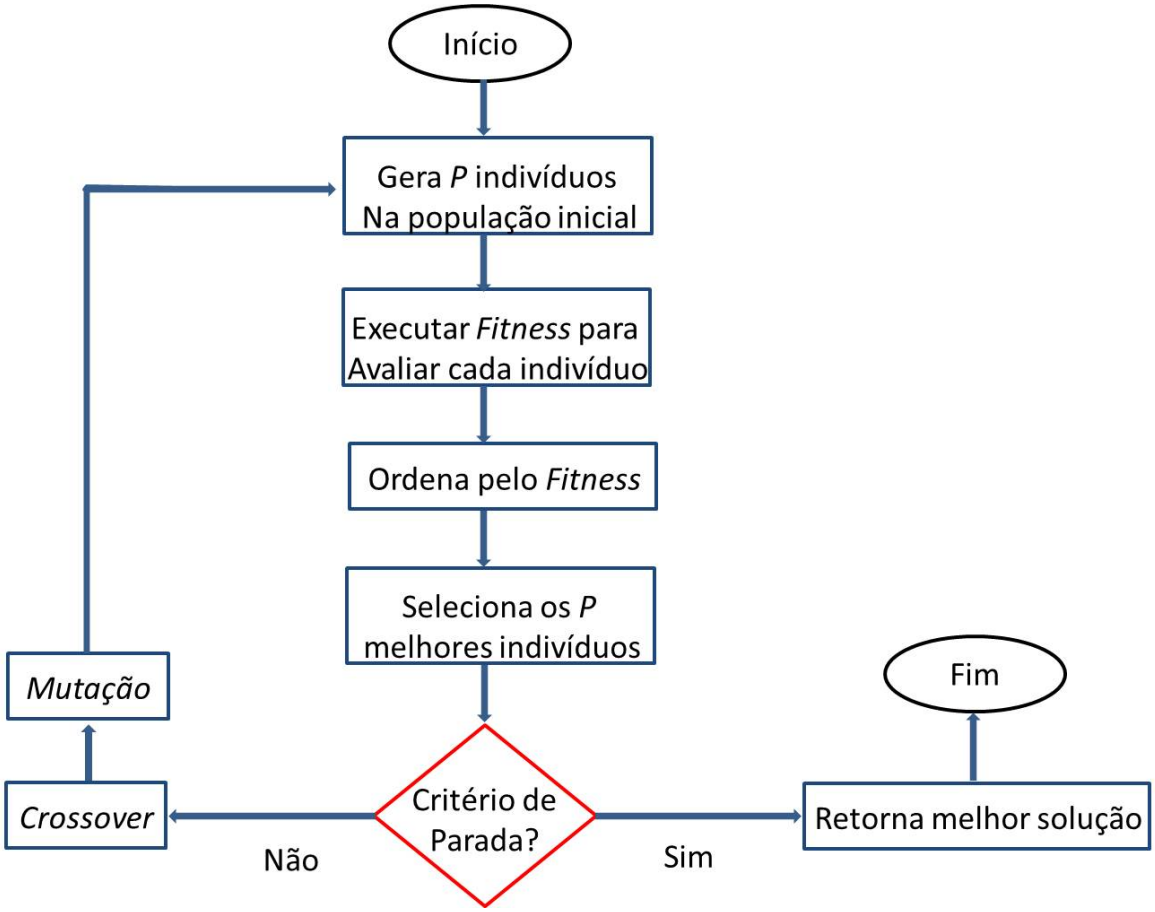

Figura 2.1: Estrutura de funcionamento de um algoritmo genético.

de um AG. Nele a variável t representa o tempo atual do algoritmo, P é a população que será gerada e d é um valor adotado como critério de parada.

Algoritmo 1 Algoritmo Genetico:

1: $d \leftarrow$ Valor como critério de parada
2: $IniciaPopulacao(P, t)$
3: $Avaliacao(P, t)$
4: **while** $t < d$ **do**
5: $t \leftarrow t + 1$
6: $SelecionaReprodutores(P, t)$
7: $CruzaSelecionados(P, t)$
8: $MutaResultantes(P, t)$
9: $AvaliaResultantes(P, t)$
10: $AtualizaPopulacao(P, t)$
11: **end while**

Após a primeira etapa mencionada anteriormente, o algoritmo entra no seu *loop* de controle (linha 4, Algoritmo1) onde pode ser definido um ou mais críterios de parada. No Algoritmo1 é definido apenas um tempo limite para interrupção do processo de busca do AG. A função $SelecionaReprodutores(P, t)$ (linha 6) é responsável por selecionar os indivíduos que melhor foram avaliados pela função $Avaliacao(P, t)$ (linha 2). Esse processo

pode ser feito de diversas maneiras, utilizando *rankings* que classificam em posições os indivíduos, por giro de roleta que aleatoriamente determina uma posição e todos aqueles que estiverem a frente dessa posição são escolhidos e os demais são descartados, ou até mesmo por torneios, onde os indivíduos são divididos em grupos e cooperam para aumentar o valor de avaliação do grupo. O objetivo é selecionar aqueles que podem melhor disseminar suas características durante a reprodução gerando melhores indivíduos, ou seja, mais próximos da solução do problema.

A função $CruzaSelecionados(P, t)$ (linha 7) realiza o processo chamado de *crossover* que gera novas soluções (indivíduos) para o problema. Inicialmente, a função faz o pareamento para determinar quais pares de indivíduos serão combinados para reproduzirem. Esse processo também possui diversas maneiras de ser feito. Entre essas, é posssível combinar um indivíduo com alto índice de avaliação com um outro de baixa avaliação, combinar indivíduos com características semelhantes e até mesmo um indivíduo com ele mesmo. Após a etapa de pareamento é feito o cruzamento utilizando os operadores[2] responsáveis por isso. Esses operadores alteram a representação de cada par de indivíduos combinando suas representações de acordo com alguma modelagem. A representação de um indivíduo é geralmente feita por um vetor binário, onde esse possui o comprimento e os dados que mapeiam uma possível solução para o problema, sendo assim um indivíduo do algoritmo. Existem diversos operadores considerados tradicionais em um AG, contudo podem ser criados novos operadores, se necessário.

A mutação definida pelo método $MutaResultantes(P, t)$ (linha 8) opera sobre os indivíduos restantes do cruzamento com uma probabilidade de efetuar alguma mudança sobre eles. A ideia dessa função é representar as diversidades geradas pela mutação na natureza, assim é feita mais uma combinação entre as soluções nessa etapa. Os dois métodos finais do algoritmo $AvaliaResultantes(P, t)$ (linha 9) e $AtualizaPopulacao(P, t)$ (linha 10) avaliam os novos indivíduos gerados e os inserem na população original P. Com isso, é feito o teste se o algoritmo deve parar ou não, caso continue, o processo de operações se repete e novos indivíduos são gerados.

Algumas das principais características dos algoritmos géneticos são:

- Busca codificada: Um AG não trabalha sobre o domínio do problema e sim sobre uma representação de suas soluções, isso pode ser útil quando é necessário utilizar a técnica em diferentes problemas.
- Busca através de hiperplanos: Através dos operadores de cruzamento é possível guiar a busca de forma que sejam evitados máximos e mínimos locais de um problema [Michael Bowman e Labiche 2010].

[2]Operador em um algoritmo de busca, é responsável por efetuar uma operação ou mudança sobre uma determinada possível solução da busca com o objetivo de modificar essa ou gerar uma nova possível solução

- Paralelismo na implementação: Considerando que cada indivíduo do algoritmo pode ser modelado como um objeto único ou um conjunto de invíduos como uma instância, é possível simplificar e acelerar o processo de construir várias soluções utilizando de paralelismo na execução deste processo.

Dessa maneira, os algoritmos genéticos são uma das princípais técnicas de busca local, tendo sua aplicação estendida a várias áreas e problemas.

2.1.2 Simulated Annealing

O *Simulated Annealing* é uma meta heurística que assim como os Algoritmos Genéticos pertence a classe de algoritmos de busca local [Aarts e Korst 1989]. Ele é um dos algoritmos mais usados na literatura para resolver problemas de otimização combinatória que apresentam restrições entre as soluções [Bandyopadhyay et al. 2008]. O SA elimina algumas desvantagens presentes em algoritmos de busca local. Uma delas é diminuir a chance de cair em ótimos locais devido ao mecanismo de busca que efetua perturbações estocásticas para geração de novas soluções. Outra é a possiblidade de aceitar novas soluções que sejam classificadas como inferiores a atual aumentando a exploração do espaço de soluções [Laarhoven e Aarts 1989]. Isso permite que soluções mais robustas sejam encontradas.

O algoritmo é conhecido como *Simulated Annealing* devido a analogia que é feita com as técnicas de resfriamento de metais e vidros para a estabilização de seus elementos. O funcionamento do algoritmo baseia-se nesses princípios. Contudo, ele também é conhecido como *Monte Carlo Annealing* [Brémaud 1999], *Statistical Cooling* [Russell e Norvig 2003], *Probabilistic Hill Climb* [Wu 2009], *Stochastic Relaxation* [Bouleimen e Lecocq 2003] e *Probabilistic Exchange Algorithm* [Laarhoven e Aarts 1989].

O processo de estabilização de metais é feito colocando esses em altas temperaturas onde suas partículas internas posicionam-se de forma aleatória. Na medida em que a temperatura diminui, essas partículas vão alcançando seu equilíbrio posicionando-se de forma a deixar o metal o mais estável possível. Quando esse ponto é atingido, é dito que o elemento alcançou a sua propriedade de estabilidade. No algoritmo é usado o mesmo princípio, onde uma temperatura T, quando possui altos valores, tende a testar diversas combinações para criar soluções. À medida que T decai, o algoritmo busca por combinações com maior equilíbrio que geram soluções com qualidade. A cada valor de T uma solução pode ter seu equilíbrio avaliado pela probabilidade de estar em um estado de energia E, dado pela distribuição de *Boltzman* (Equação 2.1). Nela, $Z(T)$ é um fator de normalização e depende da temperatura T, ΔE é a diferença de valores entre a solução atual e a nova solução gerada, K_b é a constante de *Boltzman* e o fator $exp\Big(-\frac{\Delta E}{K_b T}\Big)$ é

conhecido como fator *Boltzman*.

$$Pr\{T = E\} = \frac{1}{Z(T)}.exp\left(- \frac{\Delta E}{K_b T} \right) \tag{2.1}$$

Para gerar uma nova solução a partir de uma solução atual o SA utiliza o princípio do equilíbrio térmico como já citado. Dada uma possível solução é feita uma pequena perturbação para gerar uma nova solução. Se a diferença de energia entre a antiga e nova solução dada por ΔE for negativa, significa que a nova solução é mais apta para ser a solução final do problema que está sendo tratado pelo algoritmo e é escolhida como tal. Caso ΔE seja maior que zero existe uma probabilidade de aceitação para a nova solução. Essa probabilidade é dada por $exp\Big(- \frac{\Delta E}{K_b T}\Big)$. Esse mecanismo de geração de soluções é conhecido como método de *Monte Carlo*, e opera junto com a probabilidade de uma solução ser aceita que é conhecida como critério de *Metropolis* [Laarhoven e Aarts 1989]. Utilizando estes mecanismos o sistema eventualmente irá convergir para o seu equilíbrio, ou seja, irá alcançar a solução ótima após diversas gerações feitas pela distribuição de *Boltzman* 2.1.

Em seu funcionamento o *Simulated Annealing* opera sempre com soluções para o problema que está sendo tratado. Cada novo vizinho que o algoritmo gera a partir do estado atual pode ser considerado uma possível solução. A palavra "possível" indica que o estado atual ou um novo estado gerado pelo algoritmo, podem ser admitidos como a solução no final do mesmo. Mas, enquanto as operações do algoritmo ainda estão sendo feitas e ele não terminou sua execução, cada uma dessas soluções é considerada como sendo provável. Em sua inicialização, o SA recebe alguns parâmetros de entrada, entre esses um estado inicial para começar sua série de iterações em busca da melhor solução, esse estado é uma provável solução. Assim, o algoritmo necessita dessa solução inicial para conseguir gerar as próximas. Por definição, o SA explora o espaço de soluções de um dado problema, cada solução pode ser considerada um estado do algoritmo em uma dada temperatura, um estado corresponde a uma possível solução. O Algoritmo2 descreve o funcionamento do SA.

No Algoritmo 2, a temperatura inicial do algoritmo deve ser definida de acordo com algum critério ou baseada em características do problema tratado. Uma vez definida, a temperatura decaí após uma série de repetições onde são geradas soluções vizinhas. A velocidade que a temperatura diminui é um fator importante para o algoritmo. Se o decaimento for muito rápido o algoritmo pode não convergir para uma boa solução, por não explorar o suficiente o espaço de soluções. No caso de decaimento ser muito lento, o SA pode demorar tempo demais para encontrar uma solução o que pode invalidar sua eficiência. Assim, é preciso ponderar a redução da temperatura.

A função objetivo é responsável por avaliar a energia de uma dada solução (estado) gerada pelo SA. Essa avaliação é feita baseada em propriedades que todas as soluções

Algoritmo 2 Simulated Annealing(T_0, T_f, L, μ)

```
1: T ← T_0 (Temperatura atual recebe temperatura inicial)
2: S_0 ← gera solução inicial
3: S ← S_0 (solução atual recebe solução inicial)
4: while T > T_f (enquanto a temperatura atual for maior que a temperatura final do
   algoritmo) do
5:    for cont ← 1 até L (repete até que cont seja igual ao número de iterações) do
6:       S' ← gera um nova solução vizinha a partir de S
7:       ΔE ← Custo(S') − Custo(S)
8:       if ΔE < 0 then
9:          S ← S'
10:      else
11:         numeroAleatorio ← recebe um número aleatório entre 0 e 1
12:         if numeroAleatorio < exp(−ΔE/T) then
13:            S ← S'
14:         end if
15:      end if
16:   end for
17:   T ← T * μ (Reduz a temperatura)
18: end while
19: return S
```

apresentem e que possam avaliar o quanto um estado está próximo de ser a melhor solução para o problema. Por definição, quanto melhor for uma solução mais baixa será a energia atribuída. Assim uma solução ótima teria sua energia avaliada em zero. Sempre que os valores de energia dos estados são subtraídos um pelo o outro é possível saber qual deles é melhor candidato a solução final. Quando o algoritmo opta por um estado inferior ao que está sendo considerado solução atual, o mesmo está fazendo um retrocesso que possibilita aumentar a abrangência da busca local por soluções. A possibilidade de escolha de soluções inferiores diminui a medida que a temperatura decaí.

Exemplo de Aplicação do *Simulated Annealing*

Problemas de planejamento com restrição de recursos e pré-condições entre ações, envolvem o gerenciamento de todos esses durante o processo de busca por soluções. Normalmente, o objetivo é minimizar ou maximizar os resultados que são encontrados e que atendem a todas as restrições impostas pelo problema. Quando o *Simulated Annealing* é usado em problemas como esses, é gerado um estado de cada vez. Cada um desses têm potencial para ser a solução final do problema. Cada estado é gerado respeitando todas as pré-condições e restrições que o problema possui.

A Figura 2.2 mostra um exemplo de um problema de caminhamento em jogos. Nessa representação, o jogador se move pelo mundo do jogo através de portais espalhados nele. Neste, cada portal leva o jogador a um dos portais que está conectado a ele. É permitido ida e volta entre os portais. Isso é possível, desde que a pré-condição de um portal estar

conectado ao outro seja respeitada. Cada portal tem associado a si o seu número, o tempo de duração para levar o jogador a outro portal, além da quantidade de recursos gastos para usá-lo. A quantidade de recursos que o jogador possui disponível para usar neste exemplo é de 300. O objetivo é ir do portal 0 até o 9 gastando o menor tempo possível sem exceder a quantidade de recursos disponíveis, além de respeitar as pré-condições entre os portais.

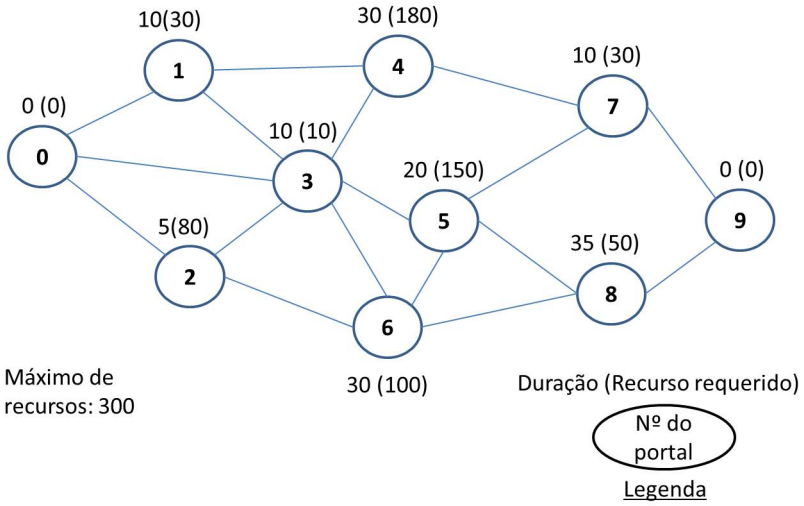

Figura 2.2: Representação de um problema de planejamento com restrições em um jogo.

No início, o algoritmo vai receber uma solução inicial para começar suas operações. Quando uma solução vizinha ou um estado vizinho for criado, será verificado se essa atende todas as restrições do domínio. No caso de atender, o estado é avaliado pelo algoritmo. Caso não seja atendida, a solução é considerada inválida, pois não consiste em um caminho válido sendo assim descartada.

Solução

Cada solução gerada pelo algoritmo é composta por um caminho entre os portais da Figura 2.2. A quantidade de recursos gasta pela solução deve ser menor ou igual à quantidade disponível para o problema. Nesse caso, quando uma transição entre dois portais é feita, a quantidade de recursos gasto para tal ação é descontada na quantidade disponível, não sendo possível desfazer essa transição. Uma possível solução para esse problema poderia ser as seguintes ações: *0,3,5,8,9*. Nessa solução, a ordem de execução das ações, ou seja, a sequência em que os portais são visitados, é a mesma em que as ações estão dispostas no plano.

A solução inicial que é passada para o SA pode ser qualquer ordem aleatória de portais e suas respectivas transições. A ordem em que as ações ficam dispostas na solução é importante. Essa ordem pode indicar qual a sequência de execução das ações. Essa

também determina o tempo de início e término de uma ação e pode ser alterada para gerar um novo vizinho a partir dela.

Cada ação (portal) dentro de uma solução encontra-se em um dos três estados a seguir: *executado*, *não executado* e *executando*. O estado *executado* significa que aquela ação já foi executada, de modo que o seu tempo de execução já atingiu o fim e ela foi completamente feita. O estado *não executado* significa que ela está no plano para ser executada, mas ainda não foi. O seu tempo de início de execução está a frente do tempo atual do plano, e ainda será feita. E por último, o estado *executando* diz que a ação está em execução no tempo atual do plano, isso significa que esse tempo atual está depois do tempo de início da ação e antes do seu tempo de término. Quando um plano é gerado tem-se que a primeira ação está no estado *executando* e as demais ações no estado *não executado*. Quando o plano chegar ao fim, todas as ações devem estar no estado *executado*.

Geração de Novas Soluções

Como já mencionado anteriormente, podemos considerar cada solução que é gerada pelo algoritmo como sendo um estado do mesmo. Assim, o processo de geração de novas soluções pode ser considerado uma transição entre estados. A solução inicial é o primeiro estado e dá início ao algoritmo, todas as demais soluções geradas serão estados que têm uma transição entre o primeiro estado e todos os outros estados gerados.

Cada transição representa alteração nos atributos das ações responsáveis pela transição e na quantidade das mesmas no plano. Se o algoritmo gera um novo estado a partir do estado atual em que ele se encontra, essa geração irá alterar a ordem das ações ou o próprio conjunto de ações, de modo que seus atributos também serão alterados. Por exemplo, o algoritmo tem como solução atual para o problema da Figura 2.2 o seguinte plano de ações *0,2,3,5,7,9* com *makespan* de 65 segundos (seg). O algoritmo então passa para a etapa de gerar um novo vizinho a partir desse plano. Nessa geração, a alteração feita no plano será a inserção da ação *portal 1* no lugar da ação *portal 2*. O novo plano é válido e possui as ações *0,1,3,5,7,9* com *makespan* de 50 segundos. Esse plano agora possui todas as ações que estão a frente da ação *1* com tempo de início e término diferentes do que eram na antiga solução, pois a ação *portal 1* tem atributos como tempo de execução diferente da ação *potal 2*. O caminho entre os portais também foi alterado.

Novas soluções devem ser geradas através de pequenas mudanças no plano de ações que representa a solução atual do SA. Quando um plano sofre diversas operações na geração de uma nova solução, essas tendem a levar o algoritmo para uma parte muito distante do espaço de planos de onde a solução atual estava. E quando isso ocorre, o algoritmo pode deixar de encontrar uma boa ou a melhor solução que estava próxima daquela que foi alterada. Apesar de alguns algoritmos adotarem essa característica para fugir de mínimos locais, o SA usa outro mecanismo para evitar esses mínimos, pois sua busca é local a partir da solução inicial. Não é recomendado utilizar várias operações sobre um plano em

uma única geração, pois em ambientes muito restritivos como o de jogos RTS é preciso fazer um complexo gerenciamento nas mudanças que essas operações causam no plano. Além de essas serem muito dispendiosas, podem deixar o algoritmo sobrecarregado, onde não seria possível retornar uma resposta em tempo real ou até mesmo encontrar uma boa solução.

Avaliando uma Solução

Sempre que uma nova solução é gerada no SA é preciso avaliá-la com intuito de mensurar a sua qualidade em relação ao problema tratado. A função objetivo do algoritmo é responsável por essa tarefa. O valor de uma solução pode ser medido levando em conta características presentes nela. Essas devem ser relevantes para medir o real desempenho da solução frente ao problema. No problema da Figura 2.2 por exemplo, o objetivo é encontrar um caminho com menor *makespan*. Com isso, cada solução que é gerada pelo algoritmo tem seu valor de *makespan* medido, e esse serve para avaliar a qualidade da mesma. Quanto menor o valor do *makespan* melhor é a solução.

Alguns algoritmos de busca adotam o uso de heurísticas para auxiliar na busca por novos estados e na escolha destes como solução para o problema. Essas heurísticas são obtidas a partir de conhecimento prévio do domínio. Também podem utilizar algum método que consegue extrair informações do problema para estimar heurísticas em relação às soluções obtidas. No SA, a função objetivo pode utilizar heurísticas para auxiliar a avaliação de uma solução. É a partir dessa avaliação que ele toma suas decisões. Com isso, é importante que a função objetivo avalie características que realmente tenham impacto sobre o problema e possam medir com clareza a qualidade da solução.

O modo como a geração de novos vizinhos é feita pode variar dependo do problema e do algoritmo utilizado. Como já dito, no SA essa geração é feita através de pequenas alterações na solução atual, gerando um único vizinho da mesma. Essas alterações podem ser feitas alterando a posição das ações, seus tempos de início e fim de execução ou retirando e incluindo ações. Por exemplo, a solução *0,2,6,8,9* possui 70 seg de *makespan* e atende todas as restrições do problema da Figura 2.2. Duas possíveis alterações que poderiam ser feitas nessa solução para gerar um novo vizinho a partir dela seriam: trocar duas ações de posição ou inserir uma nova ação no lugar de uma que já está na solução. Utilizando a primeira sugestão, suponha que as ações *2* e *6* troquem de posição, assim a nova solução seria *0,6,2,8,9*. Essa solução não atende as restrições do problema devido ao caminho inválido que foi gerado, sendo, portanto descartada. Já utilizando a alteração que insere uma nova ação no plano, suponha que a ação de número *3* seja inserida no lugar da ação *2*. Essa alteração gera o vizinho *0,3,6,8,9* que possui 65 seg de *makespan*. O novo vizinho será avaliado pela função objetivo com valor melhor do que o da solução atual, tornando-se a solução corrente do problema.

Exemplo

No exemplo da Figura 2.2 existem dez ações (portais). A ação que mais consome recurso tem o valor de 180 (portal 4) e a ação com maior tempo de execução possui o valor de 35 (portal 8). Como já mencionado, é possível ir e voltar através dos portais. Isso torna o espaço de soluções maior devido à possiblidade de fazer caminhos que passam pelo mesmo portal mais de uma vez. O resultado ótimo para esse problema consiste na solução *0,3,5,7,9* com *makespan* de 40 seg e consumo de 190 em recursos.

Antes de executar o SA, é necessário que uma primeira solução válida seja desenvolvida e passada como entrada para o algoritmo. Essa deve conter o formato esperado para a solução final, ou seja, se a expectativa é uma solução com ações em paralelo, por exemplo, é necessário que a solução inicial tenha suas ações paralelizadas para que o algoritmo opere sobre essas condições. A solução inicial utilizada neste exemplo é: *0,2,6,8,9*. As operações que podem ser feitas sobre o plano são: Inserir uma nova ação no lugar de outra ação do plano, inserir uma nova ação no plano e retirar uma ação do plano.

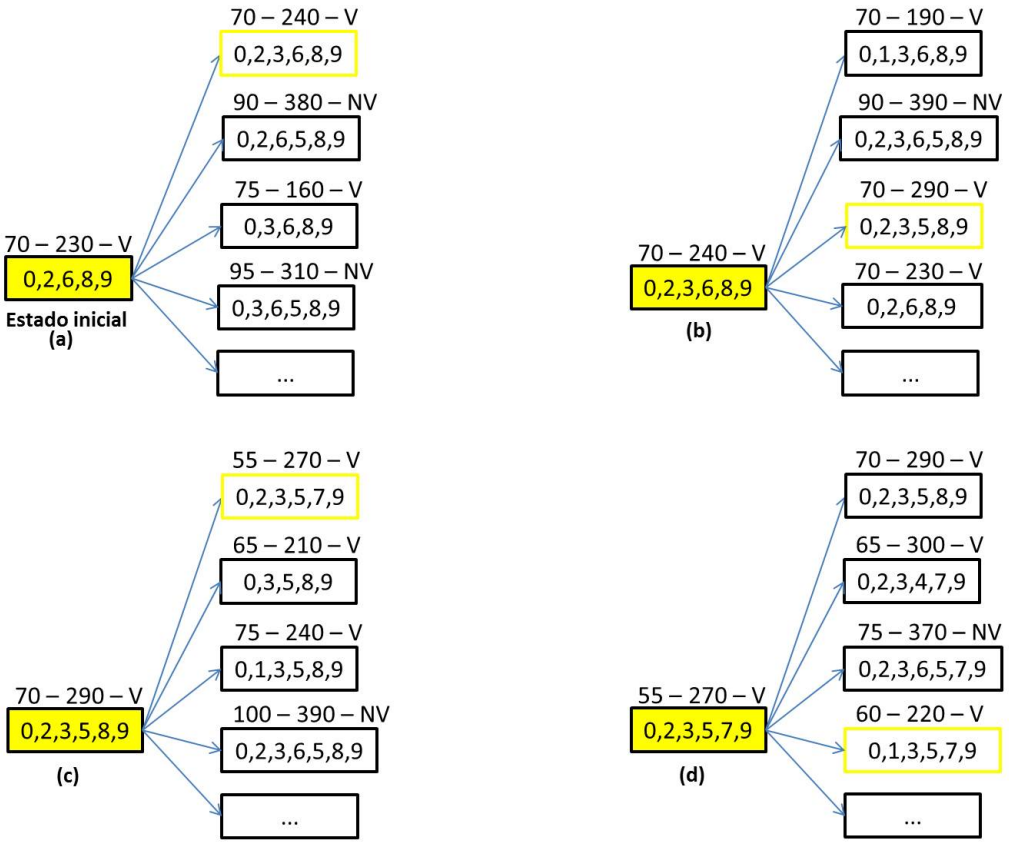

Figura 2.3: Primeiros estados da busca utilizando *Simulated Annealing* no problema da Figura 2.2.

A Figura 2.3 mostra as primeiras soluções geradas a partir da solução inicial. Acima de cada solução estão os seus respectivos valores de *makespan*, quantidade de recursos gastos e as legendas V que significa que a solução é válida e NV que significa solução não válida.

Uma solução é inválida quando ela não satisfaz alguma das restrições do problema ou não cumpre todas as pré-condições de alguma das ações que a compõe. Soluções não viáveis quando geradas não são aceitas pelo algoritmo que imediatamente gera outra nova solução vizinha. Algumas das possíveis soluções que podem ser geradas a partir da solução atual estão ligadas através de setas. A última representação composta por reticências representa todas as outras possíveis soluções que podem ser alcançadas a partir da solução atual, incluindo tanto soluções viáveis quanto não viáveis.

O algoritmo durante seu processo de busca pode optar por estados que sejam inferiores ao estado atual de acordo com a função objetivo. Para isso é utilizado o fator *Boltzman* $exp\Big(-\frac{\Delta E}{K_b T}\Big)$, levando em conta a temperatura em que o algoritmo se encontra. Se as duas soluções possuem o mesmo valor de função objetivo é utilizado esse mesmo critério.

De acordo com a Figura 2.3, o algoritmo em sua primeira iteração (a) opta por um estado que possui mesmo valor de avaliação que ele. É importante notar que os estados que aparecem nas iterações da Figura 2.3 são aqueles que podem ser gerados levando em conta as operações disponíveis e o estado atual do algoritmo. Contudo, o algoritmo gera apenas um estado em cada iteração, sendo que na iteração (a) o estado gerado foi aceito.

Na iteração (b) novamente uma solução com mesmo valor de função objetivo que a solução atual é gerada e aceita. Na iteração (a) foi usada a operação de inserção de uma nova ação no plano, que inseriu a ação *3* logo após a ação *2*. Na iteração (b) a operação utilizada foi a de inserir uma ação no lugar de outra no plano, nessa a ação *6* deu lugar a ação *5*. Nas primeiras iterações a temperatura do algoritmo ainda é alta, assim a chance dessas soluções serem rejeitadas é pequena. Já na iteração (c), dentre as possibilidades de novos vizinhos, é gerado um que possui melhor valor de função objetivo que o atual. A operação utilizada nessa iteração foi novamente a de inserção de uma nova ação no lugar de outra do plano. A ação *7* foi substituída pela ação *8*. Dentre as possibilidades de estados a serem gerados nessa iteração está o estado *0,3,5,8,9*, cuja a geração é feita usando a operação de retirar uma ação do plano. Nesse a ação *2* foi retirada. A operação utilizada em cada iteração é escolhida de forma aleatória dentre as disponíveis.

Em sua iteração (d) o algoritmo volta a optar por uma solução vizinha inferior a solução atual. A solução gerada através da inserção da ação *1* no lugar da ação *2* está próxima da melhor solução do problema que constitui-se do plano *0,3,5,7,9*. A solução antes da última operação também estava próxima do ótimo. Com isso, o algoritmo está perto de encontrar a solução final do problema. Devido à forma sutil como as operações alteram o plano, a solução ótima será encontrada nas próximas iterações. A Figura 2.4 mostra a execução completa do *Simulated Anneling* para o problema de caminhamento de portais.

Na Figura 2.4 os estados que estão em amarelo são soluções que foram geradas e escolhidas pelo algoritmo durante o processo de busca. Na figura não foram feitas ligações que demonstram que um estado pode retroceder a outro estado já gerado anteriormente.

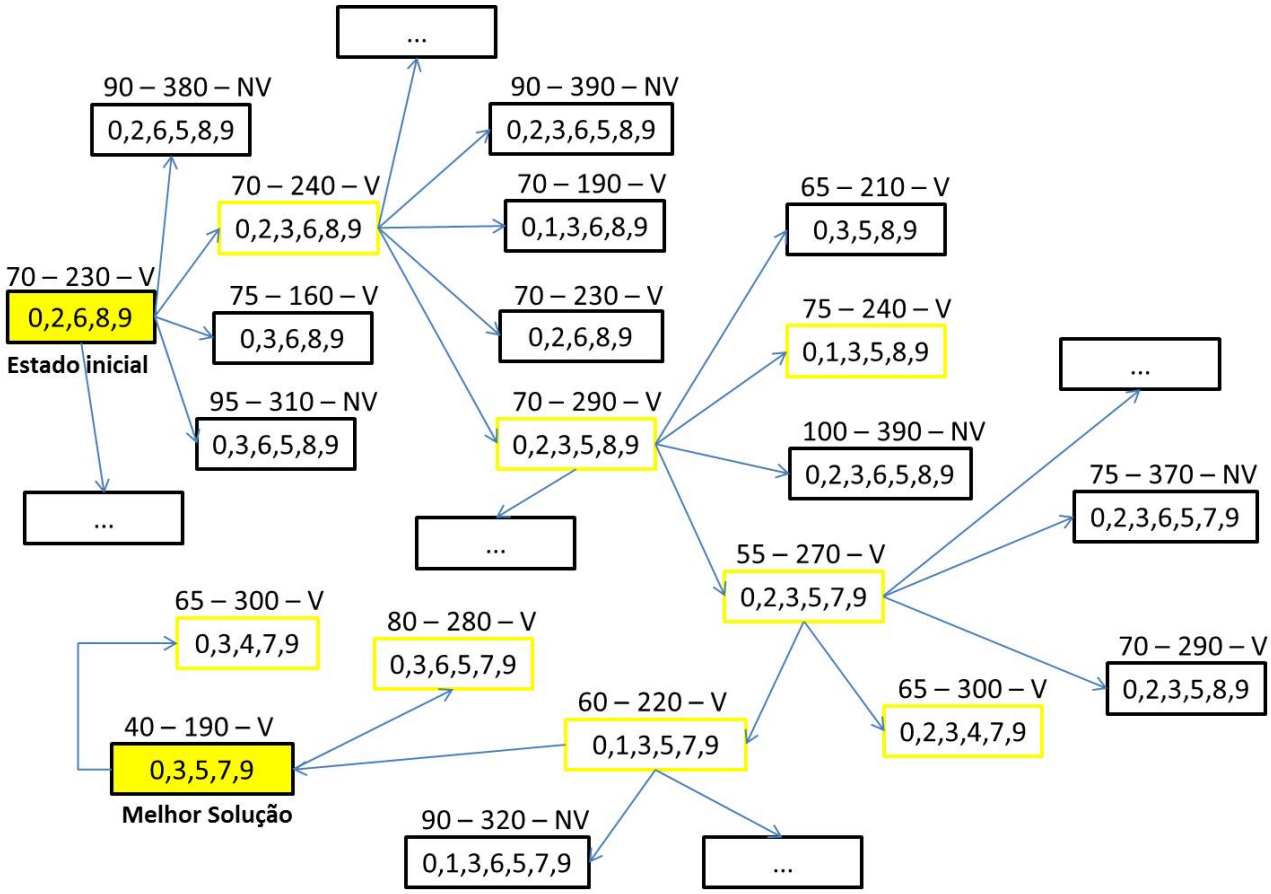

Figura 2.4: Resultado completo da execução do *Simulated Annealing* no problema da Figura 2.2.

Assim, alguns estados possuem mais de uma solução marcada de amarelo entre as possíveis soluções que podem ser geradas, o que significa que o algoritmo gerou e selecionou essa mais de uma vez.

Foram necessárias 15 iterações do algoritmo para que a melhor solução fosse encontrada. A cada iteração a temperatura do algoritmo diminui e são feitas duas gerações (repetições) de novos vizinhos. De fato, o algoritmo encontrou a solução ótima após 7 iterações, mas devido ao fato dele ainda encontrar-se em uma temperatura um pouco elevada nesse momento, novas soluções geradas foram aceitas no lugar dessa. Contudo, o algoritmo após algumas iterações alcançou a solução ótima novamente. A partir desse momento, as novas soluções que foram geradas até o término do algoritmo foram recusadas, devido ao valor da temperatura que estava baixo e não permitiu que soluções inferiores fossem aceitas.

2.2 Planejamento

Planejamento em I.A, incluindo planejamento para jogos RTS, envolve determinar um conjunto ordenado de ações. Essas, quando executadas seja por um ou mais agentes, devem partir de um estado inicial consistente com as condições do domínio resultando num estado final que satisfaça uma dada meta [Russell e Norvig 2003]. Nesta Seção, são apresentados os principais conceitos de planejamento e as abordagens necessárias para o entendimento de como é possível obter uma solução para o problema aqui tratado.

2.2.1 Conceitos Relacionados

O problema de planejamento clássico é definido por um estado inicial do domínio em questão, uma meta a ser atingida e uma descrição das ações que podem ser usadas. Estes três parâmetros devem ser definidos em uma linguagem formal que possibilite a representação do domínio do problema [Weld 1999]. A solução para este problema consiste em gerar automaticamente um conjunto de ações que quando executadas satisfaça a meta. O problema de planejamento determinístico [3] é dado formalmente pela Definição 2.1.

Definição 2.1. O problema de planejamento determinístico é definido formalmente como uma 5-upla $\langle S, i, A, g, G\rangle$, onde:

1. S é um conjunto finito de estados.
2. $i \in S$ é o estado inicial.
3. A é um conjunto finito de ações.
4. $g \in S$ é um estado meta.
5. G representa a solução do problema (normalmente chamado **plano**). O plano é formado por um conjunto ordenado e finito de ações.

O principal elemento para o sucesso no planejamento é a linguagem de representação. A linguagem deve possuir expressividade o bastante para descrever o domínio do problema. Entretanto, também é necessário que seja restrita o bastante para que permita criar algoritmos eficientes. Entre as linguagens mais tradicionais de planejamento temos *STanford Research Institute Problem Solver* (STRIPS), *Action Description Language* (ADL) [Pednault 1989] e *Planning Domain Definition Language* (PDDL) [Edelkamp e Hoffmann 2004].

A linguagem STRIPS baseia-se na lógica matemática para descrever estados e ações. Um estado é representado como uma conjunção de literais positivos. Um estado $s \in S$ pode ser representado com literais proposicionais. Por exemplo, $Alto \wedge Famoso$ poderia representar o estado de uma pessoa alta e famosa. Outra forma de representação de

[3] No momento que o plano é executado o resultado final atingido é exatamente o desejado. O ambiente é estático e as ações têm exatamente o comportamento esperado (Planejamento Clássico).

estados são os literais de primeira ordem: *Indo(Casa*$_1$*, Brasil)* $\wedge$ *Em(Veículo*$_2$*, México)* poderia representar um estado em um problema de caminhamento. Este problema consiste em chegar a um destino dado uma localidade e um meio de transporte para fazer isso.

Uma característica importante considerada durante o planejamento é **a hipótese de um mundo fechado**. Isso significa que quaisquer condições não mencionadas em um estado são consideradas falsas [Russell e Norvig 2003]. O estado inicial i e a meta g também são representados por conjunções de literais. Já as ações em A são especificadas em termos de pré-condições que devem ser satisfeitas antes da ação ser executada e pelos efeitos resultantes da sua execução. Por exemplo, a ação de deslocar em um automóvel de uma localidade para outra pode ser descrita da seguinte forma:

Ação(`Deslocar`$(t, ir, para)$
`pré-condição`: $Em(t, ir) \wedge Automóvel(t) \wedge Estrada(ir) \wedge Estrada(para)$
`efeito`: $\neg Em(t, ir) \wedge Em(t, para)$)

Esta descrição representa várias possibilidades de ações que podem ser derivadas e representadas pela instanciação das variáveis t, ir e $para$. Alguns sistemas de planejamento e planejadores separam os efeitos em uma **lista de adição** para literais positivos e uma **lista de eliminação** para literais negativos.

Na Figura 2.5 é apresentado um problema de planejamento utilizando o planejador STRIPS chamado "*Mundo dos Blocos*". O problema consiste em deslocar os blocos usando as ações `Deslocar` e `DeslocarParaMesa`. Nesse, o objetivo é alterar o *Estado Inicial* do ambiente para que seja obtido o *Estado Meta*. No "*Estado Inicial*", o bloco C está sobre o bloco A e o bloco B está sozinho. A meta é alcançar a configuração onde o bloco A está sobre o bloco B e o bloco B está sobre o bloco C.

A ação `Deslocar`(b, t, z) move um bloco b que está sobre t para z, se ambos b e z estão livres. Um bloco livre significa que não existe outro bloco sobre ele. Depois que o movimento é feito, t fica livre mas z não. Quando $t = mesa$ a ação `Deslocar`(b, t, z) tem o efeito $Livre(mesa)$, mas a mesa não ficará livre (sem blocos). Entretanto, como citado em [Russell e Norvig 2003], consideramos que $Livre(mesa)$ é verdadeira pois supomos que $Livre(mesa)$ significa que existe espaço livre para colocar blocos sobre a mesa. A ação `DeslocarParaMesa`(b, t) move um bloco b que está sobre t para a mesa.

Para evitar algumas redundâncias durante a geração do plano, algumas pré-condições são adicionadas as ações ($b \neq t$; $b \neq z$; $t \neq z$; $Bloco(b)$; $Bloco(t)$; $Bloco(z)$). As pré-condições $Bloco(b)$, $Bloco(t)$ e $Bloco(z)$, são responsáveis por exigir que cada elemento dado como entrada seja um bloco. Já as pré-condições $b \neq t$, $b \neq z$ e $t \neq z$, determinam que os elementos sejam diferentes. Com estas estratégias evitamos que sejam consideradas algumas ações redundantes. Por exemplo, a ação `DeslocarParaMesa`$(A, mesa)$ é redundante pois não altera o estado do mundo.

Estado Inicial($Sobre(A, Mesa) \wedge Sobre(B, Mesa) \wedge Sobre(C, A) \wedge$
$Bloco(A) \wedge Bloco(B) \wedge Bloco(C) \wedge Livre(B) \wedge Livre(C) \wedge Livre(mesa)$)
Estado Meta($Sobre(A, B) \wedge Sobre(B, C) \wedge Sobre(C, mesa)$)
Ação(`Deslocar`(b, x, y),
`pré-condição`: $Sobre(b, x) \wedge Livre(b) \wedge Livre(y) \wedge Bloco(b) \wedge Bloco(y) \wedge$
$(b \neq x) \wedge (b \neq y) \wedge (x \neq y)$
`efeito`: $Sobre(b, y) \wedge Livre(x) \wedge\neg\ Sobre(b, x) \wedge\neg\ Livre(y)$)
Ação(`DeslocarParaMesa`(b, x),
`pré-condição`: $Sobre(b, x) \wedge Livre(b) \wedge Bloco(b) \wedge Bloco(x) \wedge (b \neq x)$
`efeito`: $Sobre(b, mesa) \wedge Livre(x) \wedge\neg\ Sobre(b, x)$)

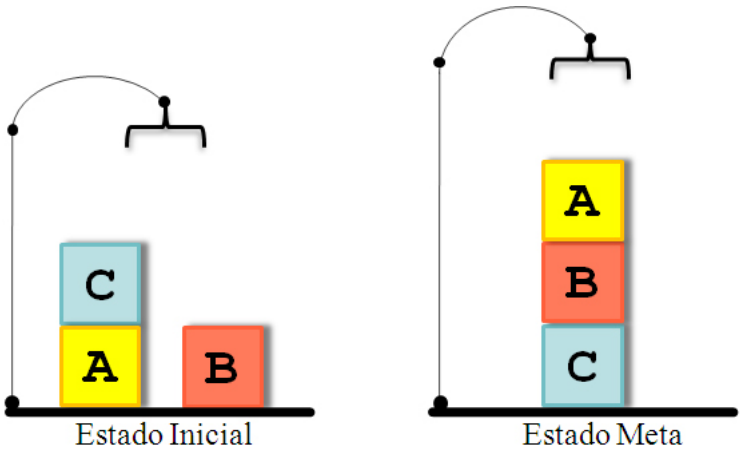

Figura 2.5: Problema "*Mundo dos Blocos*" representado pela linguagem STRIPS.

Existem diversas soluções para o problema da Figura 2.5. A seguir são apresentados dois planos que resolvem este problema [4]. Em cada estado os literais em negrito destacam as pré-condições da ação que será executada. Os literais negados são eliminados do estado resultante da ação. Ao final são sublinhados os literais que fazem parte da meta.

Plano 1: `DeslocarParaMesa`$(C,\ A) \prec$ `Deslocar`$(B,\ mesa,\ C) \prec$ `Deslocar`$(A,\ mesa,\ B)$

- **Estado Inicial**: $Sobre(A, Mesa) \wedge Sobre(B, Mesa) \wedge \boldsymbol{Sobre}(C, A) \wedge \boldsymbol{Bloco}(A) \wedge Bloco(B)$ $\wedge\ \boldsymbol{Bloco}(C) \wedge Livre(B) \wedge \boldsymbol{Livre}(C) \wedge Livre(mesa) \Longleftarrow$ `DeslocarParaMesa`(C,A)
- $Sobre(A, Mesa) \wedge \boldsymbol{Sobre}(B, Mesa) \wedge Sobre(C, Mesa) \wedge Bloco(A) \wedge \boldsymbol{Bloco}(B) \wedge \boldsymbol{Bloco}(C)$ $\wedge\ Livre(A) \wedge \boldsymbol{Livre}(B) \wedge \boldsymbol{Livre}(C) \wedge Livre(mesa) \Longleftarrow$ `Deslocar`$(B,mesa,C)$
- $Sobre(B, C) \wedge \boldsymbol{Sobre}(A, Mesa) \wedge Sobre(C, Mesa) \wedge \boldsymbol{Bloco}(A) \wedge \boldsymbol{Bloco}(B) \wedge Bloco(C) \wedge$ $\boldsymbol{Livre}(A) \wedge \boldsymbol{Livre}(B) \wedge Livre(mesa) \Longleftarrow$ `Deslocar`$(A,mesa,B)$
- **Estado Meta**: $\underline{Sobre(A, B)} \wedge \underline{Sobre(B, C)} \wedge \underline{Sobre(C, Mesa)} \wedge Bloco(A) \wedge Bloco(B) \wedge Bloco(C)$ $\wedge\ Livre(A) \wedge Livre(mesa)$

Plano 2: `DeslocarParaMesa`$(C,\ A) \prec$ `Deslocar`$(A,\ mesa,\ B) \prec$ `DeslocarParaMesa`$(A,$ $B) \prec$ `Deslocar`$(B,\ mesa,\ C) \prec$ `Deslocar`$(A,\ mesa,\ B)$

- **Estado Inicial**: $Sobre(A, Mesa) \wedge Sobre(B, Mesa) \wedge \boldsymbol{Sobre}(C, A) \wedge \boldsymbol{Bloco}(A) \wedge Bloco(B)$ $\wedge\ \boldsymbol{Bloco}(C) \wedge Livre(B) \wedge \boldsymbol{Livre}(C) \wedge Livre(mesa) \Longleftarrow$ `DeslocarParaMesa`(C,A)

[4] $a_1 \prec a_2$ significa que a ação $a_1 \in A$ deve ser executada antes de $a_2 \in A$. $s \Longleftarrow a_3$ significa que sobre o estado $s \in S$ será executada a ação $a_3 \in A$.

- $\boldsymbol{Sobre}(A, Mesa) \wedge Sobre(B, Mesa) \wedge Sobre(C, Mesa) \wedge \boldsymbol{Bloco}(A) \wedge \boldsymbol{Bloco}(B) \wedge Bloco(C) \wedge \boldsymbol{Livre}(A) \wedge \boldsymbol{Livre}(B) \wedge Livre(C) \wedge Livre(mesa) \Longleftarrow$ `Deslocar`$(A,mesa,B)$
- $\boldsymbol{Sobre}(A, B) \wedge Sobre(B, Mesa) \wedge Sobre(C, Mesa) \wedge \boldsymbol{Bloco}(A) \wedge \boldsymbol{Bloco}(B) \wedge Bloco(C) \wedge \boldsymbol{Livre}(A) \wedge Livre(C) \wedge Livre(mesa) \Longleftarrow$ `DeslocarParaMesa`(A,B)
- $Sobre(A, Mesa) \wedge \boldsymbol{Sobre}(B, Mesa) \wedge Sobre(C, Mesa) \wedge Bloco(A) \wedge \boldsymbol{Bloco}(B) \wedge \boldsymbol{Bloco}(C) \wedge Livre(A) \wedge \boldsymbol{Livre}(B) \wedge \boldsymbol{Livre}(C) \wedge Livre(mesa) \Longleftarrow$ `Deslocar`$(B,mesa,C)$
- $Sobre(B, C) \wedge \boldsymbol{Sobre}(A, Mesa) \wedge Sobre(C, Mesa) \wedge \boldsymbol{Bloco}(A) \wedge \boldsymbol{Bloco}(B) \wedge Bloco(C) \wedge \boldsymbol{Livre}(A) \wedge \boldsymbol{Livre}(B) \wedge Livre(mesa) \Longleftarrow$ `Deslocar`$(A,mesa,B)$
- **Estado Meta**: $\underline{Sobre(A, B)} \wedge \underline{Sobre(B, C)} \wedge \underline{Sobre(C, Mesa)} \wedge Bloco(A) \wedge Bloco(B) \wedge Bloco(C) \wedge Livre(A) \wedge Livre(mesa)$

No "Plano 1" podemos ver que no último estado foi obtido exatamente a meta especificada, ou seja: $Sobre(A, B) \wedge Sobre(B, C) \wedge Sobre(C, Mesa)$. O "Plano 2" não é a melhor solução para esse problema, já que necessita de mais ações e mais etapas de planejamento gerando um custo computacional maior. Contudo, ele ainda representa uma solução para problema do Mundo dos Blocos.

Apesar de o planejador STRIPS possibilitar a especificação de vários problemas reais, ele ainda é insuficiente para representar alguns tipos de problemas. Por isso, foram desenvolvidos novos modelos de representação, tais como ADL [Pednault 1989] e PDDL [Edelkamp e Hoffmann 2004]. Uma comparação entre STRIPS e ADL, com as diferenças entre ambas as linguagens, é apresentada em [Russell e Norvig 2003]. A linguagem PDDL inclui STRIPS e ADL, além de possibilitar a definição de tempo de duração para as ações e descrever os efeitos do tempo sobre as mesmas. Além destas modificações, inclui o tratamento de expressões numéricas e permite especificar uma ou mais funções de otimização para o plano (denominada métrica). Essas devem ser descritas na própria representação do problema.

Dependendo de como é definido um problema, podemos executar uma ou mais ações ao mesmo tempo e a relação de pré-condições e dependência pode ser muito forte entre elas. Além disso, existem vários ordenamentos de ações que resolvem um problema. Devido a todas essas propriedades e aos inúmeros problemas, existem vários algoritmos e linguagens de planejamento. Normalmente, planejamento é divido em duas frentes conhecidas como "Planejamento Clássico" e "Planejamento não-Clássico".

Planejamento Clássico: É um planejamento onde o estado resultante após a execução das ações ou de um plano de ações pode ser previsto completamente, pois o estado atual é modificado apenas pelas ações do plano. Nenhuma outra modificação imprevista é inserida durante a execução do plano, ou seja, uma visão determinística.

Planejamento não Clássico: Envolve planos com múltiplos caminhos diferentes de execução e relação de dependência entre as ações. Com isto, os planos não são especificados como uma sequência de ações, mas com políticas que mapeiam estados em ações e condições de execução.

Existem diversos algoritmos de planejamento. O *Graphplan* é um planejamento baseado em grafo e foi proposto por [Blum e Furst 1997]. Alguns métodos consistem no uso da lógica proposicional através de axiomas para dedução dos planos, tendo destaque os trabalhos de [Kautz e Selman 1992], [Kautz e Selman 1996] e [Kautz e Walser 1999]. O UCPOP apresentado em [Penberthy e Weld 1992] é um planejamento que permite o uso de operadores com pré-condições e efeitos. Em [Anderson et al. 1998] são apresentadas melhorias do UCPOP (*Sensory Graphplan* - SGP). Refererências mais recentes como [Mohan Sridharana 2010], [Ras e Skowron 2010] e [Bonet e Geffner 2011] trazem novas abordagens para a área de planejamento. Além destes trabalhos, existem diversos estudos correlatos comparando os métodos de planejamento e propondo melhorias. Nas próximas seções serão abordados alguns métodos de planejamento.

2.2.2 Planejamento com Busca

O algoritmo de planejamento mais simples gera todos os estados e transições entre estados de um grafo. Em seguida, encontra um caminho a partir do estado inicial que conduz até uma meta $g \in G$. Durante essa busca, poderia ser usado, por exemplo, um algoritmo de busca do menor caminho. O plano é uma simples sequência de operadores que correspondem às arestas do grafo durante a execução do algoritmo de busca pelo menor caminho. Apesar de ser uma solução "simples" este procedimento não é viável quando o número de variáveis começa a crescer, já que serão criados muitos estados. Em um problema com 20 ou 30 variáveis de valor lógico o número de estados é muito grande. Por exemplo, para 20 variáveis temos $2^{20} = 1.048.576$ estados e com 30 variáveis $2^{30} = 1.073.741.824$ estados.

Em vez disso, muitas vezes é mais eficiente evitar gerar a maior parte dos estados de forma explícita e produzir somente o estado sucessor ou predecessor do estado atual que está sendo considerado. Este tipo de busca pode ser vista como uma aplicação mais simples dos algoritmos de busca no uso geral, onde esses são empregados na resolução de uma vasta gama de problemas de busca. Os mais conhecidos são algoritmos de busca heurística tais como A*, IDA* e suas variantes [Hart et al. 1968, Pearl 1984, Korf 1985]. Estes algoritmos podem ser usados para encontrar mais rapidamente a melhor solução ou soluções aproximadas do ótimo em algoritmos de planejamento.

Há duas possibilidades principais para encontrar um caminho a partir do estado inicial até um estado meta: atravessar a transição do grafo a partir do estado inicial até a meta (Busca Progressiva - *Forward Search*), ou o contrário, iniciar a busca a partir de uma meta até o estado inicial (Busca Regressiva - *Backward Search*). A principal diferença entre estas possibilidades é que pode haver vários estados meta em um dado planejamento, mas apenas um estado inicial. A busca regressiva é um pouco mais complicada de implementar, mas permite considerar simultaneamente vários caminhos que conduzem a uma meta

[Rintanen 2005]. Para o problema de planejamento ambos os métodos de busca podem ser aplicados.

Planejamento utilizando Busca Progressiva

A Busca Progressiva começa no estado inicial do problema, selecionando e trabalhando com sequências de ações até encontrar uma sequência que atinja um dado estado meta. O estado inicial da busca é o estado inicial do problema de planejamento, ou seja, o estado nativo em que o mundo se encontra quando o planejamento vai começar. Em geral, cada estado será um conjunto de literais positivos (literais que não aparecerem podem ser considerados como falsos).

As ações que podem ser aplicadas a um estado são todas aquelas cujas pré-condições são satisfeitas. O estado sucessor resultante de uma ação é gerado pela adição de literais de efeito positivo e pela eliminação dos literais de efeito negativo, juntamente com os literais já presentes no estado. O teste que verifica se uma meta foi encontrada consiste em determinar se o estado satisfaz a meta do problema de planejamento. A Figura 2.6 apresenta o primeiro passo da busca progressiva sobre o problema da Figura 2.5.

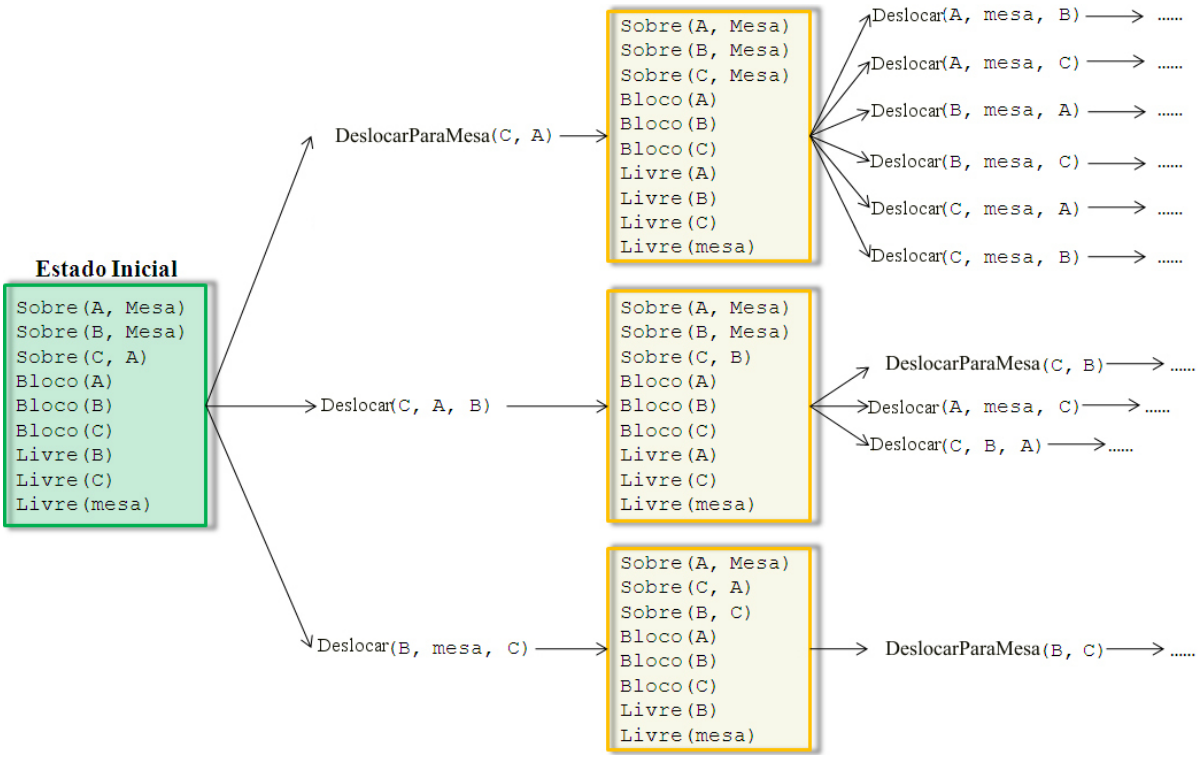

Figura 2.6: Primeiro passo de uma busca progressiva para o problema da Figura 2.5.

O espaço de estados de um problema de planejamento é finito. Portanto, qualquer algoritmo de busca em grafos que tenha a propriedade de completeza [5] será um algoritmo

[5]Completeza, completude ou completo é a propriedade atribuída a algoritmos de busca em grafos quando esses tem a garantia de encontrar uma solução caso ela exista dentro do grafo que representa o domínio do problema de busca em questão.

de planejamento completo. Algoritmos de busca normalmente utilizam algum um valor de heurística. Por exemplo, o A* é um algoritmo que utiliza essa prática, já que utiliza um valor de heurística que otimiza a busca. O problema da busca progressiva é que ela considerada todas as ações aplicáveis, incluindo aquelas irrelevantes. Logo, uma boa abordagem fracassa sem uma boa heurística.

No exemplo da Figura 2.6, a ação `Desloca`(C, A, B) tem como efeito a negação do literal $Livre(B)$. O estado sucessor não considera este literal e adiciona todos os efeitos da ação `Deslocar`(C, A, B). Ao final da busca, a solução deste problema de planejamento é composta pelas ações que fazem parte do caminho entre o estado inicial até a meta. Este grupo de ações corresponde ao plano, ou seja, pela Definição 2.1 é igual a P.

Planejamento utilizando Busca Regressiva

A Busca Regressiva aplicada sobre o planejamento considera inicialmente os literais do estado meta, e posteriormente inicia uma busca até encontrar um estado que possua todos os literais do estado inicial. A principal vantagem da busca regressiva é que ela permite considerar apenas ações relevantes. Uma ação é relevante para uma meta se alcança (tem como efeito) pelo menos um dos elementos da meta. Por exemplo, na Figura 2.7 é exibido o primeiro passo da busca regressiva sobre o problema da Figura 2.5. Nesse, são consideradas apenas aquelas ações que possuem algum efeito direto sobre a meta, ou seja, que satisfazem algum literal da meta.

Entretanto, existem muitas ações irrelevantes que também podem levar a um estado meta. Uma busca regressiva que permite que ações irrelevantes sejam consideradas ainda será completa, mas será menos eficiente. Se existir uma solução ela será encontrada e terá apenas ações relevantes. Com isso, fica claro que considerar ações que não têm efeito direto sobre a meta não é útil para o planejamento. A restrição a ações relevantes significa que a busca para trás normalmente tem um fator de ramificação muito mais baixo que a busca progressiva.

A principal questão da busca regressiva é: "Quais são as ações que devo considerar e representam estados dos quais levarão até o estado meta". Um novo estado na busca consiste nas pré-condições de uma dada ação e os literais que não foram satisfeitos por ela. Além de inserir apenas ações que alcancem algum literal desejado, nenhum efeito da ação pode negar parte da meta. Se isso ocorrer à ação não faz parte da busca naquele momento. As ações que satisfazem essa restrição são chamadas consistentes. Basicamente, um estado predecessor de uma ação $x \in X$ que satisfaz uma meta $m \in M$ é igual a:

- Todos os literais de m que não são satisfeitos pelo efeito de x.
- Cada literal da pré-condição de x sem considerar os literais já existentes.

Na Figura 2.7 o estado predecessor da ação `Deslocar`$(A, mesa, B)$ é formado por:

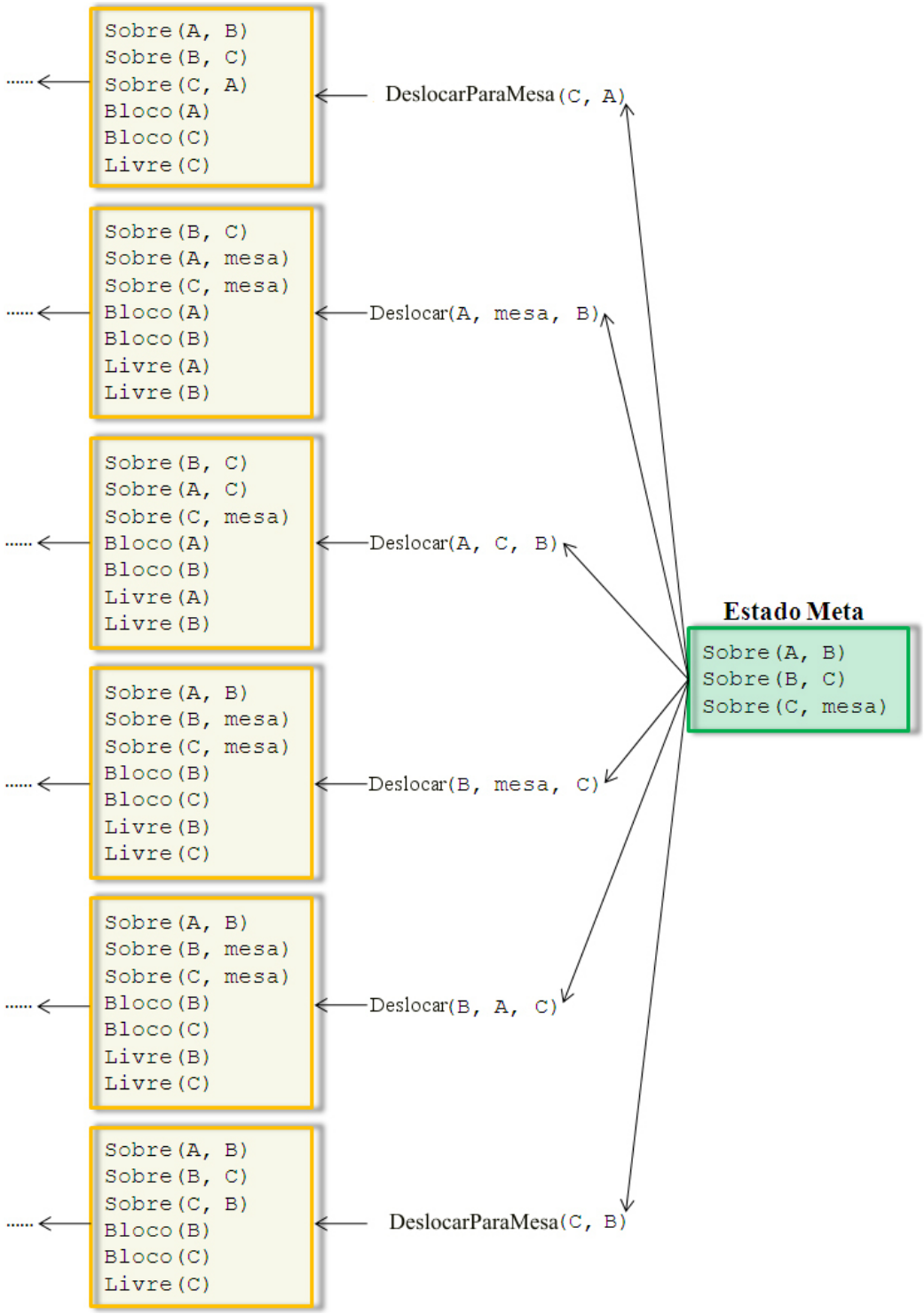

Figura 2.7: Primeiro passo da busca regressiva para o problema da Figura 2.5.

- Literais não satisfeitos pelo efeito da ação: $Sobre(B, C) \wedge Sobre(C, mesa)$.
- Pré-condições da ação: $Sobre(A, mesa) \wedge Bloco(A) \wedge Bloco(B) \wedge Livre(A) \wedge Livre(B)$.

O término acontece quando é encontrado um estado que satisfaz o estado inicial do problema de planejamento. Em outras palavras, um estado da busca que possui todos os literais do estado inicial que nesse caso é a própria meta.

2.2.3 Planejador STRIPS

É importante distinguir entre o planejador original STRIPS, e a linguagem de representação STRIPS. Devido ao seu poder de representação, a linguagem STRIPS desenvolvida para descrever operadores e modelos de mundos vem sendo usada com pequenas modificações em um grande número de planejadores.

STRIPS (*STanford Research Institute Problem Solver*) é um programa de planejamento desenvolvido por volta de 1970 no *SRI International*, formalmente conhecido como *Stanford Research Institute*. Os autores o consideraram um "solucionador de problemas". Atualmente esse tipo de sistema é chamado de "planejador", sendo uma terminologia mais apropriada no campo da IA [Fikes e Nilsson 1971].

Neste contexto, "resolver problemas" significa encontrar uma sequência de operadores no espaço de busca do mundo que podem e irão transformar o estado inicial em outro, onde uma determinada fórmula (meta) pode ser provada como verdade. Duas das principais técnicas que são utilizadas na resolução de problemas. São elas além do próprio STRIPS, o *Means-end Analysis* (MEA) para a busca da meta e resolução de teorema que responde a perguntas sobre um determinado modelo de mundo. A técnica MEA consiste em uma estratégia de busca para solução de problemas e foi introduzido em *General Problem Solver* (GSP) [Newell e Simon 1995]. GPS trata de um ambiente com objetos que são transformados através de operadores. Esses detectam a diferença sobre o ambiente e as metas, tendo o objetivo de diminuir tal diferença.

STRIPS utiliza operadores que descrevem de forma adequada o problema. Cada operador é caracterizado por três entidades: uma função de adição, uma função de deleção e uma pré-condição. Um operador é aplicável a um dado problema somente se suas pré-condições são satisfeitas. Os efeitos da aplicação de um operador para um dado problema, consistem em adicionar e deletar do estado do mundo todas aquelas cláusulas especificadas nas funções de deleção e adição prescritas pelo operador. Ou seja, as funções de adição e deleção descrevem como o operador altera o estado do mundo. Essas funções são descritas por listas de cláusulas que devem ser adicionadas ou deletadas.

Através dos operadores (ações) STRIPS tenta resolver uma meta do estado do mundo. Se a meta não é satisfeita totalmente, um operador é escolhido de modo que resolva parte da meta. Tipicamente o operador mais relevante é escolhido, ou seja, é escolhido o operador que altera o estado do mundo de modo que fique o mais próximo da meta. Este procedimento de busca com o uso de operadores corresponde à técnica MEA.

Inicialmente STRIPS foi projetado para controlar o robô Shakey [B.Wahlstrom. 1968]. A Figura 2.8 mostra uma exemplo do "Mundo de Shakey" que consiste em quatro salas dispostas ao longo de um corredor, onde cada sala tem uma porta e um interruptor de luz.

No "Mundo de Shakey" o robô pode executar os seguintes movimentos: Se mover entre

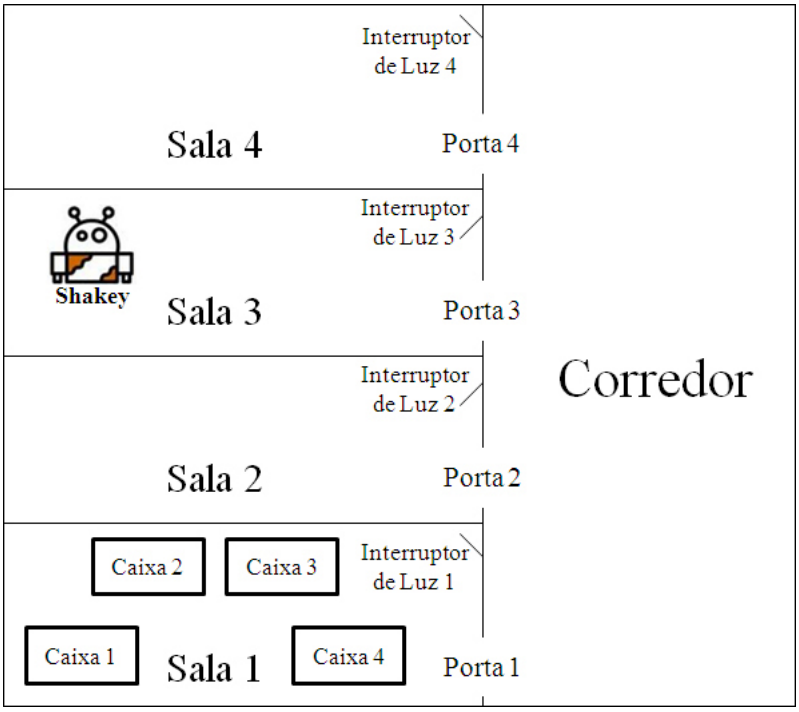

Figura 2.8: Problema "Mundo de Shakey".

pontos dentro de uma sala, pode passar pela porta entre salas, pode subir e descer de objetos que permitam essa ação, empurrar caixas e ligar e desligar interruptores de luz. Esses movimentos geram as seguintes possíveis ações:

- *Ir*(x, y): Consiste em mover o Shakey de x para y. Exige que o Shakey esteja em x e que x e y sejam posições na mesma sala. Existe uma porta entre as duas salas, estando ligadas pelo corredor.
- *Empurrar*(b, x, y): Empurrar uma caixa b da posição x para a posição y. Como pré-condições temos que b precisa ser uma caixa e x e y sejam salas.
- *Subir*(b): Subir em uma caixa b. Para isso b tem de ser uma caixa.
- *Descer*(b): Descer de uma caixa b. Para isso b tem de ser uma caixa.
- *Ligar*(s): Ligar um interruptor s. Para isso s tem de ser um interruptor e Shakey tem de estar em cima de uma caixa na posição do interruptor.
- *Desligar*(s): Desligar um interruptor s. Para isso s tem de ser um interruptor e Shakey tem de estar em cima de uma caixa na posição do interruptor.

A seguir é descrito um exemplo simplificado do planejamento utilizando STRIPS.

Exemplo

A Figura 2.9 mostra um problema de mover uma caixa entre três salas. O objetivo é empurrar a caixa entre as salas utilizando um robô. O robô está inicialmente na Sala 1 e

a "Caixa1" na Sala 2. O acesso entre as salas é realizado através de portas. Este problema será chamado de "Mover a Caixa".

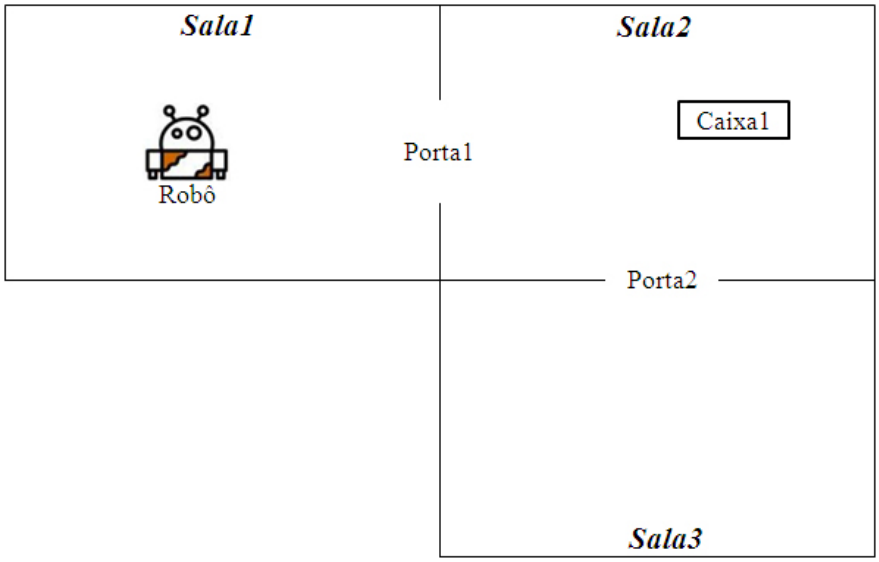

Figura 2.9: Problema "Mover a Caixa".

Na Figura 2.10 são descritos os operadores do problema "Mover a Caixa". O operador **IR**(d, r1, r2) significa que o Robô se move da sala $r1$ para $r2$ através da porta d. O operador **EMPURRAR**(b, d, r1, r2) empurra a caixa b da sala $r1$ para $r2$ através da porta d. Cada um destes operadores possui Pré-condição, Lista Deletar e Lista Adicionar.

O Estado Inicial e a Meta são apresentados na Figura 2.11. O Estado Inicial M0 é composto por literais que representam exatamente o estado inicial do problema. A meta consiste em mover a Caixa 1 da Sala 2 para a Sala 1. O planejador tenta provar que a meta G0 é verdadeira, ou seja, que ambos a Caixa 1 e o Robô estão na Sala 1. Essa primeira conjunção G0 é falsa. Consequentemente, o planejador busca um operador com uma expressão presente na Lista Adicionar que pode tornar essa conjunção verdadeira.

O operador **EMPURRAR** possui na Lista Adicionar `NASALA(b, r2)` e quando d é substituída pela Caixa 1 e $r2$ é substituído pela Sala 1, em seguida a primeira conjunção é satisfeita. O passo seguinte é tentar provar que este operador pode ser aplicado em M0. Essa tentativa falha, gerando um novo sub-problema G1.

IR(d, r1, r2)
`Pré-condição: NASALA(Robô, r1) ∧ CONECTADO(d, r1, r2)`
`Lista Deletar: NASALA(Robô, r1)`
`Lista Adicionar: NASALA(Robô, r2)`
EMPURRAR(b, d, r1, r2)
```
Pré-condição: CAIXA(b) ∧ NASALA(b, r1) ∧ NASALA(Robô, r1) ∧
              CONECTADO(d, r1, r2)
Lista Deletar: NASALA(b, r1), NASALA(Robô, r1)
Lista Adicionar: NASALA(Robô, r2), NASALA(b, r2)
```

Figura 2.10: Operadores do problema "Mover a Caixa".

Estado Inicial

```
M0:  NASALA(Robô, Sala1)
     CONECTADO(Porta1, Sala1, Sala2)
     CONECTADO(Porta2, Sala2, Sala3)
     CAIXA(Caixa1)
     NASALA(Caixa1, Sala2)
     (∀ x, ∀ y, ∀ z) (CONECTADO(x, y, z) ⇒ CONECTADO(x, z, y))
```

Metas

```
G0:  (∃ x)(CAIXA(x) ∧ NASALA(x, Sala1))
G1:  NASALA(Caixa, r1) ∧ NASALA(Robô, r1) ∧ CONECTADO(d, r1, Sala1)
G2:  NASALA(Robô, r1) ∧ CONECTADO(d, r1, Sala2)
```

Figura 2.11: Estado Inicial e Meta do problema "Mover a Caixa".

Planejamento

```
M1:  NASALA(Robô, Sala2)
     CONECTADO(Porta1, Sala1, Sala2)
     CONECTADO(Porta2, Sala2, Sala3)
     CAIXA(Caixa1)
     NASALA(Caixa1, Sala2)
     (∀ x, ∀ y, ∀ z) (CONECTADO(x, y, z) ⇒ CONECTADO(x, z, y))
M2:  NASALA(Robô, Sala1)
     CONECTADO(Porta1, Sala1, Sala2)
     CONECTADO(Porta2, Sala2, Sala3)
     CAIXA(Caixa1)
     NASALA(Caixa1, Sala1)
     (∀ x, ∀ y, ∀ z) (CONECTADO(x, y, z) ⇒ CONECTADO(x, z, y))
```

Plano

```
IR(Porta1, Sala1, Sala2)
EMPURRAR(Caixa1, Porta1, Sala2, Sala1)
```

Figura 2.12: Resultado do planejamento do problema "Mover a Caixa".

Finalmente, em G2 as pré-condições obtidas de **IR** são provadas serem verdadeiras em M0. Consequentemente, este operador pode ser aplicado. O plano final deste problema é composto pelas ações `IR(Porta1, Sala1, Sala2)` e `EMPURRAR(Caixa1, Porta1, Sala2, Sala1)`. As conjunções durante o planejamento e o plano final são apresentados na Figura 2.12.

2.2.4 Planejamento de Ordem Parcial

Antes de falarmos sobre planejamento de ordem parcial (POP), é necessário mencionar as características de um planejamento linear, que também é conhecido como planejamento total. Um planejador linear gera um plano sequencial, que consiste de uma sequência de ações que devem ser executadas em uma ordem restrita. Essa ordem é especificada durante o desenvolvimento do plano. Isso ocorre, pois esse tipo de planejamento geralmente executa uma ação a cada momento. Com isso, um plano sequencial que atinge uma dada meta executa uma e apenas uma ação a cada momento, ou seja, uma nova ação só pode ser executada quando a ação anterior tiver terminado sua execução.

A abordagem de planejamento linear é muito utilizada dentro da área de planejamento, pois o planejamento linear proporciona soluções que são encontradas de forma mais rápidas pelo planejador. Entretanto, os planos nem sempre são limitados pela linearidade das ações. Sendo assim, o planejamento de ordem parcial consiste em encontrar um ou mais planos onde as ações sejam dispostas em paralelo. Em outras palavras, algumas ações podem ser executadas ao mesmo tempo (paralelo).

Tal abordagem apresenta flexibilidade na ordem em que o plano é elaborado. Isto é, o planejador trabalha primeiro em decisões mais "óbvias" e "importantes", em vez de forçar que sejam dispostas em ordem cronológica e imutável. Segundo [Russell e Norvig 2003], tal estratégia de retardar uma escolha durante a busca é chamada de *compromisso mínimo*. O *compromisso mínimo* é um conceito útil para analisar as decisões que devem ser tomadas em qualquer problema de busca, já que pode tornar mais eficiente o planejamento. A Figura 2.13 apresenta a comparação entre o planejamento de ordem parcial e o planejamento de ordem total.

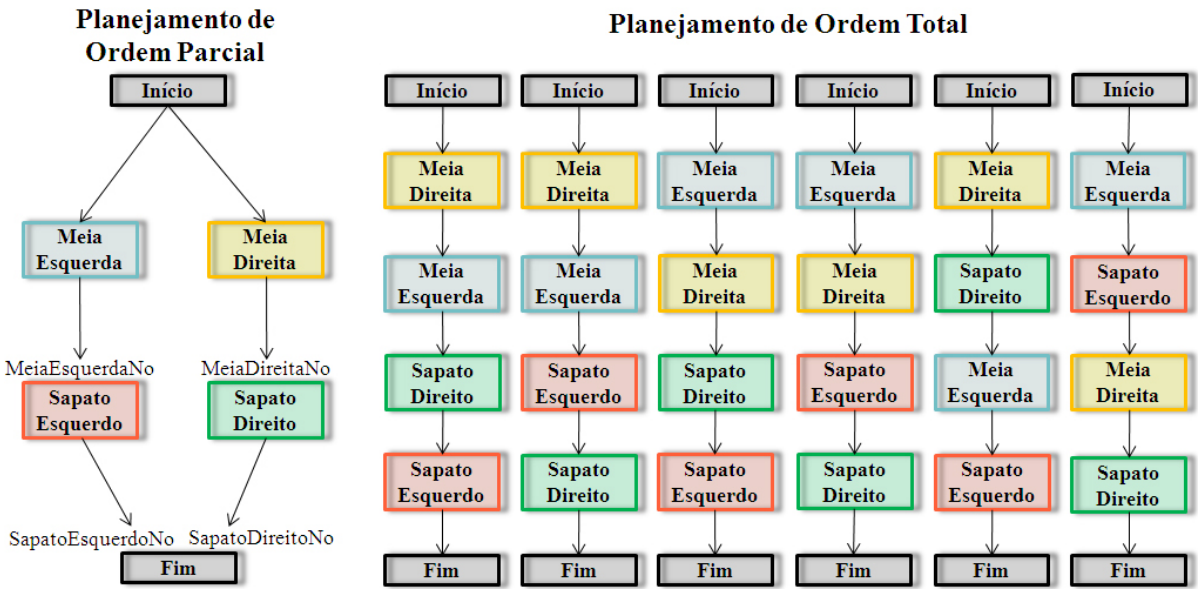

Figura 2.13: Comparação entre o planejamento de ordem parcial e total.

Originalmente, o planejamento de ordem parcial foi introduzido por [Sacerdoti 1975] como um meio de aumentar a eficiência do planejamento evitando a "escolha prematura de uma ordem em particular de ações com objetivo de alcançar uma meta". A utilidade deste método tem demonstrado como se pode resolver de forma eficiente o problema "Mundo dos Blocos", onde se encontra a *Anomalia de Sussman* [Sussman 1973]. Em [Mcallester e Rosenblitt 1991] é apresentada uma extensão do trabalho de [Sacerdoti 1975].

Análises comparativas dos planejamentos de ordem parcial e total são apresentadas em [Minton et al. 1991] e [Minton et al. 1994]. Geralmente, o planejamento de ordem parcial é melhor devido a paralelização de ações, que tendem a atingir a meta de modo mais rápido.

Entretanto, o melhor método de planejamento depende de fatores tais como especificação das heurísticas, procedimento de busca aplicado para solução do problema, e análise das características relativas ao domínio do problema. Para detalhar estes procedimentos, em [Rintanen 2005], é descrita a complexidade de alguns algoritmos.

O plano de ordem parcial consiste em um conjunto de ações, que inicialmente é formado por duas ações "fictícias" denominadas *Início* e *Fim*. A ação *Início* não possui pré-condições e tem como efeito todos os literais do estado inicial do problema. A ação *Fim* não tem efeito e tem como pré-condição os literais meta do problema de planejamento.

A ordem das ações é dada pela forma $A \prec B$. Em outras palavras, A deve ser executada antes de B, mas não necessariamente imediatamente antes. O momento dessa execução precisa ser respeitado, mas enquanto B não tiver sido executada a ação A pode ter seu início adiado. Ordens que formam ciclos não podem ser adicionadas ao plano, ou seja, $A \prec B$ e $B \prec A$ é um conflito (ciclo) e não faz parte do plano. Para o plano inicial sempre iremos ter $Inicio \prec Fim$.

O POP trabalha com uma especificação chamada vínculo (link) causal. Um vínculo causal entre duas ações A e B no plano é escrito como $A \xrightarrow{p} B$. Este vínculo significa que A alcança B através de p, ou seja, o efeito p da ação A satisfaz pelo menos uma pré-condição de B. Como já dito, se uma não for possível satisfazer a meta com uma dada ação, é preciso que essa pelo menos diminua a diferença do estado inicial para a respectiva meta. Na Figura 2.13 o vínculo $MeiaEsqueda \overset{MeiaEsquerdaNo}{\longrightarrow} SapatoEsquerdo$ significa que para colocar o sapato esquerdo é preciso ter colocada a meia esquerda. Uma ação C entra em conflito com $A \xrightarrow{p} B$ se o efeito de C é $\neg p$ e deve vir (pela ordem das ações) depois de A e antes de B. Alguns planejadores tratam o vínculo $A \xrightarrow{p} B$ como sendo protegido, pois p não pode ser negado no intervalo de $A \prec B$.

No planejamento de ordem parcial é mantido um conjunto de pré-condições em aberto (não satisfeitas). O planejador trabalha para encontrar ações que satisfaçam essas pré-condições sem incluir conflitos. O planejamento é executado até que seja encontrado um plano que não possua pré-condições em aberto. Na Figura 2.13 temos os seguintes componentes:

Ações: *MeiaEsquerda*, *SapatoEsquerdo*, *MeiaDireita*, *SapadoDireito*, *Início* e *Fim*.

Ordens: $MeiaEsquerda \prec SapatoEsquerdo$

$MeiaDireita \prec SapatoDireito$

Vínculos Causais: $MeiaEsquerda \overset{MeiaEsquerdaNo}{\longrightarrow} SapadoEsquerdo$

$MeiaDireita \overset{MeiaDireitaNo}{\longrightarrow} SapatoDireito$

$SapatoEsquerdo \overset{SapatoEsquerdaNo}{\longrightarrow} Fim$

$SapatoDireito \overset{SapatoDireitoNo}{\longrightarrow} Fim.$

Pré-condições em Aberto: {}

Definimos um plano consistente como um plano em que não existe nenhum ciclo nas restrições de ordem e nenhum conflito com os vínculos causais. Um plano consistente sem pré-condições abertas é uma solução. Antes de demonstrar um exemplo, começamos com uma formulação simplificada para problemas de planejamento proposicional.

- O plano inicial contem apenas o início (Estado Inicial) e o fim (Estado Meta), a restrição de ordem $Início \prec Fim$, nenhum vínculo causal e todas as pré-condições da meta em aberto (que devem ser satisfeitas).
- O sucessor escolhe arbitrariamente uma pré-condição aberta p em uma ação B e gera um plano sucessor para todo modo consistente possível para escolher uma ação A que alcance p.
- É realizado um teste que verifica se o plano é uma solução para o problema de planejamento original. Como são gerados apenas planos consistentes, o teste de meta precisa apenas verificar que não existem condições em aberto.

Vamos demonstrar um exemplo de planejamento de ordem parcial para o problema do "Pneu Sobressalente". O objetivo é ter um pneu sobressalente em bom estado corretamente montado no eixo do carro, onde o estado inicial tem um pneu furado no eixo e um pneu sobressalente bom no porta-malas. Neste cenário o procedimento geralmente adotado é efetuar a troca do pneu furado pelo pneu sobressalente. Contudo, principalmente à noite, uma ação possível é simplesmente abandonar o veículo e arcar com um provável prejuízo correspondente ao desaparecimento dos pneus. Algumas simplificações são realizadas para facilitar esta demonstração. Este problema é apresentado na Figura 2.14 utilizando a linguagem ADL (vai além de STRIPS pelo fato de aceitar literais negativos).

O resultado do planejamento é apresentado na Figura 2.15. São necessários 6 passos para encontrar a solução do problema. Primeiro a busca começa com o plano inicial contendo as ações *Início* e *Fim*. A esquerda de cada ação temos descritas suas respectivas pré-condições e a direita os efeitos. É importante salientar que o efeito de uma ação normalmente é a satisfação da pré-condição de outra ação, sendo escrito apenas uma vez. A explicação de cada passo é apresentada a seguir:

1. É realizado a escolha da única pré-condição em aberto $Em(Sobressalente, Eixo)$ da ação Fim. Existe apenas uma ação aplicável para essa pré-condição, sendo a ação `Montar`$(Sobressalente, Eixo)$.
2. É selecionada a pré-condição em aberto $Em(Sobressalente,\ Chão)$ da ação `Montar`$(Sobressalente,\ Eixo)$. Novamente existe apenas uma ação aplicável que satisfaz essa pré-condição: `Remover`$(Sobressalente,\ PortaMalas)$.

Estado Inicial$(Em(Furado, Eixo) \wedge Em(Sobressalente, PortaMalas))$
Estado Meta$(Em(Sobressalente, Eixo))$
Ação(`Remover`$(Sobressalente, PortaMalas)$,
`pré-condição`: $Em(Sobressalente, PortaMalas)$
`efeito`: $Em(Sobressalente, PortaMalas) \wedge \neg\, Em(Sobressalente, Chão)$)
Ação(`Remover`$(Furado, Eixo)$,
`pré-condição`: $Em(Furado, Eixo)$
`efeito`: $\neg\, Em(Furado, Eixo) \wedge Em(Furado, Chão)$)
Ação(`Montar`$(Sobressalente, Eixo)$,
`pré-condição`: $Em(Sobressalente, Chão) \wedge \neg\, Em(Furado, Eixo)$
`efeito`: $\neg\, Em(Sobressalente, Chão) \wedge Em(Sobressalente, Eixo)$)
Ação(`DeixarDuranteNoite`(),
`pré-condição`:
`efeito`: $\neg\, Em(Sobressalente, Chão) \wedge \neg\, Em(Sobressalente, Eixo) \wedge$
$\neg\, Em(Sobressalente, PortaMalas) \wedge \neg\, Em(Furado, Chão) \wedge$
$\neg\, Em(Furado, Eixo)$)

Figura 2.14: Descrição do problema "*Pneu Furado*".

3. Escolhemos a pré-condição em aberto $\neg Em(Furado, Eixo)$ da ação

   `Montar`$(Sobressalente, Eixo)$. Apenas para demonstrar um conflito selecionamos a ação `DeixarDuranteNoite`() em vez de `Montar`$(Furado, Eixo)$. Mas a ação

   `DeixarDuranteNoite`() também tem o efeito $\neg Em(Sobressalente, Chão)$, o que significa que ela está em conflito com o vínculo:

   $Remover(Sobressalente, PortaMalas) \overset{Em(Sobressalente,Chao)}{\longrightarrow} Montar(Sobressalente, Eixo)$.

4. Para solucionar este conflito, adicionamos uma restrição de ordem que coloca

   `DeixarDuranteNoite`() antes de `Remover`$(Sobressalente, PortaMalas)$. A única pré-condição aberta nesse momento é $\neg Em(Sobressalente, PortaMalas)$. Entretanto, qualquer ação escolhida entra em conflito com outra ação. Assim, somos obrigados a remover a ação `DeixarDuranteNoite`(), pois essa ação não funciona como um modo de trocar pneu.

5. Mais uma vez selecionamos a pré-condição $\neg Em(Furado, Eixo)$. Dessa vez escolhemos a ação `Remover`$(Furado, Eixo)$.

6. Para finalizar o planejamento, são satisfeitas as pré-condições

   $Em(Sobressalente, PortaMalas)$ e $Em(Furado, Eixo)$ pela ação *Início*.

Em [Russell e Norvig 2003] é apresentado um pseudocódigo do algoritmo POP. Este método consiste em uma busca baseada na escolha e conflito de operadores.

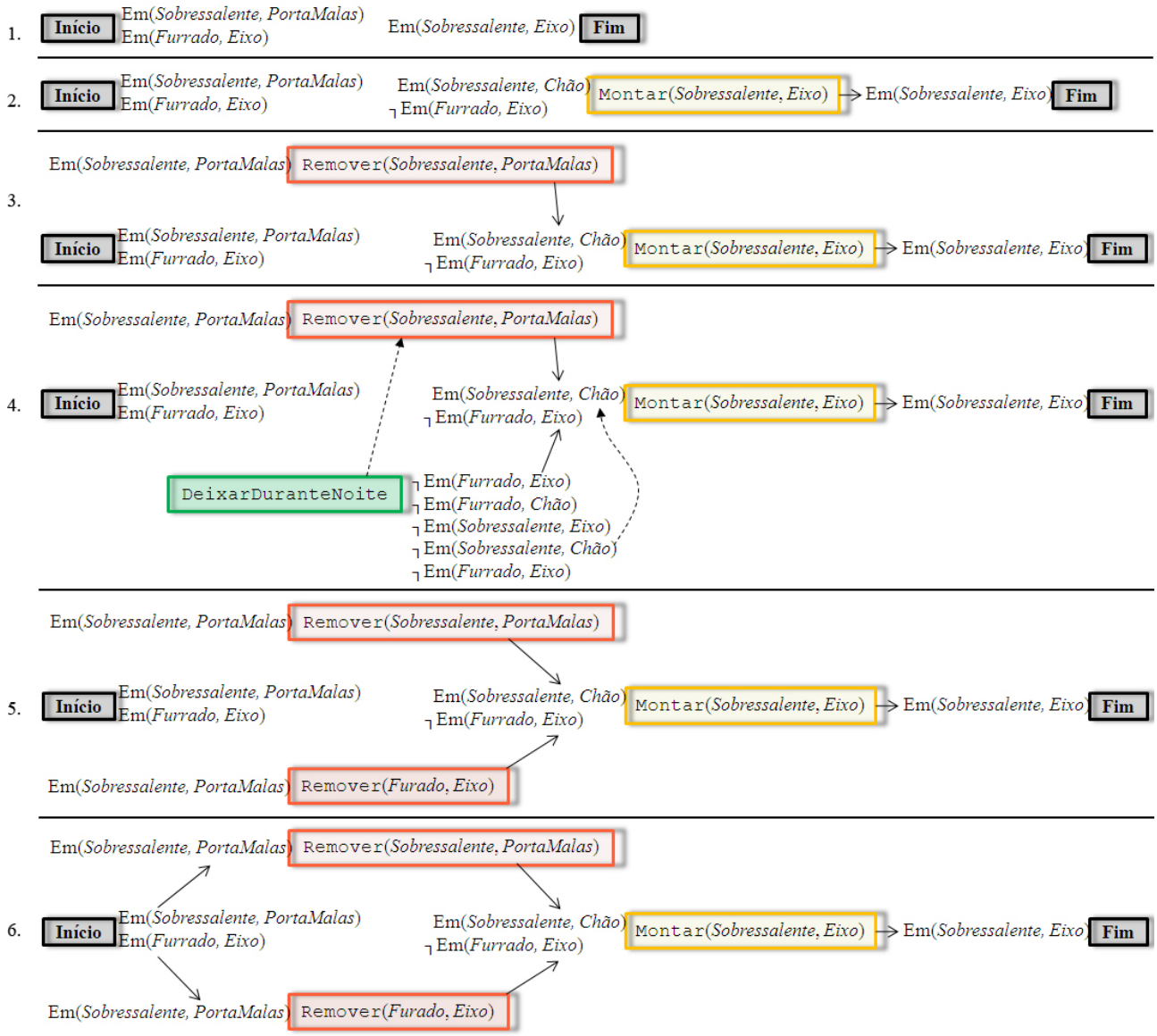

Figura 2.15: Planejamento de ordem parcial para o problema da Figura 2.14.

Capítulo 3

Produção de Recursos para Jogos RTS e Trabalhos Correlatos

Os trabalhos e conceitos de produção de recursos em jogos RTS que deram base para o desenvolvimento dessa dissertação, que também auxiliam no entendimento dos problemas e desafios presentes nessa área, serão apresentados no decorrer desta seção. Inicialmente são descritos os principais conceitos de jogos RTS. Uma abordagem que envolve a aplicação de planejamento em jogos do tipo RTS é apresentada em [Chan et al. 2007, Chan et al. 2008]. Nesse trabalho, o objetivo é desenvolver planos de ações que levem o jogo de um estado inicial de recursos a um dado estado meta (objetivo). Este problema é composto pelo estado inicial de recursos do jogo, pelo conjunto de ações e a meta de recursos.

O domínio de ações utilizado no trabalho [Chan et al. 2007] é o jogo *Wargus*, tal domínio é apresentado na Figura 3.1. Este mesmo domínio de ações é usado para os testes de planejamento do trabalho de [Chan et al. 2007]. As ações presentes em jogos RTS possuem muitas pré-condições e restrições em suas execuções. Para construir um plano de ações é necessário atender todas as restrições de uma determinada ação antes de colocá-la nele.

Uma vez que é obtido um plano de ações que atinge certa meta no trabalho de [Chan et al. 2007], esse plano é submetido a um algoritmo de escalonamento. Esse tem por objetivo paralelizar o máximo de ações possíveis para que o *makespan* seja diminuído.

A arquitetura de planejamento de [Chan et al. 2007] é composta por dois elementos. O primeiro é o Planejador Sequencial que é responsável por retornar um plano sequencial. O segundo elemento da arquitetura estabelece quais ações podem ser executadas em paralelo. Para essa tarefa, é usado o algoritmo de Escalonamento. Este procedimento busca quais ações podem ser executadas no tempo anterior de término ou início de outra ação.

O planejador sequencial determina quais ações devem fazer parte do plano para que a meta seja alcançada. Neste passo, todas as ações são estabelecidas sequencialmente. O problema de estabelecer uma sequência de ações é que o *makespan* é muito superior ao *makespan* de um plano com ações em paralelo. Sendo assim, algumas ações podem ser

```
resource townhall
resource peasant
resource barracks
resource supply
resource footman
resource gold
resource wood

action build-townhall :duration 1530
 :borrow 1 peasant :consume 1200 gold 800 wood
 :produce 1 townhall
action build-peasant :duration 225
 :borrow 1 townhall :consume 400 gold 1 supply
 :produce 1 peasant
action build-barracks :duration 1240
 :borrow 1 peasant :consume 700 gold 450 wood
 :produce 1 barracks
action build-supply :duration 620
 :borrow 1 peasant :consume 500 gold 250 wood
 :produce 4 supply
action build-footman :duration 200
 :borrow 1 barracks :consume 600 gold 1 supply
 :produce 1 footman
action collect-gold :duration 510
 :require 1 townhall :borrow 1 peasant
 :produce 100 gold
action collect-wood :duration 1570
 :require 1 townhall :borrow 1 peasant
 :produce 100 wood
```

Figura 3.1: Domínio de recursos e ações do jogo *Wargus* [Chan et al. 2008].

executadas ao mesmo tempo (paralelo), tornando o *makespan* menor.

Por exemplo, no domínio do *Wargus* se o estado inicial é (*2 peasant, 1 townhall*) e a meta (*200 Gold*), o planejador sequencial determinaria que o plano é (*collect-gold, collect-gold*). Neste caso, apenas ao término da primeira ação que a outra poderia ser executada. Entretanto, é possível executar essas ações ao mesmo tempo, pois existem *2 peasant* disponíveis, que são pré-condição para que a ação (*collect-gold*) seja executada.

Outro trabalho de planejamento em jogos RTS é o de [Branquinho et al. 2011b, Branquinho et al. 2011a]. Esse trabalho vai em direção ao mesmo objetivo do trabalho de [Chan et al. 2007]. Entretanto, na abordagem de [Branquinho et al. 2011b] é utilizado Planejamento de Ordem Parcial e o algoritmo de Busca e Aprendizado SLA*. Nele, o POP é responsável por desenvolver um plano de ações parcial que atinge uma dada meta. Assim que o plano é feito, o SLA* se encarrega de escalonar o plano colocando o máximo de ações possíveis em paralelo.

O domínio de ações utilizado por [Branquinho et al. 2011b] assim como o de [Chan et al. 2007] é o jogo *Wargus*, representado pela Figura 3.1. Sua abordagem é baseada no

trabalho de [Chan et al. 2007], onde o objetivo principal é encontrar soluções melhores. Nesse caso soluções melhores são planos de ações com *makespan* menor. Sempre que um plano de ações é submetido ao escalonamento utilizando o SLA* o resultado ótimo para tal escalonamento é atingido. A única restrição dessa abordagem é o tempo (*runtime*) gasto pelo escalonador, que é bem maior que o de [Chan et al. 2007]. Entretanto, o algoritmo de [Chan et al. 2007] não tem garantia de melhor solução na tarefa de escalonamento.

Em ambas as abordagens, fica claro que a produção de recursos em jogos RTS passa por duas etapas. A primeira onde é construído um plano de ações que considera o estado atual do jogo como estado inicial de recursos, e a meta estipulada como estado final de recursos a ser atingido através de um plano de ações. Esse plano é quem promove a produção de novos recursos através da execução de suas ações. A segunda etapa é aquela onde o plano de ações obtido na etapa anterior é submetido a algum processo de melhoria em sua construção. Nesse processo são feitas alterações como mudanças na ordem de execução, nos parâmetros e até nos tipos das ações presentes no plano, com o objetivo de melhorar o desempenho do mesmo.

Nas abordagens de [Chan et al. 2007] e [Branquinho et al. 2011b] a primeira etapa da produção de recursos é feita usando planejadores. Na primeira abordagem é utilizado um planejador sequencial para o desenvolvimento do plano de ações, já na segunda um planejador de ordem parcial é utilizado. Para a segunda etapa, onde é feita uma melhoria no plano de ações, ambas as abordagens buscam diminuir o *makespan* do mesmo, como uma forma de otimizá-lo. Nesta etapa diferentes tipos de técnicas são usadas, e serão melhor apresentadas no decorrer desta seção. Embora a maioria dos trabalhos de planejamento em jogos RTS considerem as duas etapas como padrão em suas pesquisas, existem outras metodologias que podem ser consideradas para a produção de recursos em jogos RTS. Um trabalho que vai em direção a uma abordagem diferente é o trabalho de [Fayard 2005].

No trabalho de [Fayard 2005] é proposto o uso de busca estocástica para auxiliar no desenvolvimento de um plano de ações. Uma vez que obtêm-se um um plano de ações inicial, o objetivo do seu trabalho é utilizar o *Simulated Annealing* para especificar novas ações que possam modificar e compor esse plano. O resultado esperado é um plano com novas ações e consequentemente a geração de novos recursos. O objetivo deste é tentar mensurar e avaliar os parâmetros dos recursos presentes em diferentes classes de um jogo RTS, através de comparações entre os planos de ações obtidos para cada uma dessas. Com essa abordagem, é possível balancear tais parâmetros e consequentemente os recursos em um jogo de RTS. O domínio de ações e recursos utilizados por [Fayard 2005] é o jogo *StarCraft* descrito de forma resumida na Figura 3.11. Esse trabalho serviu como inspiração para essa pesquisa, junto com os trabalhos de [Chan et al. 2007] e [Branquinho et al. 2011b]. Todos eles serão apresentados com mais detalhes nas próximas seções.

As próximas seções irão detalhar as abordagens apresentadas e juntamente com essas as informações mais relevantes para o entendimento desta pesquisa. A Seção 3.1 apresenta

os principais conceitos sobre jogos RTS, exemplificando o domínio utilizado nesta pesquisa. Na Seção 3.2 são apresentados os tipos de planejadores. Na Seção 3.3 são descritas as características de escalonamento de ações e sua importância nesta pesquisa. Já na Seção 3.4 é apresentada a abordagem de balanceamento de parâmetros em jogos RTS e sua contribuição para este trabalho aqui proposto. Por fim, a Seção 3.5 apresenta os detalhes da construção e objetivo da abordagem de escolha de metas através da maximização da produção de recursos aqui apresentada.

3.1 Jogos RTS

Jogos de estratégia em tempo real ou Jogos RTS (um acrônimo para *Real Time Strategy*) são jogos de estratégia militar onde as decisões e ações ocorrem em um ambiente de tempo real [Buro 2004].

Um dos primeiros títulos de sucesso entre jogos de RTS foi *Dune II: The Building of a Dynasty*. Nesse os principais aspectos e características presentes nos jogos atuais foram desenvolvidos, tais como: Diferentes quantidades de recursos divididos em classes distintas de personagens e produção de unidades e batalhas contra o inimigo [Adams 2006]. Após o sucesso de *Dune II* vieram diversos títulos como *Command and Conquer*[1] e famosos títulos *Warcraft* e *Strarcraft*[2], as figuras 3.2 e 3.3 mostram imagens dos dois últimos. Além destes, existem diversos outros jogos que fazem sucesso no universo de estratégia em tempo real. Para essa dissertação, ênfase é dada ao jogo *Strarcraft* que é o domínio utilizado aqui e será descrito no decorrer desta seção.

Figura 3.2: Imagens dos jogos *Warcraft II* e *Warcraft III*.

O principal objetivo em um jogo RTS é conseguir desenvolver um exército e utilizá-lo em batalhas para derrotar os inimigos. Essas batalhas podem ser de caráter ofensivo, onde a base do adversário é atacada ou de caráter defensivo neutralizando e combatendo uma investida do inimigo. O papel de um ou de vários inimigos em uma partida pode

[1] *Command & Conquer* foi criado pela empresa *Westwood Studios*. Atualmente é desenvolvido pela *Electronic Arts*: `http://www.ea.com/`

[2] *Warcraft* e *StarCraft* são produzidos pela *Blizzard Entertainment*: `http://www.blizzard.com/`

Figura 3.3: Imagens dos jogos *StarCraft* e *StarCraft II*.

ser desempenhado por um jogador humano ou por um computador utilizando um agente inteligente como jogador. No decorrer de uma partida, as decisões tomadas dentro do jogo são destinadas à produção de recursos e táticas de batalha. Para produzir recursos é necessário executar ações, tais como: coletar recursos, construir unidades militares e desenvolver a tecnologia da base. Quais recursos serão produzidos e a ordem na qual as ações que produzem os mesmos serão executadas é uma tarefa muito importante a ser feita. Além do mais, as informações relativas ao mapa e ambiente do jogo são desconhecidas e no decorrer da partida sofrem alterações. Isso torna ainda mais importante um correto planejamento.

Jogos RTS em sua maioria possuem duas fases distintas. A primeira onde o jogo é iniciado e todos os jogadores possuem um tempo para desenvolver seu exército via produção de recursos. Na segunda fase, os recursos produzidos na fase anterior são utilizados em batalhas para vencer os oponentes. Assim, determinar quais recursos serão construídos e maximizar o desenvolvimento desses durante a primeira fase é de vital importância para o sucesso no jogo.

Recursos em um jogo de estratégia em tempo real podem ser todos os tipos de construções base, unidades militares, matéria prima e civilizações. Para reunir matéria prima é necessário que um tipo específico de unidade militar execute essa ação, sendo que essa unidade é responsável por construir outros recursos. Já para construir alguma base é preciso que a quantidade de matéria prima necessária para desenvolver tal recurso esteja reunida. Uma das tarefas de planejamento é determinar a quantidade e a ordem em que esses recursos serão coletados ou construídos, de modo que seja possível executar ambos.

Para executar técnicas e algoritmos de planejamento em um ambiente de jogo RTS são utilizados *Mods*[3] ou jogos desenvolvidos exclusivamente para tal tarefa. O *mod Wargus* é um módulo do jogo *World of Warcraft II*, ele possui unidades militares e recursos a serem coletados e utilizados dentro do jogo. Já a *engine Stratargus* é um ambiente de código aberto onde é possível desenvolver jogos RTS baseados no *Warcraft* para testar algoritmos

[3]Mods são jogos desenvolvidos a partir do fragmento de um jogo maior. Dessa forma usuários podem acessar o código fonte e controlar a execução de parte de um jogo para modificá-lo ou fazer testes

de IA. Apesar de serem duas ferramentas muito utilizadas em testes, ambas possuem uma grande limitação na quantidade de recursos que oferecem aos jogadores. No *Wargus* que é a ferramenta mais popular, são cerca de sete recursos disponíveis o que limita e simplifica as tarefas de planejamento.

O *StarCraft* foi escolhido como domínio para os testes por ser o jogo RTS que mais possui recursos (por volta de 40 para cada classe) e forte quantidade de pré-condições (pré-requisitos) e restrições por recurso [Churchil e Buro 2011]. Assim, a abordagem foi construída para operar sobre um domínio que eleva ao máximo os desafios e tarefas de planejamento em jogos de RTS. Existem três diferentes classes de personagens no *StarCraft*. Essas são *Terran*, *Protoss* e *Zerg*. As classes possuem diferentes tipos de recursos, sendo que neste trabalho foco é dado aos recursos da classe *Terran*. A principal base dessa classe é o *CommandCenter* (Centro de Comando) essa é responsável por habilitar a construção de diversas outras bases, sendo cada uma dessas bases um tipo específico de recurso. O *CommandCenter* é responsável por criar o *Scv* (Construtor) um dos recursos mais importantes do jogo. Esse é responsável por coletar recursos do tipo *Minerals* e *Gas* que são utilizados nas construções, além de ser o recurso que executa as ações de construir bases e estruturas. Quando um *Scv* executa uma ação de coletar minerais ele reúne 50 *Minerals* que são depositados no *CommandCenter* mais próximo; já quando coleta gás ele reúne 25 *Gas* e deposita em outra base chamada *Refinery*. O *CommandCenter* pode sofrer atualizações em sua tecnologia tendo capacidade de executar novas funcionalidades. De fato, várias bases possuem essa característica. Quando uma partida é iniciada no *StarCraft*, o jogador já começa com 1 *CommandCenter*, 4 *Scv*, 50 *Minerals* disponíveis para começar sua produção de recursos. Esse é o estado inicial de recursos nativo do *StrarCraft*.

A Figura 3.7 exibe o *CommandCenter* e suas principais atualizações tecnológicas dentro do jogo. A imagem (A) mostra a estrutura em seu estágio inicial, as imagens (B) e (C) são as atualizações que podem ser feitas no *CommandCenter*, quando construídas as estruras (B) e (C) ficam acopladas a estrutura (A).

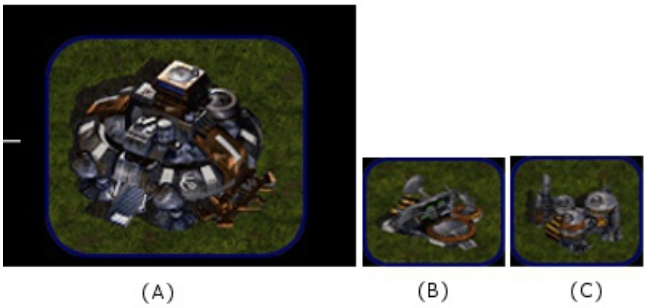

Figura 3.4: *CommandCenter* e suas atualizações tecnológicas.

O Recurso *Scv* executa as ações de coletar minerais e gás, além de construir as prin-

cipais estruturas do jogo. Para criar um *Scv* é necessário consumir 50 *Minerals* e ter um *CommandCenter* disponível. Assim, enquanto o *Scv* estiver sendo criado o *CommandCenter* usado para essa tarefa ficará ocupado, ou seja, o *CommandCenter* é responsável por desenvolver o *Scv* e só poderá executar uma nova ação quando essa última tiver sido finalizada. Quando um recurso é responsável por executar uma ação que constrói outro recurso dizemos que o primeiro é o recurso base do último. Dentre as várias pré-condições de um recurso, existe aquela que representa o recurso responsável por executar a ação e criá-lo dentro do jogo, sendo esse considerado seu recurso base. Todos os recursos do jogo possuem um recursos base. Por exemplo, *CommandCenter* é o recurso base do *Scv*. A Figura 3.5 mostra o *Scv*.

Figura 3.5: O recurso *Scv* e suas ações de coletar *Minerals* e *Gas*.

Uma área onde é possível encontrar minerais é chamada de *Mineral Field* e uma onde encontram-se reservas de gás é chamada de *Vespene Geiser*. A Figura 3.6 exibe essas respectivas áreas, na imagem (a) temos o *Mineral Field* e na imagem (b) o *Vespene Geiser*. O tempo estipulado para o *Scv* executar uma ação de *collect-minerals* ou de *collect-gas* foi estipulado baseado em testes, onde foi estimada a média de tempo gasto por esse em diferentes áreas de coletas espalhadas pelo jogo. Os valores de 45 segundos (seg.) para coletar 50 *Minerals* e 20 segundos para coletar 25 *Gas* levam em conta a distância para as áreas de coleta.

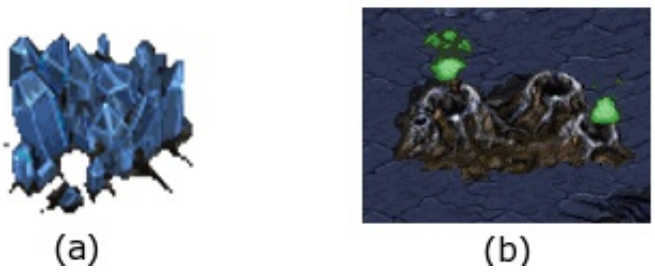

Figura 3.6: *Mineral Field* de onde são coletados *Minerals* e *Vespene Geiser* usado para extrair *Gas*.

O *Supply* é outro recurso consumido quando unidades são produzidas. Utilizado especificamente por unidades militares, esse recurso limita a quantidade de soldados, veículos

e aeronaves que são construídos. Para aumentar a quantidade de *Supply* é necessário construir um *SupplyDepot*. Essa estrutura aumenta em mais 8 a quantidade de *Supply*. A Figura 3.6 exibe o *SupplyDepot*.

Figura 3.7: Imagem do recurso *SupplyDepot*.

Existem diversos tipos de unidades militares no jogo. Esses recursos são os que mais contribuem para o aumento do poder militar do jogador, devido a suas capacidades de atacar e defender. As unidades militares se dividem em unidades terrestres (*Ground Units*) com soldados e veículos e unidades aéreas (*Air Units*) com as aeronaves. Os recursos militares dos tipos combatentes e veículos são as unidades móveis de infantaria caracterizados por soldados e tanques. Cada um desses possui alguma habilidade especial que é utilizada em combate. Dentre esses, estão o *Firebat*, *Marine*, *Vulture* dentre outros. A Figura 3.8 mostra as principais unidades militares terrestres presentes no jogo. Já as unidades aéreas do jogo são aquelas unidades que movem-se rapidamente pelo ar e podem atacar qualquer outro recurso. Essas unidades só podem ser atacadas por outras unidades aéreas e algumas unidades terrestres especiais. A Figura 3.9 mostra as unidades militares aéreas do jogo. Além das características físicas e de combate, as unidades militares diferem-se também na quantidade de recursos gastos para produzi-las. Quanto mais poderosa a unidade, mais minerais, gás e *supply* serão gastos na sua produção.

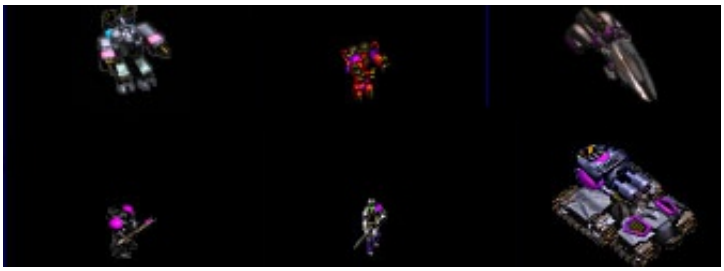

Figura 3.8: Principais unidades terrestres do *StarCraft*.

Para produzir qualquer tipo de unidade militar é preciso ocupar por um tempo alguma Estrutura (*Building*) responsável por construir essa unidade. No caso, produzir um *Firebat*

Figura 3.9: Principais unidades aéreas do *StarCraft*.

requer ocupar uma *Barracks* por um tempo. Outras estruturas também são usadas para produzir unidades como a *Factory* ou o *Starport*. As estruturas também habilitam a produção de recursos militares mais avançados e de armas especiais que são utilizadas por esses. Por exemplo, o recurso *BatleCruiser* só pode ser construído se a estrutura *PhysicsLab* estiver presente no jogo, pois essa possui a tecnologia necessária para habilitar a produção do recurso. Outro exemplo é a estrutura *ScienceFacility* que habilita que a unidade militar *Ghost* possa usar um ataque especial dentro do jogo. A Figura 3.10 mostra as principais estruturas presentes no *StarCraft*.

No *StarCraft* todos os recursos são produzidos através de ações. Ações essas que possuem pré-condições e efeitos. As pré-condições de uma ação são os recursos que precisam estar disponíveis para que ela seja executada. Já o efeito de uma ação é a produção do seu respectivo recurso. Na Figura 3.11 é descrito de forma resumida o domínio de ações do *StarCraft*. Nessa, estão ilustradas as principais características das ações que são usadas no jogo. As ações estão ligadas a quatro especificações de recursos: *Borrow*, *Consume*, *Require* e *Produce*.

Quando uma ação está ligada a uma especificação *borrow* significa que essa ação necessita pegar/tomar "emprestado" aquele recurso para ser executada. Esse recurso fica ocupado, ou seja, não pode ser utilizado enquanto a ação que está ocupando-o não for finalizada. Quase sempre, uma ação ao ser executada ocupa (*borrow*) o seu recurso-base, que é aquele recurso responsável por executar a ação dentro do jogo. Contudo, o recurso-base de uma ação é também base para outras ações que devem disputar o tempo que esse recurso tem disponível para executá-las. Por exemplo, para construir um *SiegeTank* o recurso *Factory* fica ocupado por 32 segundos executando a ação *build-siegetank*, após o seu término o *Factory* volta a ficar disponível para outras ações. Recursos como esse são conhecidos também como recurso renovável, pois sempre que termina de ser utilizado fica disponível para novas tarefas.

Ações que possuem recursos *require* em suas pré-condições necessitam que esses estejam disponíveis quando elas forem executadas. Estar disponível quer dizer que o recurso

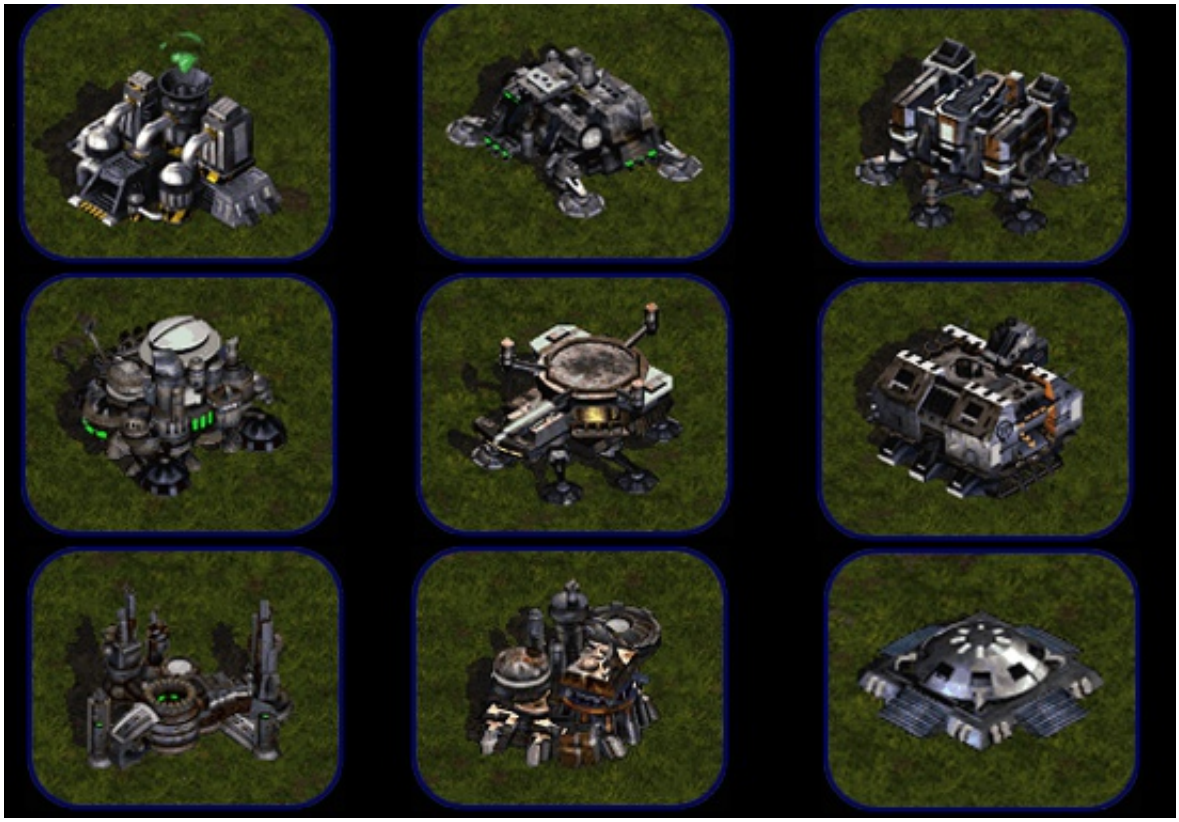

Figura 3.10: Principais estruturas do *StarCraft*.

já foi construído e está presente no jogo. Diferente do *borrow* o *require* não precisa ocupar o recurso para a execução da ação, é necessário apenas que o mesmo esteja disponível no jogo. Contudo, se o recurso não estiver disponível a ação não pode ser executada. Um exemplo é a ação $build - firebat$, para executá-la além de minerais, gás e seu recurso-base é preciso ter disponível no jogo um recurso *Academy* também. Sem tal recurso a ação não pode ser feita.

As ações do jogo têm por característica consumir os recursos utilizados para sua execução. Assim é necessário estar sempre reunindo mais recursos desse tipo antes de executar uma nova ação. Esses recursos podem ser *Minerals*, *Gas* ou *Supply*. Tais recursos são da especificação *consume* e estão presentes em todas as ações do jogo. Para produzir uma *Barracks* é preciso reunir 150 *Minerals* que serão deduzidos do montante do jogo quando a ação for iniciada. Se for preciso executar a mesma ação novamente, terão de ser reunidos mais 150 *Minerals*. Esses recursos são conhecidos também como não renováveis, pois uma vez que são usados não ficam disponíveis outra vez sendo preciso coletá-los novamente.

Uma ação ligada a um recurso *produce* é uma ação que produz um determinado recurso. No domínio utilizado neste trabalho, uma ação sempre produz o recurso com seu respectivo nome. O recurso produzido é colocado entre os recursos disponíveis do jogo. A ação

```
resource CommandCenter
resource Barracks
resource Factory
resource CovertOps
resource Scv
resource Firebat
resource Ghost
resource SiegeTank
resource BatleCruiser
resource Minerals
resource Gas

action build-commandcenter :duração 75 seg.
 :borrow 1 Scv :consume 400 Minerals
 :produce 1 CommandCenter
action build-barracks :duração 50 seg.
 :require 1 CommandCenter :borrow 1 Scv :consume 150 Minerals
 :produce 1 Barracks
action build-factory :duração 50 seg.
 :require 1 Barracks :borrow 1 Scv
 :consume 200 Minerals 100 Gas :produce 1 Factory
action build-covertops :duração 25 seg.
 require:  1 ScienceFacility :borrow 1 Scv
 :consume 50 Minerals 50 Gas :produce 4 supply
action build-scv :duration 13 seg.
 :borrow 1 CommandCenter :consume 50 Minerals 1 Supply
 :produce 1 Scv
action build-firebat :duração 15 seg.
 require:  1 Academy :borrow 1 Barracks
 :consume 50 Minerals 25 Gas 1 Supply :produce 1 Firebat
action build-ghost :duração 32 seg.
 require:  1 Academy 1 CovertOps :borrow 1 Barracks
 :consume 75 Minerals 75 Gas 1 Supply :produce 1 Ghost
action build-siegetank :duração 32 seg.
 require:  1 MachineShop :borrow 1 Factory
 :consume 150 Minerals 100 Gas 2 Supply :produce 1 SiegeTank
action build-batlecruiser :duração 84 seg.
 require:  1 ControlTower 1 PhysicsLab :borrow 1 Starport
 :consume 400 Minerals 300 Gas 8 Supply :produce 1 BatleCruiser
action collect-minerals :duration 45 seg.
 :require 1 CommandCenter :borrow 1 Scv
 :produce 50 Minerals
action collect-gas :duration 20 seg.
 :require 1 Refinery :borrow 1 Scv
 :produce 25 Gas
```

Figura 3.11: Algumas das principais ações do domínio de recursos do *Strarcraft*.

build-scv produz 1 recurso *Scv*. Já a ação *collect-minerals* produz 50 *Minerals* e assim por diante.

Uma ação possui recursos que serão utilizados para sua produção, recursos que serão consumidos e recursos que serão produzidos. Além destes, uma ação possui um tempo de execução que representa o tempo necessário para que ela seja executada dentro do jogo. Esse tempo é definido em segundos e não em ciclos ou FPS (*frames per second*) como geralmente é feito em abordagens utilizando jogos RTS. Isso é feito, devido as diversas fontes de consulta na internet que trazem o tempo gasto para a execução de cada ação do *Strarcraft* em segundos. Assim é possível ter uma estimativa mais clara e precisa dos tempos durante a apresentação dos experimentos. Besides consuming, use and produce resources, an action also has a time, which is the time required for it to be executed in the game.

3.2 Planejadores

3.2.1 Planejador Sequencial

O planejador sequencial de [Chan et al. 2007] é baseado na técnica MEA. No seu trabalho, o uso de um planejador sequencial é o suficiente para garantir um plano de ações que possa ser submetido ao processo de escalonamento feito posteriormente. Esse planejador opera de modo a selecionar uma submeta que decrementa a diferença entre o estado inicial e a meta. As ações necessárias para resolver essa submeta são então executadas. Ao selecionar uma submeta, essa pode ser considerada uma nova meta e ser resolvida determinando a submeta necessária para resolvê-la, e assim por diante. Através de um processo recursivo, essas e todas as demais submetas vão sendo resolvidas, até que todas as metas estejam satisfeitas.

Durante a execução de um planejador, é preciso assumir que não existe nenhum tipo de recurso que seja produzido e consumido por uma mesma ação ao mesmo tempo. Todas as ações executam um tipo específico de efeito sobre os recursos do jogo, seja ele de produzir ou consumir os recursos. O Algoritmo3 mostra um pseudocódigo do planejador sequencial. O seu funcionamento será descrito de forma informal.

Para descrever alguns exemplos iremos utilizar o domínio da Figura 3.1, uma vez que os trabalhos de [Chan et al. 2007, Branquinho et al. 2011b] utilizam esse domínio. O algoritmo recebe como entrada o estado inicial de recursos do jogo S e a meta a ser atingida G. Após guardar o estado atual do jogo na variável PS com a função $Proj(S)$ (linha 1), o algoritmo escolhe diversas vezes através de recursividade uma meta ainda não satisfeita onde $R_i < g_i$ e tenta construir um plano $Plan'$ que a resolva. Ele faz isso até que o plano de ações completo tenha sido resolvido e não exista nenhuma meta pendente.

O algoritmo entre as linhas 5 e 10 estabelece todas as variáveis e valores de recursos

Algoritmo 3 MEA(S, G)

1: $PS \leftarrow Proj(S)$, estado projetado do jogo
2: **if** $\forall i, R_i \geq g_i$ é satisfeito por PS **then**
3: **return** $\emptyset$
4: **end if**
5: $R_i \leftarrow$ algum recurso onde $R_i < g_i$ em PS
6: $A_i \leftarrow$ ação que produz R_i
7: $r_i \leftarrow$ quantidade de R_i em PS
8: $\alpha \leftarrow$ quantidade de R_i produzida por A_i
9: $k \leftarrow ceil((g_i - r_i)/\alpha)$
10: $Acts \leftarrow$ plano sequencial com A_i repetido k vezes
11: $G^* \leftarrow \emptyset$ {Plano para satisfazer as pré-condições de A_i}
12: **for all** $R_j = p$ que são pré-condições de A_i **do**
13: **if** $R_j = p$ é uma précondição "*require*" ou "*borrow*" de A_i **then**
14: $G^* \leftarrow G^* \cup R_j \geq p$
15: **end if**
16: **if** $R_j = p$ é uma pré-condição "*consume*" de A_i **then**
17: $G^* \leftarrow G^* \cup R_j \geq k.p$
18: **end if**
19: **end for**
20: $Pre \leftarrow MEA(S, G^*)$
21: $Plan' \leftarrow concatenar(Pre, Acts)$
22: $S' \leftarrow$ estado do jogo depois de executar sequencialmente $Plan'$ a partir de PS
23: **return** $concatenar(Plan', MEA(S', G))$

necessários que devem ser produzidos para satisfazer a meta na iteração atual. A variável A_i determina qual ação deve ser executada, r_I, α e k determinam a quantidade de vezes que a ação deve ser executada e quantos recursos ela irá gerar dado a quantidade atual de existente de recursos no jogo. Os *loops* seguintes determinam qual ação vai compor uma nova meta G^* que será passada como parâmetro para o algoritmo ser invocado novamente. Assim são geradas todas as submetas necessárias. Por fim, o algoritmo concatena as ações contidas nas submetas gerando um plano de ações completo.

Uma das vantagens do MEA é que é possível estabelecer todas as ações necessárias que compõe um plano sem necessidade de verificar se alguma meta não foi cumprida. Para que isso aconteça, o algoritmo procura resolver sempre às pré-condições de recursos renováveis de uma ação antes de resolver as de recursos não renováveis. Assim, as pré-condições de recursos renováveis quando resolvidas, sempre ficaram resolvidas devido ao aumento desses recursos que acontece a cada nova meta que é resolvida.

O MEA então, quando produz um plano de ações, utiliza o mínimo de recursos necessários para atingir uma dada meta. Cada ação acrescentada pelo algoritmo é necessária para completar a meta, assim não é preciso também fazer verificações redundantes. Com isso, o algoritmo tem a quantidade de iterações relativa à quantidade de ações necessárias para atingir a meta estipulada.

É preciso ficar atento ao grafo de dependências dos recursos. Pois, se houver ciclos

nesses é possível que o MEA fique preso neles tentando resolver uma meta para sempre. Um exemplo de ciclo seria uma ação de *collect-gold*. Essa, necessita de uma *Townhall* e de um *Peasant*, caso tenhamos apenas esse último disponível, o algoritmo iria tentar executar uma ação de *build-townhall* . Essa iria necessitar de *Gold* que iria necessitar de *Peasant* que novamente iria necessitar da *Townhall* que está ausente. Gerando um ciclo infinito de satisfação de pré-condições. Entretanto, é possível estender o algoritmo para detectar tais ciclos, e para constar tanto no *Wargus* quanto no *StarCraft* o estado inicial contém os recursos necessários para evitar tais ciclos. O que pode acontecer é que no decorrer do jogo com as alterações que o mundo sofre, é possível que o estado do jogo em um dado momento esteja sem os recursos necessários para evitar tais ciclos, e se esse for considerado o estado inicial para um planejador o mesmo pode cair em um ciclo.

A seguir será apresentado um exemplo do funcionamento do algoritmo MEA.

Exemplo

Neste exemplo, o objetivo é encontrar um plano sequencial para o estado inicial (*1 Townhall, 3 Peasant, 400 Gold, 200 Wood*) e meta (*1 Townhall, 4 Peasant*). A Figura 3.12, mostra a primeira parte da meta que é resolvida pelo algoritmo para o problema em questão.

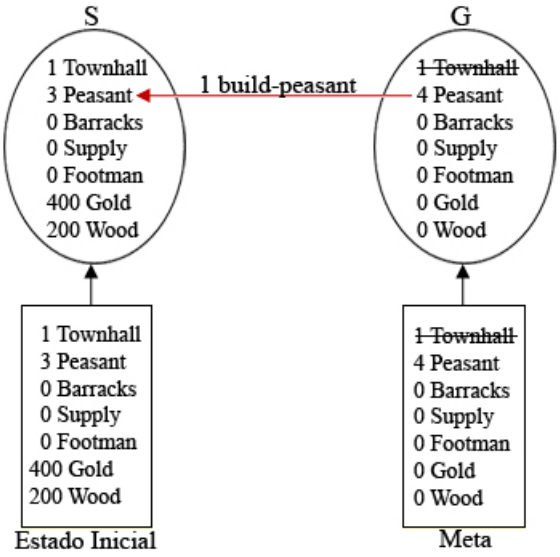

Figura 3.12: Primeiro passo do problema de planejamento sequencial.

A busca começa verificando cada recurso da meta G. O objetivo é determinar qual recurso não é satisfeito por S. Neste caso, é necessário criar *1 peasant* para satisfazer a diferença entre S (*3 Peasant*) e G (*4 Peasant*), já que a primeira parte da meta relativa ao recurso *Townhall* já estava satisfeito no início do algoritmo. Para isso, é preciso adicionar uma ação *build-peasant* ao plano. Após incluir esta ação é realizada uma chamada recursiva para o Algoritmo 3, tendo como objetivo a satisfação das pré-condições de

build-peasant. É importante salientar que as metas de G que não forem satisfeitas neste momento, serão resolvidas em um passo posterior da recursão.

O algoritmo continua selecionando novas ações que decrementam a distancia do estado atual para a meta. Cada uma dessas transforma-se em uma submeta que deve ser resolvida antes que uma nova ação que interfere diretamente na meta G seja escolhida. Por fim, temos um conjunto de ações que devem ser executadas em na ordem estrita em que são colocadas no plano. Se essa ordem não for seguida, não será possível executar determinadas ações. Isso ocorre, pois sempre que uma ação vai ser executada suas pré-condições que são satisfeitas por outras ações estão atrás dela na sequência de posições, portanto a ação só deve ser acionada quando as que estão antes dela já tiverem sido executadas. A Figura 3.13 mostra a sequência final do algoritmo MEA e o plano gerado.

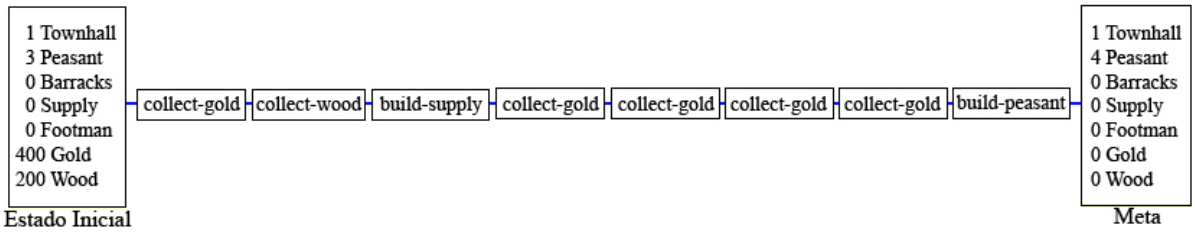

Figura 3.13: Plano sequencial obtido no exemplo tratado.

3.2.2 Planejador de Ordem Parcial

Além do planejador sequencial, nesta pesquisa também foi desenvolvido um planejador de ordem parcial. Como base para o desenvolvimento deste, foi utilizado o planejador de [Branquinho et al. 2011b]. Chamado de MEAPOP, esse planejador constrói o plano de ações sem estabelecer a ordem exata da execução de suas ações. O planejamento de ordem parcial consiste em qualquer algoritmo de planejamento que possa inserir duas ações em um plano sem especificar qual delas deve ser executada primeiro [Russell e Norvig 2003].

O correto entendimento do funcionamento do MEAPOP é importante para esta pesquisa. Isso porque foi proposto o desenvolvimento de um planejador capaz de gerar um plano e ao mesmo tempo escalonar suas ações. Tal planejador será explicado com detalhes na Seção 5.1. O MEAPOP foi escolhido como base para o desenvolvimento desse algoritmo devido à capacidade de proporcionar planos de ações que possuem flexibilidade na execução das ações e ligações dinâmicas entre essas, características que auxiliam nas tarefas do escalonador. A Figura 3.14 demonstra essa característica.

Neste problema o estado inicial é (*1 Townhall, 1 Peasant, 2 Supply, 600 Gold*) e a meta é (*1 Townhall, 3 Peasant*). Temos em A um plano sequencial gerado pelo MEA. Este plano é dado como entrada para um algoritmo de escalonamento, tendo como resultado o plano B. No plano A existe o problema que as ações *collect-gold* não aproveitam

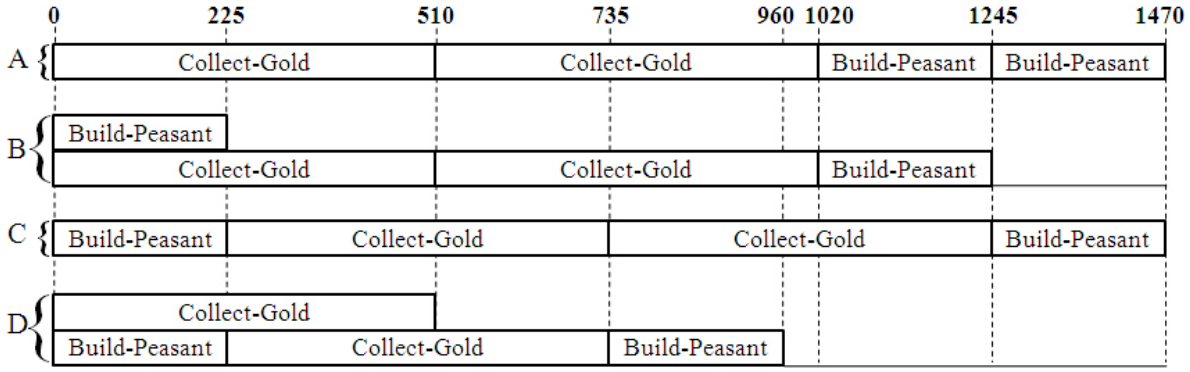

Figura 3.14: Comparação entre planos que foram escalonados utilizando planejador sequencial e planejador de ordem parcial.

o recurso *peasant*, disponibilizado pela ação *build-peasant*. Isso ocorre porque as ações *collect-gold* são escalonadas antes de *build-peasant*. Deste modo, o recurso *peasant* não é usado no escalonamento. O plano C mostra o plano parcial obtido com MEAPOP. Como a primeira ação *build-peasant* não necessita dos recursos produzidos pelas ações *collect-gold*, é possível alocar tal ação no início do plano. Quando o plano C é dado como entrada para o mesmo algoritmo de escalonamento usado em A é obtido o plano D. Neste plano o *peasant* produzido no planejamento é usado para executar uma das ações *collect-gold*. Comparando o *makespan* dos planos B e D, observamos que o melhor plano corresponde ao D.

O Algoritmo 4 mostra o pseudocódigo do MEAPOP. O parâmetro de entrada do algoritmo é a meta a ser atingida G a partir do estado inicial S. S e R são variáveis globais e tem seus atributos configurados e mantidos durante o processo de planejamento. R é uma lista que armazena os recursos já criados pelas ações que foram colocadas no plano. Recursos já criados podem ser usados para satisfazer pré-condições de outros recursos sem a necessidade de executar novas ações.

O primeiro *loop* do Algoritmo4 armazena a quantidade de recursos em S, R e G. Nessa parte, o algoritmo verifica cada recurso R_i não satisfeito pela condição $((s_i + r_i) < g_i)$ e constrói um estado intermediário usando o procedimento $SatisfazMeta(s_i, r_i, g_i, R_i, G)$. O procedimento tem por objetivo diminuir a distância entre S e G, caso os recursos de G sejam satisfeitos por S uma relação entre a meta e esse estado é estabelecida utilizando o método $succe(R, G)$ (linha 6). O algoritmo $AlocaRecursos(s_i, r_i, g_i, R_i$ (linha 7) é utilizado para alocar qualquer quantidade de recursos que possa ser usado para completar totalmente ou parcialmente a meta. Por exemplo, se no estado inicial já existem *300 Wood* e meta necessita de *600 Wood*, a quantidade já disponível é alocada para ser usada para esse estado da meta.

Se a quantidade de recursos disponíveis em S mais R não forem suficientes para satisfazer a meta G, novas ações são determinadas para satisfazer essa diferença. Para isso, é

Algoritmo 4 MeaPop(G)

```
1: satisfazG ← true
2: for all resource R_i do
3:    s_i ← quantidade de R_i em S
4:    r_i ← quantidade de R_i em R
5:    g_i ← quantidade de R_i em G
6:    succe(R, G)
7:    AlocaRecursos(s_i, r_i, g_i, R_i)
8:    if (s_i + r_i) < g_i then
9:       satisfazG ← false
10:      SatisfazMeta(s_i, r_i, g_i, R_i, G)
11:   else if s_i < g_i then
12:      satisfazG ← false
13:   end if
14: end for
15: if satisfazG then
16:   succe(S, G)
17: end if
18: return S
```

utilizado o algoritmo $SatisfazMeta(s_i, r_i, g_i, R_i, G)$ (linha 10). Este procedimento escolhe uma ação capaz de produzir o recurso R_i que deve ser satisfeito. Em seguida é realizado o cálculo de quantas vezes a ação deve ser feita para satisfazer a respectiva meta. O Algoritmo5 apresenta o pseudocódigo do $SatifazMeta$.

Algoritmo 5 SatisfazMeta(s_i, r_i, g_i, R_i, G)

```
1: A_i ← ação que produz R_i
2: α ← unidades de R_i produzidas por A_i
3: k ← ceil((g_i − s_i − r_i)/α)
4: if R_i se for recurso do tipo "consume" then
5:    A ← ConstroiMeta(k, A_i)
6:    succe(A, G)
7:    MeaPop(A)
8:    R.add(A, α ∗ k + s_i + r_i − g_i)
9: else
10:   for i ← 1, k do
11:      A ← ConstroiMeta(1, A_i)
12:      succ(A, G)
13:      MeaPop(A)
14:      R.add(A, α)
15:   end for
16: end if
```

A escolha de como as ações devem ser criadas dentro do plano é feita de acordo com o tipo do recurso R_i. Se R_i é *consume* será criado um elemento no plano representando um conjunto de k ações A_i. Caso contrário, serão criadas k ações A_i. Para realizar essa tarefa é usada a função *ConstroiMeta* (linhas 5 e 11 do Algoritmo 5), que determina as

pré-condições necessárias para as k ações serem executadas, armazenando este resultado em A. O valor A estabelecido é tratado como sendo um estado que faz parte do plano e deve ser satisfeito.

Para cada elemento do plano criado é construída a relação deste elemento com a meta através do método *succe(R, G)* (linhas 6 e 12). Posteriormente é realizada uma chamada do procedimento *MeaPop* (linha 7) para satisfazer as pré-condições das k ações. Ao final do *MeaPop*, os recursos são adicionados na variável R. Se forem do tipo *consume* é considerada apenas a quantidade excedente, ou seja, a diferença entre a meta e os recursos criados ($\alpha * k + s\_i + r\_i - g\_i$).

Outros planejadores também foram considerados para serem utilizados nessa pesquisa e como fonte de comparação com nossas técnicas. Os planejadores [Do e S. 2003] e [Gerevini e Serina 2003] são duas abordagens bastante conhecidas na área de planejamento. No entanto, não puderam ser utilizados nessa pesquisa, pois teriam de ser adaptados para o nosso problema, modificando as principais partes de seus funcionamentos. Isso, porque ambos possuem foco em uma abordagem mais clássica de planejamento e escalonamento. De fato, a maioria das referências pesquisadas possui foco diferente do nosso.

3.3 Escalonamento

Como já fora dito previamente nesta pesquisa, a tarefa de escalonamento em um plano de ações visa alterar tal plano de modo que suas ações possam ser executadas em paralelo e seu *makespan* possa ser diminuído. Diversos algoritmos vêm sendo melhorados e novas estratégias são criadas para dar mais eficiência a essa tarefa. Nesta pesquisa, o escalonamento faz parte da sua segunda etapa, onde obteve grande contribuição na melhora dos resultados que já haviam sido alcançados. Essa melhoria e os demais detalhes serão apresentado no Capítulo 5.

O algoritmo de escalonamento proposto por [Chan et al. 2007] foi usado como fonte de inspiração para o desenvolvimento do escalonador proposto nesta pesquisa. Em seu funcionamento, o algoritmo procura para cada ação A_i mover o seu tempo de início para o mais cedo possível, de tal forma que suas pré-condições estejam satisfeitas. Para uma ação A_i que começa no tempo t_i^s tem-se que $R^+(t_i^s)$ é o estado de recursos no tempo t_i^s após os efeitos adicionados ao estado do jogo para todas as ações que terminam em t_i^s, sendo que $R^-(t_i^s)$ é o estado de recursos do jogo antes que os efeitos sejam adicionados. O algoritmo busca por uma combinação onde as condições prévias de A_i sejam satisfeitas por $R^+(t^S)$,e não por $R^-(t^S)$, ou seja, a satisfação das condições de A_i é devido às ações que terminam com tempo t^s. Quando isso ocorre, o plano remarca a ação A_i com tempo de execução inicial igual ao no tempo t^s. Em seguida as demais ações do plano vão tentar ser antecipadas também. O pseudocódigo deste método é apresentado pelo Algoritmo 6.

Algoritmo 6 Schedule(*Plano*)

1: **for** $i \leftarrow 1, ..., k$ **do**
2: $t^s \leftarrow t_i^s$
3: $R^-(t^s) \leftarrow$ estado de recursos antes dos efeitos das ações que terminam em t são adicionados.
4: **while** pré-condições de A_i são satisfeitas por $R^-(t^s)$ **do**
5: $t^s \leftarrow$ época (início e/ou término de uma ação) de decisão anterior.
6: **end while**
7: $Plano \leftarrow Plano - (A_i, t_i^s, t_i^e) + (A_i, t^s, t_i^e - t_i^s + t^s)$
8: **end for**

Quando escalonamos uma ação é necessário garantir que cada ação entre o novo tempo de início e o antigo tempo de término continua sendo executada. O princípio que norteia um eficiente escalonamento é de que sempre podemos programar uma ação antes de uma ação anterior que ela possua. Isso claro, se ambas não consumirem ou pegarem emprestadas os mesmos recursos.

Vamos considerar um exemplo onde temos um par de ações A e B, sendo que B será escalonada. A ação A está antes de B no plano de ações com os recursos *consume* e *borrow* em R e assumindo que está prestes a escalonar B. Consideramos que o plano de ações é um plano válido. Primeiramente, se A e B são adjacentes, o estado antes de A possui os recursos suficientes em R para A e B serem executadas. Assim, neste estado A e B podem ser executadas em paralelo, ou se possível, B pode ser escalonada antes de A. Entretanto, se A e B estão separadas por quaisquer ações que produzem R a fim de satisfazer as pré-condições de B, então, o nosso processo não mudaria B antes dos efeitos dessas ações. Embora este procedimento não garanta a produção de um plano com o *makespan* ótimo, ele é rápido (em geral o tempo é polinomial de ordem quadrática no número de ações) e funciona bem na prática.

A seguir será apresentado um exemplo do processo de escalonamento utilizando o Algoritmo 6 . Nesse, será utilizado o plano de ações obtido na Figura 3.13. O domínio de ações do jogo *Wargus* será utilizado neste exemplo. O objetivo é paralelizar ao máximo as ações desse plano.

Figura 3.15: Tentativa de escalonar a ação *collect-gold* no tempo 0.

O plano é percorrido de forma linear em relação ao posicionamento das ações. O

primeiro passo do escalonamento, exibido na Figura 3.15, descreve a tentativa de escalonar a ação *collect-gold* no tempo 0. Os recursos no tempo 0 (*1 Townhall*, *3 Peasant*, *0 Barracks*, *0 Supply*, *0 Footman*, *400 Gold*, *200 Wood*) satisfazem as pré-condições da ação *collect-gold* (**borrow***: 1 Peasant* **require***: 1 Townhall*). Não existe nenhum tempo anterior ao 0, sendo assim, *collect-gold* é mantida neste tempo.

No passo seguinte, a *collect-wood* é escolhida para ser escalonada. A ação é primeiramente posicionada no tempo mais adiante, que é igual a 510 (Figura 3.16). Como neste tempo a ação tem suas pré-condições satisfeitas, é feita a tentativa de posicioná-la no tempo anterior, igual a 0. Neste caso, a ação *collect-wood* tem suas pré-condições satisfeitas neste tempo também e como é o menor tempo possível ela é mantida nele. Agora temos duas ações em paralelo, que são mostradas na Figura 3.17).

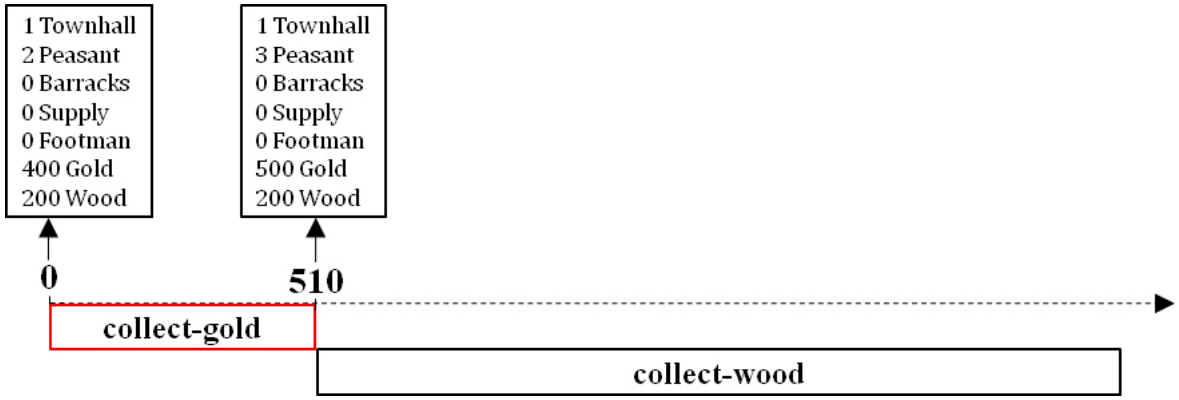

Figura 3.16: Tentativa de escalonar a ação *collect-wood* no tempo 510.

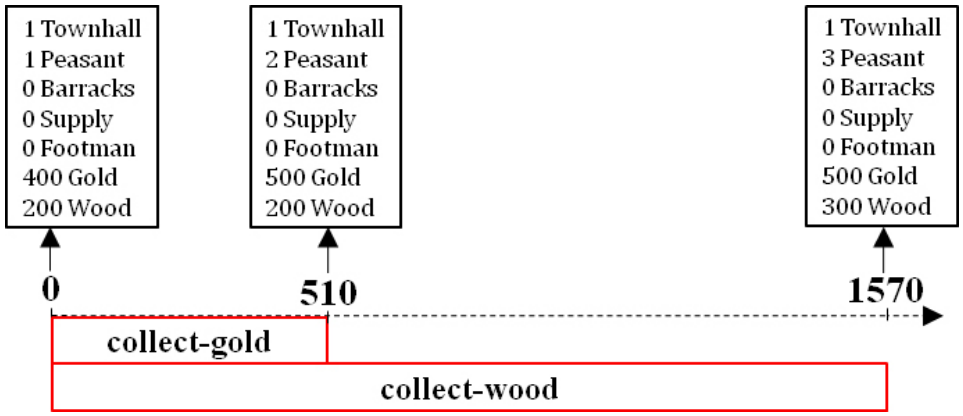

Figura 3.17: A ação *collect-wood* é escalonada no tempo 0.

Agora temos a escolha da ação *build-supply* para o escalonamento. A verificação de satisfação das pré-condições começa no tempo 1570 (Figura 3.18), que é o mais adiante no plano. Neste tempo ela pode ser mantida, deste modo, tentamos escaloná-la no tempo anterior 510. Entretanto, no tempo 510 não existem recursos suficientes que satisfaçam suas pré-condições (Figura 3.19). Com isso, a ação *build-supply* é escalonada no tempo 1570, que foi exatamente o último tempo em que teve suas pré-condições satisfeitas (Figura 3.20).

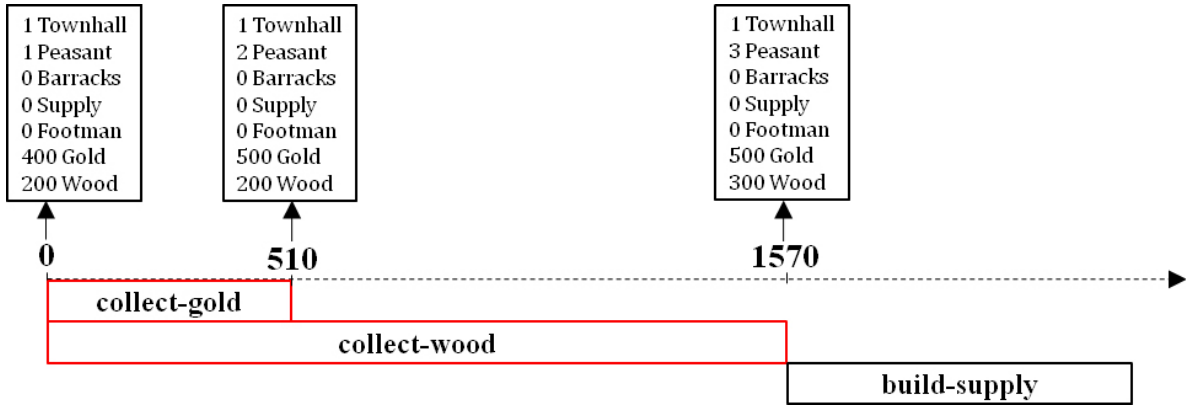

Figura 3.18: Tentativa de escalonar a ação *build-supply* no tempo 1570.

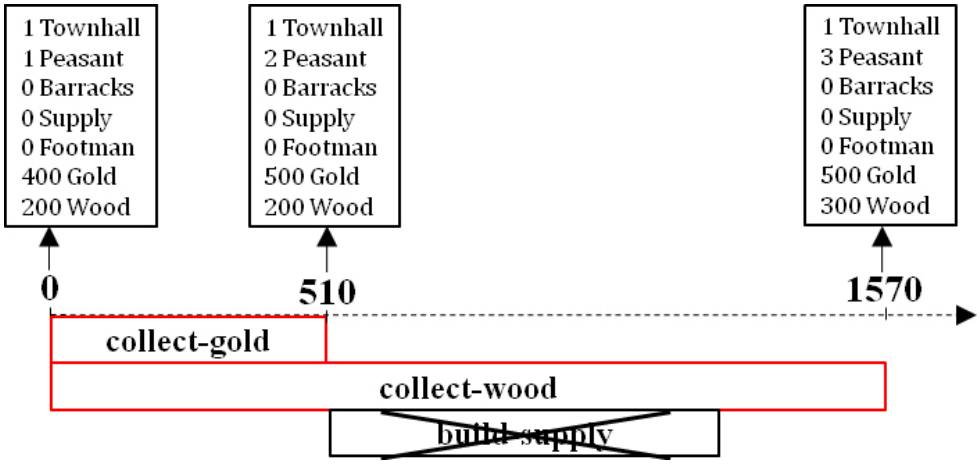

Figura 3.19: Tentativa de escalonar a ação *build-supply* no tempo 510.

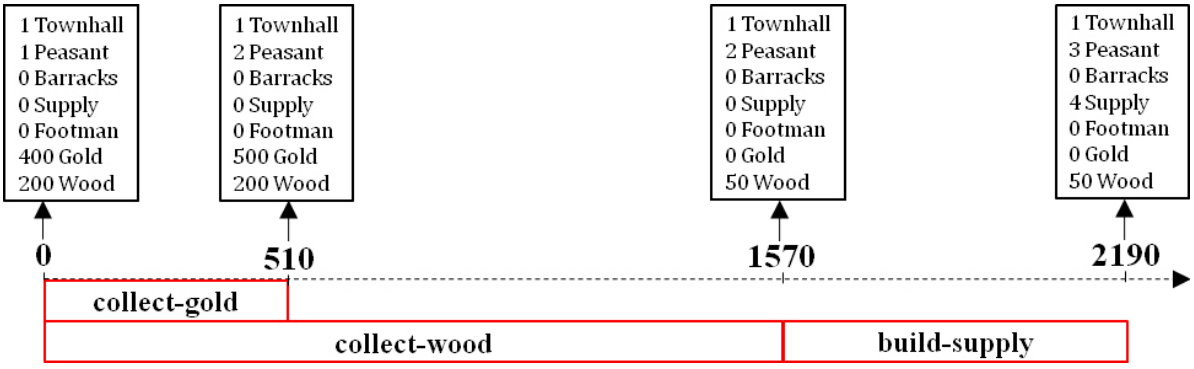

Figura 3.20: A ação *build-supply* é escalonada no tempo 1570.

As demais ações do plano irão passar pelo processo de escalonamento. O resultado final é apresentado na Figura 3.21. O *makespan* do plano sequencial e do plano escalonado são, respectivamente, 4915 e 2415. Isso demonstra que um plano pode ter melhor eficiência se for aplicado um procedimento de escalonamento.

O escalonador proposto por [Branquinho et al. 2011b], consiste no uso do algoritmo *SLA**, adaptado para tal tarefa. O Algoritmo 7 mostra o pseudocódigo do escalonador utilizando SLA*. Será feita uma breve descrição desse. O algoritmo trabalha utilizando

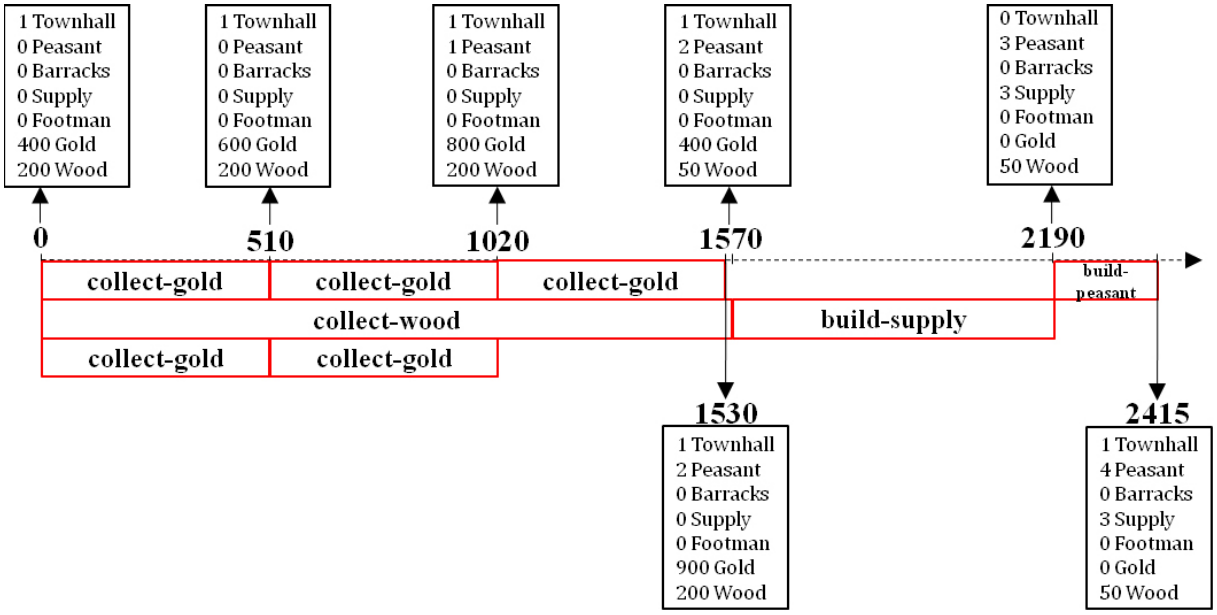

Figura 3.21: Resultado do escalonamento do plano da Figura 3.13.

atividades. Uma atividade pode conter uma ou várias ações do mesmo tipo, sendo que o escalonamento considera uma atividade de cada vez. Por exemplo, se para um dado recurso for necessário executar a ação *collect-gold* três vezes, essas ações juntas irão formar uma atividade.

Algoritmo 7 SLA*$(s_{inicial}, s_{meta})$

1: $PILHA.add(s_{atual})$
2: **loop**
3: $s_{atual} \leftarrow PILHA.top()$
4: **if** $s_{atual} = s_{meta}$ **then**
5: **return** $PILHA$
6: **end if**
7: $s' \leftarrow arg\ min_{s'' \in succe(s_{atual})}(k(s_{atual}, s'') + h(s''))$
8: **if** $h(s_{atual}) \geq k(s_{atual}, s') + h(s')$ **then**
9: $PILHA.add(s')$
10: **else**
11: $h(s_{atual}) \leftarrow k(s_{atual}, s') + h(s')$
12: **if** $s_{atual} \neq s_{inicial}$ **then**
13: $PILHA.remove()$
14: **end if**
15: **end if**
16: **end loop**

Cada estado do Algoritmo 7 consiste em um conjunto de ações que estão em progresso, ou seja, estão sendo escalonadas. O algoritmo possui uma função responsável por encontrar estados sucessores (linha 7). Essa função chamada de $EstadosSucessores(Recursos, Atividades)$, cria os estados sucessores a partir do estado atual. Essa função recebe uma lista de recursos que estão disponíveis ($Recursos$) e uma lista contendo um conjunto de

ações (*Atividades*), que são consideradas pelo algoritmo como sendo uma atividade.

Cada estado que é criado pelo algoritmo é armazenado em uma variável que é responsável por armazenar todos os estados que forem criados. A função responsável por isso é $AdicionaEstados(A_j, S, Recursos, Atividades)$, que cria cada estado do algoritmo e o adiciona à variável que os armazena. Como parâmetros, além das listas de recursos e atividades, o algoritmo recebe a atividade atual A_j e o número de ações que já foram escalonadas em estados anteriores de acordo com suas respectivas atividades S. A função então cria um novo estado e em seguida o algoritmo continua o escalonamento das atividades restantes. O SLA* possui a vantagem de encontrar a solução ótima para o escalonamento de um plano de ações. Entretanto, o tempo gasto para tal tarefa é alto e dependendo da situação pode tornar-se inviável para um ambiente de tempo real.

Outros trabalhos também foram considerados para essa pesquisa. O trabalho de [Churchil e Buro 2011] foca na produção de recursos com escalonamento, nele é utilizado uma abordagem baseada em heurísticas e algoritmos como *Fast-Foward* (FF). O objetivo do seu trabalho é fazer frente às referências já citadas como [Chan et al. 2007] e [Branquinho et al. 2011b], os resultados obtidos em relação a *makespan* são bons quando comparados com esses. Já o trabalho de [Andrew Trusty 2008] busca diminuir o *makespan* de planos baseado na técnica de *Case Based Reasoning* e algoritmo genético, sua abordagem baseia-se em utilizar planos intermediários para inicializar o algoritmo de busca e encontrar soluções melhores. Entretanto, os dois trabalhos focam na mesma tarefa das referências usadas nesta pesquisa e ambos conseguem resultados próximos não superando as mesmas.

3.4 Balanceamento de Parâmetros

O trabalho que inspirou o desenvolvimento desta pesquisa aqui apresentada é o de [Fayard 2005]. Nesse, o autor propõe o uso de busca estocástica para atingir uma determinada quantidade de recursos em um determinado tempo. O objetivo é obter diferentes quantidades de recursos para as diversas classes presentes em um jogo RTS. Uma vez que esses recursos são obtidos, é feita uma série de comparações em relação a essas quantidades, para verificar se as diferentes classes possuem quantidades de recursos semelhantes em suas produções. Essa verificação visa descobrir se essas diferentes classes do jogo possuem seus recursos balanceados, ou seja, todas as classes do jogo apresentam produtividade semelhante em relação a produção de recursos.

No trabalho de [Fayard 2005], caso seja feita a geração de recursos para uma dada classe com um determinado tempo limite, e o resultado for superior ao de outra classe, significa que essas estão desbalanceadas. Isso significa que uma classe consegue produzir recursos com mais rapidez ou eficácia que outra. Em seu trabalho, o autor propõe que sua abordagem de produção de recursos pode ser usada para verificar se as classes de um

dado jogo de RTS estão com sua produção de recursos equilibrada. Se houver algum tipo de desequilíbrio entre as classes, é preciso alterar os diferentes parâmetros que os recursos dessas possuem, tais como: Quantidade de recursos consumida, tempo de construção, quantidade de recursos que produz, etc.

Em sua abordagem de balanceamento de parâmetros, o autor também utiliza o SA como algoritmo de busca estocástica. Nele, o SA é responsável por inserir, retirar e trocar ações de lugar para que sejam gerados novos recursos. O intuito é de conseguir obter diferentes quantidades de recursos dependendo do tempo limite. O tempo limite é um limitador que o autor usa para determinar o quanto o SA pode buscar por uma quantidade de recursos. Essa quantidade de recursos é também um plano de ações, assim como já estamos usando o plano de ações como uma forma de se obter recursos dentro de um jogo. O tempo limite usado pelo autor não é relativo ao *makespan* do plano, e sim um tempo (*runtime*) em minutos que o algoritmo pode ficar sendo executado.

O domínio de recursos utilizado por [Fayard 2005] é do jogo *StrarCraft* o qual também é utilizado nesta pesquisa, representado de forma parcial pela Figura 3.11. O autor justifica o uso do *StarCraft* pelo fato deste ser um dos jogos RTS com maior quantidade de recursos e quantidade de pré-condições entre suas ações. Com o objetivo de estender sua abordagem para outros jogos do tipo estratégia em tempo real, ele cita que obter sucesso dentro do domínio do *StarCraft* qualifica sua abordagem para tal objetivo.

No trabalho de [Fayard 2005], o autor não deixa muito claro os detalhes relativos a implementação de sua abordagem, nem tão pouco faz demonstrações de algoritmos e as técnicas utilizadas. Entretanto, em seus resultados foi visto que a produção de recursos funcionou bem, onde para diferentes classes foi possível obter diferentes quantidades de recursos utilizando planos de ações válidos, ou seja, nenhum recurso foi produzido sem que todas suas pré-condições fossem totalmente satisfeitas. Tais resultados foram um dos motivadores para o desenvolvimento da abordagem aqui proposta. Mas, o que de fato inspirou a construção dessa pesquisa foi o uso de busca estocástica para alcançar recursos em um jogo RTS.

Em relação aos experimentos do trabalho de [Fayard 2005], em um deles o autor demonstra que entre as classes *Terran* e *Protoss* do jogo *StarCraft* existe um desequilíbrio entre seus recursos. A classe *Terran* conseguiu obter uma quantidade de recursos maior em relação à quantidade obtida pela classe *Protoss*. De fato, tal desequilíbrio reflete diretamente sobre o usuário do jogo, que ao optar por uma classe pode ter seu desempenho reduzido em relação ao seu nível como jogador, caso opte por uma classe com recursos desbalanceados.

Vamos a um exemplo que ilustre o balanceamento de parâmetros. Neste, SA foi utilizado para gerar 6 diferentes quantidades de recursos entre as classes *Terran* e *Zerg*. Será feita uma média da quantidade de recursos gerados por essas classes em todas as execuções, para ter um valor médio que represente os recursos gerados. Os recursos

possuem um peso em relação ao seu tipo, quanto mais recursos forem necessários para se construir um determinado recurso melhor avaliado ele será. Esse peso ajuda a avaliar a quantidade de recursos que uma classe pode produzir, não usando somente o critério de quantidade, que pode gerar resultados que não refletem corretamente as características de cada recurso. A Figura 3.22 mostra a médias das 6 execuções relativa as duas classes.

Classe Terran

Qtd de Recursos	Peso do Recurso
8	6
13	5
9	4
4	3
10	2
9	1
Média dos pesos	32

Classe Zerg

Qtd de Recursos	Peso do Recurso
6	6
17	5
11	4
1	3
8	2
8	1
Média dos pesos	32

Figura 3.22: Média de recursos encontrados e seus respectivos pesos na abordagem de balanceamento de parâmetros.

Na Figura 3.22, é mostrado a média relativa à quantidade de recursos encontrados nas 6 execuções do SA para as classes *Terran* e *Zerg*. Para cada quantidade de recursos é exibido o peso atribuído a essa. O peso varia em relação ao custo para construir cada recurso, onde quanto mais dispendioso for ele em relação ao consumo de outros recursos maior será o seu peso. Recursos mais poderosos ou mais importantes para o desenvolvimento de novos recursos costumam ser mais caros, ou seja, consomem mais recursos na hora de serem produzidos. Nesse experimento, foi constatado que as duas classes possuem seus recursos balanceados, pois mesmo com diferentes quantidades de recursos produzidos a média final foi à mesma. Caso o resultado tivesse apontado que uma das classes estava inferior à outra, os parâmetros de ambas deveriam ser alterados e novos testes executados para verificar se o equilíbrio foi alcançado. Neste experimento, para cada execução o tempo limite de execução para o SA foi de 20 minutos. Uma das desvantagens dessa abordagem é o tempo gasto pelo SA em suas execuções, de tal forma que esse tempo a impede de ser utilizada no ambiente de tempo real do jogo.

3.5 Abordagem para Escolha de Metas

Esta pesquisa aqui apresentada descreve a abordagem desenvolvida pelo autor para resolver o problema de escolha de metas em jogos RTS. Tal problema, foi identificado

durante a análise dos principais trabalhos já citados neste capítulo. Os trabalhos de [Chan et al. 2007] e [Branquinho et al. 2011b] deram consistência para que pudesse ser vislumbrado um espaço para o desenvolvimento de uma nova abordagem para jogos RTS, além de base técnica para o desenvolvimento da mesma. Já o trabalho de [Fayard 2005] serviu para motivar o uso de uma técnica de busca como elemento chave para a abordagem desenvolvida.

Nos trabalhos de [Chan et al. 2007] e [Branquinho et al. 2011b] é explorado o uso de diferentes planejadores e algoritmos de escalonamento. O objetivo é encontrar um plano de ações que conduz o jogo de um estado inicial de recursos até um estado final estipulado, onde o último produz os recursos que definem a meta a ser atingida. No entanto, nessas abordagens quando é feito o planejamento para encontrar tal plano de ações, temos que o estado inicial de recursos do planejamento é o estado atual em que o jogo se encontra, mas o estado final, ou seja, os recursos que serão produzidos formando uma meta a ser atingida, não tem uma definição formal de seu estado de recursos. Normalmente, esses recursos que serão produzidos são definidos com quantidades aleatórias ou de forma subjetiva.

Em todos os exemplos já apresentados neste capítulo, é possível perceber que as metas a serem alcançadas são definidas de forma não criteriosa nos trabalhos que envolvem planejameto para jogos RTS. Com isso, é proposta aqui uma abordagem para eliminar essa deficiência, onde essa consegue determinar quantos e quais recursos devem ser atingidos através de maximização da produção de recursos. Para isso, é utilizado busca e planejamento para desenvolver os planos de ações que produzem os recursos que são definidos e estipulam metas baseadas em critérios. O objetivo é desenvolver um planejamento para jogos RTS, onde as metas que são atingidas durante o jogo não sejam escolhidas sem critério ou qualidade para o planejamento. O algoritmo *Simulated Aneealing* é utilizado na busca e foi adaptado para operar de acordo com o ambiente de tempo real de jogos e técnicas de planejamento. A abordagem aqui proposta executa o planejamento completo para o problema, ou seja, para estipular uma meta é feita uma busca que executa planejamento sobre a produção de recursos, maximizando essa produção e retornando ao fim um plano de ações com todos os atributos já configurados, onde é necessária apenas a execução de suas ações dentro do ambiente do jogo.

Essa pesquisa utiliza o SA baseado nos resultados obtidos por [Fayard 2005], onde esse utiliza o algoritmo no domínio de jogos RTS. Contudo, nesse trabalho o SA não é utilizado dentro do jogo, e sim de forma *off-line*, sendo executado fora do ambiente de tempo real. Em nosso trabalho o SA foi adaptado e incrementado com técnicas que possibilitaram o seu uso de acordo com as restrições de um jogo RTS, como tempo de resposta e boas soluções. Com isso, o SA é utilizado para, a partir de um plano de ações inicial, gerar novos planos até que seja encontrado um plano de ações com qualidade que possa ser considerado uma meta, onde suas ações irão produzir os recursos necessários para atingir essa meta.

O plano de ações inicial que é passado ao SA é um plano contendo diversas ações que geram uma quantidade aleatória de recursos. Tal plano é gerado através de um planejador. Para a geração desse plano inicial, é estipulado um *makespan* limite (tempo limite) que limita o tempo máximo de execução das ações do plano dentro do jogo. Assim, o plano de ações inicial do SA é gerado baseado no tempo de execução das ações, ou seja, no tempo em que elas levam para ser executadas dentro do jogo. Esse tempo pode variar entre segundos e minutos. Por exemplo, se for preciso gerar um plano com 2 minutos de planejamento basta definir esse tempo limite e será gerado um plano que alcança uma quantidade aleatória de recursos em 2 minutos. Assim, quando o SA receber tal plano, ele irá gerar novos planos mantendo essa restrição de tempo até encontrar um novo plano que maximize os recursos produzidos em 2 minutos. Podemos considerar o plano inicial como uma possível solução.

Na abordagem aqui proposta, o plano inicial é gerado a partir do estado inicial de recursos e atinge uma quantidade aleatória desses, baseado em um tempo limite para serem executados dentro do jogo. Assim, os planejadores usados foram alterados para operarem sob essas condições. A pesquisa foi dividida em duas etapas, na primeira foi proposta uma metodologia sequencial com um planejador sequencial. Na segunda etapa um planejador parcial com escalonamento foi proposto. Ambos foram construídos com base nas referências já apresentadas. O domínio de ações utilizado aqui é o do jogo *StarCraft*, que foi escolhido devido a suas características já mencionadas, como por exemplo a quantidade de recursos no jogo. Enquanto o domínio do *Wargus* possui apenas 7 recursos o *Starcraft* possui mais de 40 por classe.

Uma vez desenvolvido pelo planejador, o plano de ações é passado para o SA. Quando o SA encontrar um plano de ações final que possui seus recursos maximizados baseado em um dado critério, esse plano representará o estado final cujo os recursos produzidos estipulam a meta a ser atingida. Entretanto, o SA não só especifica a meta a ser atingida, pois foram desenvolvidas técnicas para que além dos recursos a serem atingidos, o algoritmo possa retornar o plano de ações que atinge essa meta produzindo tais recursos. As técnicas e critérios usados na maximização da produção de recursos serão apresentados nas seções a seguir.

As referências focam em construir um plano de ações para atingir uma meta subjetiva como em [Chan et al. 2007, Branquinho et al. 2011b], ou para atingir uma meta baseada em um tempo de execução [Fayard 2005]. A abordagem aqui proposta busca por um plano de ações baseado em um dado critério proposto para a produção de recursos e que será maximizado no decorrer deste trabalho sempre restrito a um tempo limite. Essa abordagem, na verdade, apresenta semelhanças na forma como um jogador humano planeja e constrói seus recursos dentro do jogo, sendo comparado com tal nos experimentos que serão apresentados ao final desta pesquisa. As principais diferenças com as referências citadas serão apresentadas nos próximos capítulos e deixam claro que a abordagem aqui

proposta visa preencher uma lacuna presente no campo de planejamento para jogos RTS. Tais características tornam difíceis comparações entre os resultados obtidos aqui e os obtidos nos trabalhos já mencionados.

Capítulo 4

Uso de Simulated Annealing para Produção de Recursos em Tempo Real

Neste capítulo será descrita a metodologia do *Simulated Annealing* dentro da abordagem para escolha de metas em jogos RTS aqui proposta. O algoritmo sofreu adaptações e recebeu técnicas que foram desenvolvidas para que conseguisse operar sobre o domínio de tempo real. Tais adaptações e técnicas serão descritas no decorrer deste capítulo. Assim, o algoritmo conseguiu cumprir todas as restrições que esse domínio impõe. Uma vez escolhido como algoritmo de busca a ser utilizado nessa pesquisa, o SA tem grande importância para os resultados obtidos e seu funcionamento será enfatizado aqui.

SA é uma meta heurística que pertence a classe dos algoritmos de busca local [Aarts e Korst 1989]. Ele foi escolhido como o algoritmo de busca devido à robustez em seu funcionamento, pelos resultados obtidos no trabalho de [Fayard 2005] onde o algoritmo foi utilizado, além de apresentar bons resultados durante os testes iniciais. O SA recebe como entrada uma possível solução para o problema, nesse caso um plano de ações desenvolvido por algum dos planejadores que serão descritos com detalhes nos próximos capítulos. A função geradora de vizinhos executa operações de troca de posições ou de inserção de novas ações no plano para indicar uma nova solução.

No algoritmo, a avaliação de uma possível solução é executada pela função objetivo. Essa é responsável por contar os pontos de ataque das ações que estão no plano e geram recursos, para medir a força do exército que será gerado. Pontos de ataque são os atributos presentes em cada recurso do jogo. Esses determinam o poder de ataque de um determinado recurso. Dessa forma, é possível encontrar e determinar quais estados (planos) possuem maior força de ataque. Se um novo estado é melhor avaliado do que o estado anterior ele torna-se a solução atual. Mesmo que um novo estado não seja melhor avaliado do que o anterior, ainda existe uma chance dele ser aceito como solução. O Algoritmo 8 mostra o pseudocódigo do SA.

Algoritmo 8 SA($G_{inicial}$, P, M, N, μ)

```
1: G_atual ← G_inicial
2: G_melhor ← G_atual
3: T_0 ← TempInicial()
4: j ← 0
5: while T_0 > 0 or j < N do
6:    i ← 1
7:    nSucesso ← 0
8:    while i < M do
9:       G_vizinho ← GeraVizinho(G_atual)
10:      Δr ← AvaliaPlano(G_atual) - AvaliaPlano(G_vizinho)
11:      if Δr < 0 ou Random() < PrbPlano() then
12:         G_atual ← G_vizinho
13:         nSucesso ← nSucesso + 1
14:      else
15:         j ← j + 1
16:      end if
17:      i ← i + 1
18:   end while
19:   if nSucesso ≥ P then
20:      T_0 ← α * T_0
21:   end if
22:   if AvaliaPlano(G_atual) > AvaliaPlano(G_melhor) then
23:      G_melhor ← G_atual
24:   end if
25: end while
26: return G_melhor
```

Solução Inicial

O algoritmo recebe uma solução inicial (plano de ações) e executa operações sobre a mesma. Essa solução $G_{inicial}$ é gerada por um dos planejadores que serão descritos nos próximos capítulos. Os recursos que são desenvolvidos pelas ações desse plano são escolhidos de forma aleatória e tem seu *makespan* restrito ao tempo limite determinado para a meta. Nesta abordagem esse tempo limite que é usado para construir o plano inicial varia entre intervalos de 2 a 5 minutos. A ideia ao utilizar esses valores é induzir o SA a trabalhar sempre com planos contendo recursos variados, para evitar que o algoritmo fique tendencioso a algum valor de função objetivo na solução inicial. Assim, ele consegue gerar boas soluções mesmo com um plano inicial que não tenha alto valor de função objetivo. Uma vez que o tempo limite é alterado, o número de ações que compõe o plano inicial também é alterado.

Mecanismo de Resfriamento

A temperatura inicial do SA deve ser definida com um bom critério, pois essa tem influência direta no tempo de execução (*runtime*) do algoritmo. Na abordagem proposta,

essa temperatura T_0 é definida pela função $TempInicial()$ (linha 2), que baseia-se no número médio de ações que contém o plano de entrada do SA. O tempo limite afeta diretamente a quantidade média de ações que o plano inicial terá. Ajustar esse tempo limite para cada meta significa que é possível gerar planos com quantidades menores de ações. Como já mencionado os planos de ações que específicam as metas a serem atingidas possuem esse tempo entre 2 e 5 minutos. O objetivo é simular o comportamento de um jogador humano, já que esse planeja com base em intervalos curtos de tempo que proporcionam reatividade caso o ambiente do jogo mude. Portanto, a temperatura inicial é estabelecida com cerca de cinco vezes a quantidade de ações que possui a solução inicial, sendo esse um valor apropriado para o domínio.

Uma vez definido o critério para estimar o valor da temperatura inicial, é preciso fazer o mesmo para o mecanismo de resfriamento. O resfriamento é feito por um valor que decai a temperatura após uma quantidade de iterações. Através de experimentos foi identificado que um valor próximo de alguns décimos do valor da temperatura inicial é um bom número. Utilizar métodos que tem por base funções logarítmica não é uma boa opção. Essas deixam o algoritmo muito lento devido ao número de iterações e tempo de cálculo que aumentam, que é um comportamento inapropriado para um ambiente de jogos RTS.

Foi desenvolvido um método adaptativo de redução da temperatura. A temperatura relativa a T_{i+1} diminui por um fator α (linha 19) se para um dado número de iterações P os novos planos gerados são sempre aceitos. O valor de α é baixo e próximo de 1. Um valor muito baixo pode fazer com que algoritmo demore muito para convergir devido ao aumento exagerado de iterações. Já um valor muito alto faz com o mesmo não converta caindo em pontos específicos do espaço de busca, de onde não consegue sair. Assim, P deve ser grande o bastante para explorar as possibilidades de operações no plano. Baseado nos experimentos realizados foi determinado que valores entre 15 e 20 são boas representações para P.

$$T_{i+1} = \begin{cases} T_i & \text{se o número de planos aceitos} < P. \\ \alpha.T_i & \text{se o número de planos aceitos } \geq P. \end{cases}$$

A probabilidade de um novo plano ser aceito caso ele não seja melhor avaliado que o anterior é dado pela equação 4.1:

$$1 - 0.5 \tanh \left[2\Delta r(v_i - v_{i+1})T_{i+1}\right] \tag{4.1}$$

Na equação 4.1, v_i é o número de ações viáveis que o plano possui na iteração atual. A variável Δr é o fator de derivação, que é obtido pela diferença entre os valores da função objetivo (pontos de ataque) do plano atual e do novo plano gerado (linha 9 do Algoritmo 8). Esse valor é então dividido pela temperatura.

Utilizando a equação 4.1 o algoritmo comporta-se de forma diferente dependendo do valor da temperatura em que ele se encontra. A função responsável por calcular a probabilidade de aceitação é a $PrbPlano()$ (linha 9). Para ser aceito, seu valor deve ser maior do que o gerado pela função $Random()$ (linha 9), que produz um valor aleatório entre 0 e 1. O parâmetro M determina quantos vizinhos irão ser gerados antes da temperatura diminuir (linha 7). O seu valor também deve seguir a premissa de possibilitar uma boa busca pelo espaço de planos, dado a temperatura em que o algoritmo encontra-se no momento. Para o domínio de jogos RTS através dos experimentos foi determinado o valor médio de 30 para M.

Função Geradora de Vizinhos

Existem diversas maneiras de gerar um novo plano vizinho $G_{vizinho}$ a partir de uma atual solução G_{atual} no SA. Essa etapa é essencial para o SA conseguir boa convergência, pois a forma como o plano é alterado deve ser compatível com as características do domínio de um jogo RTS. Tais alterações devem ser feitas através de pequenas mudanças na solução, onde essas devem explorar diferentes partes do espaço de soluções. A seguir serão descritos quatro possíveis mecanismos para essa tarefa, dada a abordagem aqui proposta.

A. Duas ações são aleatoriamente escolhidas no plano e trocadas de lugar, uma assumindo a posição da outra. Quando isso ocorre, as ações tentam ser executadas no tempo em que ação que deu lugar a elas estava sendo executada. Assim são trocadas as posições e os tempos de início e fim de execução das ações.

B. Um ação do plano é aleatoriamente escolhida e substituída por uma nova ação que é aleatoriamente escolhida entre todas as ações disponíveis no domínio do jogo.

C. Um dos seguintes movimentos é aleatoriamente escolhido: Selecionar aleatoriamente duas ações no plano e trocar suas ações de lugar ou substituir uma ação aleatoriamente escolhida no plano por uma nova ação aleatoriamente escolhida entre as disponíveis no domínio do jogo. A probabilidade de substituir deve ser maior que a de troca, porque a inserção de novas ações ajuda o algoritmo a explorar melhor o espaço de soluções.

D O mesmo mecanismo que o descrito anteriormente, no entanto a nova ação que irá entrar no plano ao invés de ser escolhida entre todas as ações disponíveis no domínio do jogo, deverá ser escolhida entre as ações que estão disponíveis no próprio plano de ações.

O mecanismo A é útil quando as ações do plano são conhecidas ou têm-se algum conhecimento prévio delas pelo algoritmo, de modo a avaliar a pertinência delas para o

plano que será alterado. Tais informações não são coletadas ou mantidas pelo algoritmo. Dessa forma, o plano inicial deve ser construído com uma função objetivo de alto valor para que sejam feitas boas permutações. Isso faz com que o algoritmo fique com uma implementação tendenciosa, uma situação que será evitada neste trabalho. O mecanismo B consegue retornar bons resultados devido à inserção de novas ações no plano, o que explora bem a possiblidade de novos recursos. Mas, o mecanismo C é de longe o que apresenta os melhores resultados, pois ele combina duas estratégias de exploração. Com ele, é possível explorar diversos lugares dentro do espaço de soluções, sem perder o foco de busca local do algoritmo. O mecanismo D possui o mesmo problema do mecanismo A, ele necessita de conhecimento prévio das ações que compõe o plano.

Quando um novo plano é gerado pela função $GeraVizinho(G)$ (linha 8), os verificadores de consistência são usados para gerenciar e avaliar as mudanças que irão ocorrer no plano. Esses serão apresentados nos próximos capítulos.

Critério de Parada

O critério de parada é responsável por interromper a execução do algoritmo, quando uma execução torna-se inadequada ou excede algum limite imposto. Foram definidas duas condições que fazem com que o SA interrompa seu funcionamento. A primeira é quando nenhum progresso é feito em relação à aceitação de novos planos gerados depois de um número N de iterações. Isso ocorre geralmente quando o algoritmo já se encontra em baixas temperaturas e uma boa solução ou a ótima já pode ter sido encontrada. Assim, é mais interessante para o algoritmo interromper sua execução retornando a solução atual como solução final da busca, já que novos planos serão rejeitados. N é um valor passado como parâmetro e como outros valores do algoritmo, deve ser estipulado através de experimentos. Aqui o valor de N é igual a 50.

A segunda condição de parada que é utilizada, é quando o total de iterações do algoritmo excede um valor Z. Nesta situação, o algoritmo irá ultrapassar um limiar de iterações totais que ele pode fazer. Quando isso ocorre o plano final é retornado pelo SA. Esse critério é útil para manter o algoritmo compatível com as exigências de tempo de jogos e sistemas que operam em tempo real. O valor de Z não permite que o algoritmo despenda mais tempo do que a média gasta por ele na maior parte das execuções. A verificação desse critério é feita pelo algoritmo sempre que a função $GeraVizinho(G_{atual})$ (linha 8) é chamada. O valor de Z é calculado baseado em experimentos, onde a média de iterações do algoritmo é medida e usada como critério de parada.

Quando um critério de parada é chamado é importante que o algoritmo consiga retornar uma solução, para que não sejam definidos valores ou variáveis nulas no algoritmo. Para isso, é utilizada uma condição (linha 21) que sempre mantém o melhor resultado encontrado até o momento na variável G_{melhor}. Assim, sempre que um critério de parada for invocado ou mesmo quando o algoritmo terminar sua execução fica garantido que a

melhor solução encontrada até o momento será retornada.

Função Objetivo

A função objetivo executa a avaliação dos planos que são gerados pelo algoritmo. Seu objetivo é calcular a força do exército gerado por cada plano. Nessa tarefa, a função faz o somatório dos pontos de ataque de cada ação viável presente no plano. Uma ação viável é aquela ação que está dentro do plano e que pode ser executada caso ele seja escolhido como solução final. Ela pode ser executada porque dentro do plano estão outras ações também viáveis que satisfazem suas pré-condições. O conceito de ação viável serve para diferenciar essas ações das demais consideradas inviáveis.

Ações inviáveis são aquelas que também estão dentro do plano de ações, mas não podem ser executadas. Na prática elas estão esperando que em futuras gerações de vizinhos sejam introduzidas novas ações que possam satisfazer suas pré-condições de forma que elas possam torna-se viáveis novamente. As soluções geradas pelo SA fazem com que determinadas ações que estão no plano tornem-se inviáveis, devido às diversas mudanças que são feitas durante esse processo. Quando um plano é avaliado ou escolhido como solução final, apenas as ações viáveis são utilizadas para compô-lo . Esse conceito será visto com detalhes nos próximos capítulos.

Algumas ações tornam-se inviáveis durante as gerações de vizinhos. De fato, a tendência é que essa situação ocorra diversas vezes durante as iterações do SA. Avaliar um plano de ações com valor 0, caso ele tenha ações inviáveis dentro dele não é uma boa escolha, porque a maioria dos planos gerados seriam descartados pelo algoritmo, impedindo-o assim percorrer o espaço de soluções de forma adequada. Outro motivo seria o fato de que as ações inviáveis não contribuem na avaliação de um plano, mas podem contribuir e ajudar na geração de novos planos caso voltem a ser viáveis novamente.

Então, a avaliação feita pela função objetivo seguindo a premissa de contar apenas ações viáveis é a melhor opção. Planos mais consistentes e com exércitos mais fortes irão ter vantagem sobre planos que não possuam essas características. Outro método utilizado é o de penalidade no momento da avaliação de um plano. Esse método utiliza um valor μ (mi) que é multiplicado pelo valor dos pontos de ataque do plano quando ele é avaliado. O valor de mi é iniciado com 1 em todas as iterações do algoritmo. Durante a geração de um vizinho esse valor decai por um valor muito baixo para cada ação que se torna inviável. O inverso também acontece para cada ação que torna-se viável novamente o valor é incrementado, no entanto nunca é maior do que 1. O valor de mi decai por uma constante baixa para que tal mecanismo não influencie de forma abrupta as escolhas do algoritmo. Ele também não é maior do que 1 para não superestimar um plano em relação a outro, caso sejam colocadas mais ações viáveis do que inviáveis nele, pois o algoritmo por si só irá detectar isso. A equação 4.2 mostra o cálculo do valor de μ. Nessa equação,

n é a quantidade ações que o plano possui, tanto viáveis como inviáveis.

$$\mu = 0.1 \text{ - } ((n/2)(n * 10)) \tag{4.2}$$

A função objetivo soma os valores de pontos de ataque de cada ação. Assim o valor de um plano será maximizado dependendo da quantidade de recursos militares que ele produz. Considerando um jogo RTS, esse valor funciona como uma heurística para a função objetivo, pois um dos fatores que mais influenciam na vitória de um jogador é a qualidade no desenvolvimento do seu exército, que é usado tanto em ataques quanto na defesa dos demais recursos.

Dentro da maioria dos jogos RTS, maximizar a produção militar faz com que os demais recursos avançados do jogo sejam produzidos também. Isso, porque inserir recursos militares mais fortes em um plano requer a produção de recursos avançados que são pré-condições para esses recursos militares. Com esse princípio é possível obter planos de ações que exploram a produção completa de recursos do jogo. Por exemplo, para produzir um recurso *Ghost* é necessário ter antes além de *Barracks*, os recursos *Academy* e *Covert Ops* que são recursos avançados do jogo. Esses permitem que novas atualizações sejam feitas e que novos recursos possam ser construídos.

Para que o SA consiga convergir e encontrar boas metas dentro do *StarCraft*, é necessário que duas ferramentas o auxiliem nessa tarefa. A primeira é um planejador responsável por construir a solução inicial. A segunda é o verificador de consistência responsável por gerenciar e validar todas as mudanças feitas pelo algoritmo na geração de novas soluções. Tais técnicas são de vital importância nessa abordagem e serão apresentas a seguir. Elas foram desenvolvidas em duas arquiteturas diferentes: Uma sequencial para uma primeira avaliação dos resultados e validação da pesquisa e outra com escalonamento. Primeiramente será descrita a abordagem sequencial e posteriormente aquela com escalonamento.

Análise de Complexidade

Nesta seção será feita uma análise de complexidade do algoritmo SA. Essa análise é baseada na premissa de que o SA pode ser visto como uma versão do *Simulated Annealing* clássico (apresentado na seção 2.1.2) acrescido das técnicas e algoritmos desenvolvidos neste trabalho. Tais acréscimos fazem do SA uma nova versão do *Simulated Annealing*. Assim, partimos do pressuposto que é possível analisar os algoritmos que compõe o SA e fazer menções dos impactos que causam as demais técnicas que foram utilizadas no algoritmo em relação a sua complexidade. O objetivo desta análise é demonstrar de modo inicial e não totalmente aprofundado a complexidade do algoritmo desenvolvido aqui, uma vez que uma análise completa e detalhada de todo o algoritmo e suas implicações é plausível de se tornar um trabalho a parte dessa dissertação. Contudo, uma análise mais

intrínseca e completa do *Simulated Annealing* pode ser encontrada em [Izquierdo 2000].

A complexidade aqui é vista como uma forma de tentar avaliar o consumo de determinada unidade pelo SA, quando esse busca resolver alguma instância do problema de escolha de metas em jogos RTS. Neste caso, a unidade é o tempo computacional. A forma mais comum de expressar esse consumo de tempo é em termos do tamanho da instância do problema tratado.

Para avaliar uma instância do problema é necessário estabelecer o tamanho da entrada do SA, uma vez que essa entrada é composta por um conjunto de ações que representa o plano inicial. A quantidade de ações que estão contidas em um plano inicial que é passado para uma instância do SA é definido de acordo com o tempo limite imposto para a meta desejada. Iremos considerar então, que nesta análise, a entrada (plano inicial) contém a quantidade média de ações que são escolhidas para compor uma meta de 2 minutos. A escolha desse tempo limite e logo da quantidade de ações é devido à eficiência do SA com metas baseadas neste intervalo de tempo limite, sendo esse enfatizado no decorrer do trabalho. O tamanho da entrada para o SA baseado nas ações do plano inicial durante a análise será denotado por n.

Com as considerações feitas anteriormente, fica claro que outros parâmetros do SA como resfriamento, tamanho da vizinhança e do espaço de soluções não são levados em conta. A análise dos algoritmos que compõe o SA será feita levando em conta apenas à arquitetura com escalonamento (capítulo 6), uma vez que essa arquitetura é uma das principais contribuições deste trabalho e é utilizada com frequência nos experimentos. Esses algoritmos são:

- Escalonador (algoritmo 14): responsável por escalonar as ações durante o planejamento com o POPlan e com o verificador de consistência SHELRChecker. Possui complexidade quadrática, isto é $\theta(n^2)$.
- POPlan (algoritmo 12): responsável por gerar o plano de ações inicial ($G_{inicial}$) que é passado ao SA. Possui complexidade exponencial, isto é $\theta(a^n)$.
- SHELRChecker (algoritmo 15): responsável por verificar a consistência dos planos de ação que são gerados pelo SA com o mecanismo de geração de vizinhos (linha 9, algoritmo 8). Possui complexidade exponencial, isto é $\theta(a^n)$.

Com as análises dos principais algoritmos que compõe o SA, iremos considerar que a avaliação de complexidade total do algoritmo é resultado da complexidade dos algoritmos que o compõe, considerando o pior caso. Assim, temos que o POPlan é executado apenas uma vez em uma instância do SA gerando o plano inicial, com o SHELRChecker sendo executado diversas vezes dentro do segundo *loop* (linha 8) do SA. A quantidade de vezes que esse último é executado é altamente dependente da configuração dos parâmetros do algoritmo, tais como N (linha 5), M (linha 8), P (linha 19) e do valor de α (linha 20).

Assim, essas quantidades serão descartadas para levarmos em conta apenas os maiores valores de tempo. Contudo serão feitas algumas menções em relação a esses parâmetros. Com os valores dos algoritmos individuais, a complexidade do SA pode ser dada pela equação 4.3:

$$a^n + n^2(a^n) \tag{4.3}$$

Com a equação 4.3 como referência, a complexidade de pior caso do SA pode ser dada por a^n, ou seja, complexidade exponencial, de acordo com o número de ações (n) do plano de ações inicial.

A construção do plano inicial pelo POPlan utiliza de funções aleatórias e o seu valor de complexidade vai sempre depender dessas funções. A análise do POPlan leva em conta apenas as complexidades individuais dos procedimentos que o compõe no algoritmo 12. O SA na linha 5 (algoritmo 8) permanece no *loop* até que a temperatura atinja o valor 0 ou até que um limite máximo de iterações seja alcançado. A velocidade em que temperatura inicial é definida (linha 3) e a velocidade em que ela decai em relação a α (linha 20), dependem da quantidade de ações n no plano inicial. Além disso, reduzir a temperatura do algoritmo depende também da quantidade de planos aceitos (linha 19). O uso da variável P (linha 19) é para garantir que o algoritmo quando estiver com um determinado plano como solução possa explorar ao máximo os demais planos vizinhos a este. O valor de M (linha 8) foi definido de forma empírica neste trabalho tendo o valor fixo de 30. M define quantas vezes o segundo *loop* do SA será executado. No entanto, poderia ser feita uma função para determinar o valor de M com base no valor de n, assim a complexidade do algoritmo seria alterada com o aumento ou não da chamada dos seus procedimentos algoritmos. Essas considerações mostram como as técnicas e procedimentos que fazem com que o SA possa convergir têm influência significante sobre a complexidade do algoritmo.

Apesar do resultado da análise de complexidade do SA, os resultados obtidos pelo algoritmo nos experimentos foram satisfatórios e atenderam as exigências de tempo de resposta e reatividade de jogos RTS. Tais resultados foram possíveis levando em conta o crescimento que foi identificado em relação à complexidade do algoritmo, à medida que o número de ações n aumentava. Essa percepção foi vital no desenvolvimento de estratégias que auxiliam o algoritmo a manter-se dentro das restrições de tempo real. Essas estratégias serão apresentadas no decorrer da dissertação.

Capítulo 5

Arquitetura Sequencial para Escolha de Metas

Neste capítulo será descrita a arquitetura sequencial da abordagem de escolha de metas em jogos RTS. Aqui serão descritas as duas ferramentas que compõem o SA, o planejador e também o verificador de consistência ambos da arquitetura sequencial. Essas duas técnicas controlam toda a geração e validação dos planos considerados durante o processo de busca do SA. Ao fim desse processo o plano de ações final, que é escolhido pelo SA através de suas ações, produz os recursos que determinam a meta que foi escolhida.

O decorrer deste capítulo está organizado da seguinte forma: Na Seção 5.1 é descrito o planejador sequencial e o seu funcionamento; o princípio de funcionamento e operações do verificador de consistência sequencial estão na Seção 5.2.

5.1 Planejador Sequencial

O Planejador sequencial foi desenvolvido baseado na técnica MEA [Chan et al. 2007]. Ele foi alterado devido às características da abordagem proposta sendo diferente do planejador de [Chan et al. 2007] e de outros planejadores sequenciais. O planejador aqui proposto não opera com uma meta associada a uma quantidade fixa de recursos para a construção do plano inicial. Ao invés disso, o plano de ações inicial deve atingir uma quantidade aleatória de recursos. Como parâmetro, o planejador é limitado a um *makespan* máximo para a execução das ações do plano. Esse limitador é o tempo (*makespan*) limite que influencia na quantidade de recursos que a meta terá durante o desenvolvimento do plano inicial e também durante a busca do SA.

Não utilizar uma meta com quantidade fixa de recursos no desenvolvimento do plano inicial é uma estratégia para inicializar o SA com planos de ações variados. Isso auxilia o algoritmo a não obter um comportamento tendencioso a certos tipos de planos e seus respectivos valores de função objetivo. O planejador recebe então um recurso escolhido de

forma aleatória como meta. Ele desenvolve o plano colocando as ações necessárias para construir esse recurso. Se durante esse processo e posteriormente no seu fim, o tempo limite imposto para o plano não tiver sido excedido, um novo recurso será aleatoriamente escolhido como nova meta. O algoritmo executa esse processo repetidamente até que seja construído um plano de ações com o limite de tempo estabelecido. Por exemplo, se for definido um tempo limite de 3 minutos (180 seg.), o planejador irá construir um plano onde as ações geram uma quantidade de recursos em 3 minutos de execução dentro do jogo. O Algoritmo 9 descreve o pseudocódigo do planejador.

Algoritmo 9 Planejador Sequencial(R_{meta}, R_{disp}, T_{limite})

```
 1: Plan ← Extrair(R_meta)
 2: for each Ação Act ∈ Plan do
 3:    Plan.push(R_predecessor)
 4:    if R_predecessor ← Predecessores(Act, R_disp) then
 5:       R_predecessor.existe ← true
 6:    end if
 7: end for
 8: for each Ação Act ∈ Plan do
 9:    if Act.bTime + TempoAtual <= T_limite then
10:       checaLegalidade(act, R_disp)
11:       Act.iniTime ← actualTime
12:       Act.endTime ← actualTime + Act.bTime
13:       if Act.existe = false then
14:          Atualiza(R_disp, Act)
15:       end if
16:    else
17:       Sair()
18:    end if
19: end for
20: return Plan
```

No início do algoritmo a função $Extrair(R_{meta})$ (linha 1) seleciona o recurso que é a meta a ser alcançada no momento e insere sua respectiva ação no plano de ações $Plan$. Uma vez feito isso, o algoritmo então inicia uma busca pelas ações que estão no plano para inserir também as pré-condições (recursos predecessores) de cada uma delas no plano (linhas 2 a 9). A função $Predecessores(Act, R_{disp}))$ (linha 3) percorre o grafo de pré-condições da ação, verificando se cada recurso presente nesse grafo está contido na lista de recursos disponíveis do jogo (R_{disp}). Caso algum recurso predecessor já estiver disponível, então sua respectiva ação é inserida na variável $R_{predecessor}$ e tem o parâmetro $existe$ marcado como verdadeiro ($true$). Isso significa que a ação vai entrar no plano, mas não precisa ter suas pré-condições verificadas, pois o respectivo recurso já foi construído no jogo e está disponível. Caso a pré-condição não exista, a ação respectiva para a produção desse recurso ($R_{predecessor}$) é atribuída ao plano sem marcação.

Através do processo de verificação de pré-condições descrito anteriormente, quando

é adicionada uma ação que é pré-condição de outra, as pré-condições dessa ação que foi adicionada são verificadas imediatamente. Isso garante que sempre que uma ação for executada todos os seus predecessores já foram executados também, pois eles são colocados no plano sempre antes da ação que necessita deles, evitando retrabalho de verificação. A Figura 5.1 demonstra o processo de busca pelos predecessores de uma ação a partir de seu respectivo recurso. Nessa busca, o estado inicial da busca é o estado nativo de recursos quando o *StarCratf* é iniciado, com 4 *Scv* e 1 *CommandCenter*. Quando o *loop* de verificação chega ao fim (linha 9), o plano já possui todas as ações necessárias para gerar uma quantidade de recursos aleatória dentro do tempo limite estabelecido. O algoritmo neste momento inicia a execução de sua segunda etapa.

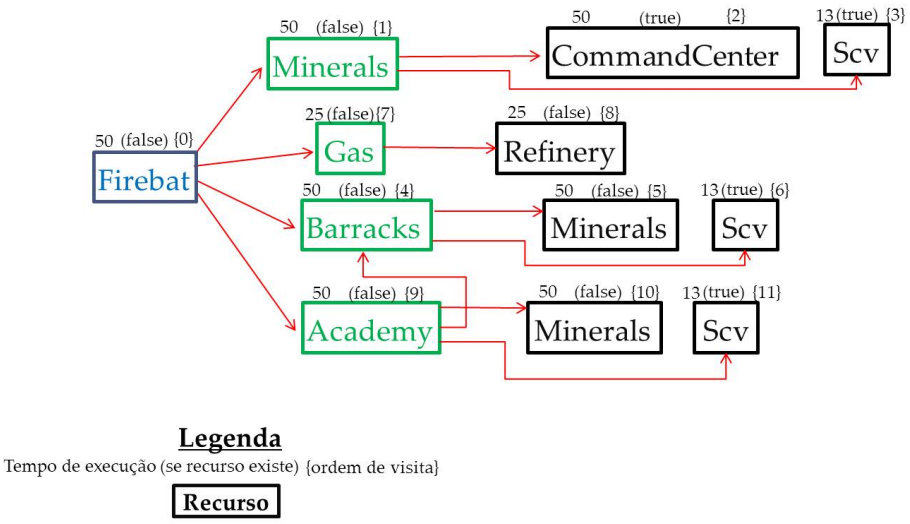

Figura 5.1: Ordem em que as pré-condições da ação que produz o recurso *Firebat* são visitadas.

Com todas as ações já inseridas no plano, é preciso configurar os atributos e validar as mesmas. Na segunda parte do algoritmo essas tarefas são feitas. Novamente é iniciado um *loop* que percorre as ações do plano (linha 10). Para cada ação do plano, inicialmente é verificado (linha 11) se o tempo que de execução dela, ou seja, o tempo que ela leva para ser executada dentro do jogo mais o tempo atual em que o jogo se encontra ($TempoAtual$), não irão exceder o tempo limite do plano ($Tlimite$). Caso a ação exceda, o método *Sair* (linha 23) é invocado. Esse método finaliza o planejamento com a ação que teve seus atributos configurados por último e retorna o plano como sendo o plano final.

Quando o tempo de execução de uma ação não excede o tempo limite, o algoritmo então verifica se a ação corresponde a um recurso que deve ser criado ou se é relativa a um recurso que já existe (linha 12). O método $checaLegalidade(act, R_{disp})$ (linha 13, 18) verifica os atributos dos recursos predecessores da ação, verificando se estão disponíveis validando sua execução. Após isso, os atributos de tempo da ação são configurados, esses são relativos ao momento em que a ação começa a ser executada e o momento em que sua

execução vai ser finalizada respectivamente. O intervalo entre o tempo de início e fim de execução de uma ação é o tempo que ela leva para ser executada e também o tempo em que ela consome recursos dentro do jogo.

Uma ação consiste de uma *tupla* (b, i, e, c, $Predecessores$), onde b é o tempo de execução completo da ação, i é o tempo de início de sua execução e e o tempo final, c é o custo da ação em termos de recursos que esta consome e $Predecessores$ é uma lista com os recursos que são pré-condições da ação. No algoritmo o método $Atualiza(act, R_{disp})$ (linha 16) é invocado quando uma ação tem seus atributos configurados e está sendo construída. Ele é responsável por inserir o recurso criado por essa ação entre os recursos disponíveis do jogo, atualizando a lista R_{disp}. Os demais recursos que podem ser gerados como *Minerals* ou *Gas* também são atualizados. Esse método não é invocado quando a ação trata-se de um recurso que já estava disponível no jogo, pois essa possui todos os seus atributos configurados e os recursos gerados por ela já foram adicionados ao planejamento.

Por fim, o plano de ações que atinge o recurso aleatório passado como meta é retornado (linha 26). Caso o tempo limite ainda não tenha sido atingido, um novo recurso será passado como meta para o planejador gerar um novo plano que ao fim será concatenado com esse último. O planejador repete esse processo até que o tempo limite tenha sido atingido. O controle desse processo de criação de planos e concatenação é feito por um *loop* externo, que permite que um recurso seja passado como meta a cada execução. Esse sistema ajuda a verificar quando alguma ação irá exceder o tempo limite do plano. Quando isso acontece o planejamento é interrompido e o plano construído até o momento é válido e retornado para ser acrescentado com os demais planos já desenvolvidos formando o plano final. O Algoritmo 10 mostra a implementação do método que gerencia o planejador sequencial.

Algoritmo 10 Gerar Plano($E_{inicial}$, T_{limite})

1: $R_{disp} \leftarrow E_{inicial}$
2: $limite \leftarrow false$
3: **while** $limite = false$ **do**
4: $\quad R_{meta} \leftarrow RandomRecurso()$
5: $\quad PlanoFinal \leftarrow$ Planejador Sequencial($R_{meta}, R_{disp}, T_{limite}$)
6: $\quad limite \leftarrow ConfTempo(PlanoFinal)$
7: **end while**
8: **return** $PlanoFinal$

No Algoritmo 10, o estado inicial de recursos do jogo $E_{inicial}$ e tempo limite T_{limite} são os parâmetros. O algoritmo basicamente configura as variáveis necessárias para chamar o planejador sequencial. A função $RandomRecurso()$ (linha 4) escolhe um recurso aleatório para ser usado como meta. O controle do planejador e concatenação de planos (linhas 3 a 7) é feito até que o tempo limite seja alcançado, por fim o plano final é retornado.

5.2 Verificador de Consistência Sequencial

Em planejamento, quando são feitas mudanças em um plano de ações é necessário verificar como essas mudanças irão afetar o plano. Essa necessidade torna-se ainda maior quando a relação de dependência e restrições entre as ações é forte, como no caso do *StarCraft*. A simples inserção ou remoção de uma ação pode significar a inviabilização de diversas ações que lá já estão. Entretanto, tal modificação pode fazer com que o plano tenha condições para aceitar novas ações ou tornar ações que lá já estão e são inviáveis em viáveis novamente. Essa relação é dada pela perda e ganho de recursos e necessita ser verificada com eficácia para que não prejudique o planejamento. Deste modo, será apresentada a primeira versão do algoritmo de verificação de consistência, capaz de efetuar as mudanças em um plano de ações, verificar sua consistência, além de ordenar e gerenciar seus recursos.

O verificador de consistência é necessário para também manter as soluções geradas dentro das características desejadas. Se o planejador desenvolve um plano linear, com as ações em determinada ordem e mantendo todas as restrições entre as ações, é necessário que as soluções geradas sigam o mesmo padrão. Porém, quando uma nova solução é gerada a tendência é que seu plano de ações perca parte de suas características devido às mudanças que afetam as ações. Quando uma ação é afetada, todas as demais que tem alguma ligação com ela seja de pré-condição ou de ordem de execução dentro do plano são afetadas também. Assim, o verificador de consistência é usado quando o SA vai gerar uma nova solução mantendo essa dentro das características do domínio de recursos do *StrarCraft*. Ele executa a operação no plano, verifica as mudanças feitas validando às mesmas e atualiza todas as variáveis de controle do planejamento. O Algoritmo 11 mostra o pseudocódigo do mesmo.

Existem diversas maneiras de gerar um novo plano vizinho no SA. O mecanismo utilizado nesta abordagem é o seguinte: É escolhido aleatoriamente um dos dois movimentos, escolher uma ação aleatoriamente dentro do plano e substituir ela por uma nova ação escolhida aleatoriamente entre todas do domínio ou escolher duas ações de modo aleatório dentro do plano e trocar as duas de lugar, uma assumindo a posição da outra. Dessa forma, o algoritmo recebe em seus parâmetros iniciais o plano de ações $Plano$, uma lista $NovasAct$ que contém a operação escolhida juntamente com a nova ação que irá entrar no plano ou as duas ações que irão trocar de lugar, além dos recursos disponíveis no jogo no atual momento R_{disp}.

No início do seu funcionamento, o Algoritmo 11 verifica se uma ou várias ações irão ficar inviáveis devido a operação que vai ser executada no plano. Seja essa de substituir uma ação ou de efetuar uma troca de posições. Essa verificação é feita pela função $ChecaPlan(NovasAct)$ (linha 1), e as ações que irão ficar inviáveis são adicionadas na lista de ações inviáveis $actionsInv$. Na função, quando uma ação vai ser substituída

Algoritmo 11 Verificador Sequencial($Plano$, $NovasAct$, R_{disp})

```
1: actionsInv ← ChecaPlan(NovasAct)
2: for all Ação ActInv ∈ actionsInv do
3:    for all Ação Act ∈ Plano do
4:       if Act.Predecessores = ActInv then
5:          Act.viavel ← false
6:          actionsInv.push(Act)
7:          penalti += μ
8:          Atualiza(Act, R_disp)
9:       end if
10:   end for
11: end for
12: AdaptaPlano(NovasAct, R_disp)
13: continua ← true
14: Ação Act
15: while continua = true do
16:   if Act ← ChecaPreds(actionsInv, R_disp) then
17:      Act.viavel ← true
18:      actionsInv.erase(Act)
19:      penalty -= μ
20:      Atualiza(Act, R_disp)
21:   else
22:      continua ← false
23:   end if
24: end while
25: return Plano
```

a nova ação que entra no plano é sempre marcada como inviável, pois não se sabe se os recursos existentes e produzidos até o momento de sua entrada são suficientes para satisfazer suas pré-condições. No caso de troca, é preciso averiguar se as ações possuem suas pré-condições atendidas com os novos tempos iniciais e finais de execução que elas assumem.

Quando ao menos uma ação é colocada na lista de ações inviáveis, o algoritmo consegue encontrar as demais ações que irão torna-se inviáveis a partir dessa. Isso é feito percorrendo o grafo de recursos predecessores de cada ação do plano (linhas 2 a 11). Se alguma ação tiver um recurso em sua lista de pré-condições e essa estiver com a ação que a produz dentro da lista de ações inviáveis, significa que essa ação vai ficar inviável devido a uma pré-condição que não está mais sendo satisfeita. Como as ações inviáveis são colocadas sempre no final da lista *actionsInv*, nas próximas iterações as ações que não são mais viáveis e entraram na lista na iteração atual serão verificadas e assim por diante. Esse encadeamento de ações pode ser descrito com o seguinte exemplo de sentença: $\neg$*collect-minerals* $\rightarrow$ $\neg$*build-barracks* $\rightarrow$ $\neg$*build-academy* $\rightarrow$ $\neg$*build-firebat*, onde $\neg$ significa que uma ação está sendo negada e $\rightarrow$ representa a implicação de tal negação. Assim, podemos dizer que inviabilizar a ação *collect-minerals* significa inviabilizar a ação *build-barracks* que tem

a anterior como pré-condição, que implica em negar *build-academy* que tem *build-barracks* como pré-condição. Essa cadeia tem fim na negação de *build-firebat*, que é o último recurso a tornar-se inviável devido à primeira ação inviável que era *collect-minerals*.

Sempre que uma nova ação entra na lista de ações inviáveis, a variável *penalti* é incrementada (linha 7). No momento de avaliar o plano que será retornado pelo verificador de consistência o valor da variável será utilizado. A função $Atualiza(Act, R_{disp})$ (linha 8) é chamada para determinar quais recursos não mais existirão e quais irão ficar disponíveis para que outras ações possam usar. Isto é feito baseado na ação que vai se tornar inviável e que não irá mais produzir os recursos que antes produzia, e nas ações que estavam produzindo recursos para que essa ação os consumisse. Uma vez que a ação não consome mais os recursos, esses ficarão disponíveis dentro da lista R_{disp}.

Por exemplo, se uma ação que produz um recurso *ScienceFacility* torna-se inviável, o mesmo acontecerá com *CovertOps* e *NuclearSilo*, pois ambas ficaram sem um de seus recursos predecessores. No entanto, o plano de ações terá 88 segundos de tempo extra e será disponibilizado 250 *Minerals* e 300 *Gas*, provenientes da inviabilização dos recursos anteriores. Tais ocorrências habilitam a chegada de novas ações ou fazem com que ações que estavam inviáveis no plano possam voltar a ser viáveis com os recursos que ficaram disponíveis. Caso o montante de recursos que será liberado não cobrir a perda das ações que ficaram inviáveis, devido as suas contribuições ao plano, a variável *penalti* é considerada para auxiliar o algoritmo em sua decisão.

Com a definição de quais ações estão inviáveis e a quantidade de recursos disponíveis, o algoritmo chama o método $AdaptaPlano(NovasAct, R_{disp})$ (linha 12). O método efetua mudança no plano executando a operação que foi selecionada. Além disso, o método também rearranja o plano de ações em relação à ordenação das ações. Uma vez que o plano é sequencial a mudança nos tempos de execução de uma ação significa mudanças nesses mesmos tempos entre todas as ações que estão à frente dela na sequência de execução.

Em seu final, o algoritmo checa se existem ações que podem voltar a ser viáveis devidos às mudanças que já foram gerenciadas anteriormente. O algoritmo percorre a lista de ações inviáveis verificando os recursos predecessores de cada uma. A função $ChecaPreds(actionsInv, R_{disp})$ (linha 16) é usada nessa tarefa, onde essa recebe a lista de ações inviáveis que será percorrida e também a lista contendo os recursos disponíveis. Se alguma ação tem condição de tornar-se viável novamente a função a retornará. Tal ação é removida da lista de inviáveis, o valor da variável *penalti* é decrementado e a função de atualização de recursos $Atualiza(Act, R_{disp})$ (linha 20) remove aqueles que foram consumidos pela ação que tornou-se viável novamente.

A complexidade da função $ChecaPreds(actionsInv, R_{disp})$ depende da quantidade de ações que ficam viáveis novamente (junto com o grafo de pré-condições de cada uma). Neste caso, o encadeamento de ações se aplica, pois uma ação que torna-se viável pode fazer com que outras fiquem viáveis também. Caso nenhuma ação seja escolhida entre

as inviáveis o algoritmo interrompe a repetição (linha 22) e finaliza sua execução, já que nenhuma ação pode voltar a ser viável. Um exemplo seria uma ação de *Minerals* que volta ser viável, os recursos produzidos por essa podem fazer com que outra ação seja possível de ser executada também, sendo necessário uma nova verificação sobre a lista de ações inviáveis.

Finalmente, o algoritmo retorna a nova solução que será avaliada pelo SA, onde essa pode ou não ser escolhida como solução atual. Uma característica presente nesta arquitetura sequencial é que na maior parte dos experimentos, as soluções finais encontradas pelo SA além de possuir força de ataque superior ao plano inicial, também possuem *makespan* menor em relação a esse. Entretanto, diminuir o *makespan* das soluções finais não é um dos objetivos propostos neste trabalho. A seguir é apresentado um exemplo que descreve o funcionamento do verificador de consistência sequencial.

Exemplo de Aplicação do Verificador Sequencial

Para esse exemplo, a execução do verificador de consistência será baseada em um plano de ações inicial que alcança (solução) o recurso *Firebat*. Tal plano é ilustrado na Figura 5.2. O estado desse plano pode ser representado da seguinte forma: (13, 0, 42, 6, 6, 16, 520). Esse valores correspondem respectivamente ao número de ações, número de ações inviáveis, quantidade de *Minerals* disponível, quantidade de *Gas* disponível, quantidade de *Supply* utilizado, força de ataque e *makespan* do plano de ações. O tempo limite do plano é de 530 segundos, tal valor não pode ser excedido.

Primeiramente, suponha que em sua primeira execução o SA determine a inserção da ação *build-marine* no lugar da ação de número 10 do plano e atual solução do algoritmo. Deste modo, a ação *collect-minerals* é colocada na lista de ações inviáveis, essa é uma pré-condição de *build-academy*. Assim, a ação *build-academy* também é colocada na lista junto com *build-firebat* a qual necessita dessa última. Nesse instante, o estado do plano passa a ser (13, 3, 186, 31, 4, 0, 410), com duas ações inviáveis a mais, com mais recursos do tipo gás e minerais, menos recursos do tipo *supply* usados, menos força de ataque e *makespan* menor. O algoritmo então executa a operação e insere a nova ação atualizando os recursos consumidos e gerados. A ação é viável baseado nos recursos disponíveis e o estado do plano é alterado para (13, 2, 136, 31, 5, 12, 425). A Figura 5.3, ilustra essa primeira iteração do verificador. Nela, acima de cada ação estão os tempos de início e fim de execução e as marcações de ação viável (V) e ação inviável (NV), a figura ilustra as duas etapas do algoritmo. A primeira onde as ações que tornam-se inviáveis são definidas, e a segunda onde a ação é inserida e definido quem pode voltar a ser viável.

O algoritmo não consegue viabilizar nenhuma das duas ações que tornaram-se inviáveis nessa iteração, uma vez que nova ação que entrou no plano não contribui para nenhuma delas. Embora existam *Gas* e *Minerals* suficientes para a ação *build-firebat*, essa não pode ser viabilizada devido a ausência da ação *build-academy* que satisfaz uma de suas

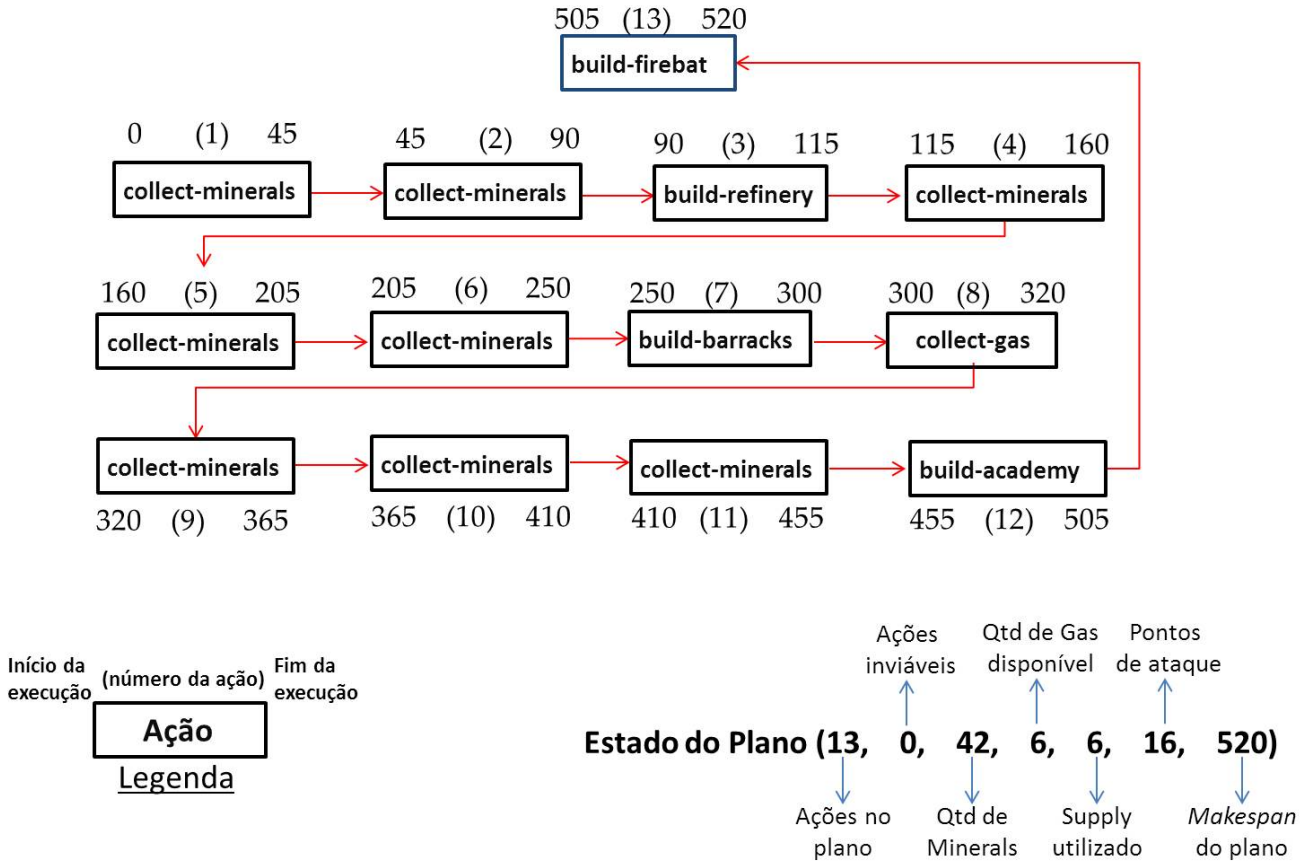

Figura 5.2: Um plano de ações para construir o recurso *Firebat*.

pré-condições. O plano é então retornado para o SA, que irá optar por ele, mesmo tendo quantidade de pontos de ataque inferior a solução atual.

Com a nova solução aceita, suponha que o SA em sua segunda execução opte por substituir a ação *collect-gas* de número 8, por uma outra nova ação *build-marine*. A ação que vai ser substituída produz um dos recursos predecessores da ação *build-firebat*, no entanto essa última ação já está inviável no plano. Assim, a remoção da *collect-gas* não afeta outras ações do plano, apenas diminui a quantidade desse recurso entre os disponíveis e diminui o *makespan* do plano. Neste momento, o estado do plano é (13, 3, 136, 0, 5, 12, 405). Nesse, a ação *build-marine* que acabou de entrar possui suas pré-condições presentes no plano. No entanto, a ação *collect-minerals* (número 11) que é uma de suas pré-condições está inserida em uma posição posterior, ou seja, *build-marine* está sendo executada antes de sua pré-condição. Assim, ela continua no plano como inviável e também será inserida na lista de ações inviáveis. Nenhuma ação torna-se viável novamente na verificação final do algoritmo, e o estado final dessa solução continua o mesmo de antes. A Figura 5.4 ilustra essa segunda iteração do algoritmo.

Agora, com o SA em sua terceira execução suponha que ele opte por uma troca de posições entre ações do plano. A troca é entre a ação *collect-minerals* (número 11) e a

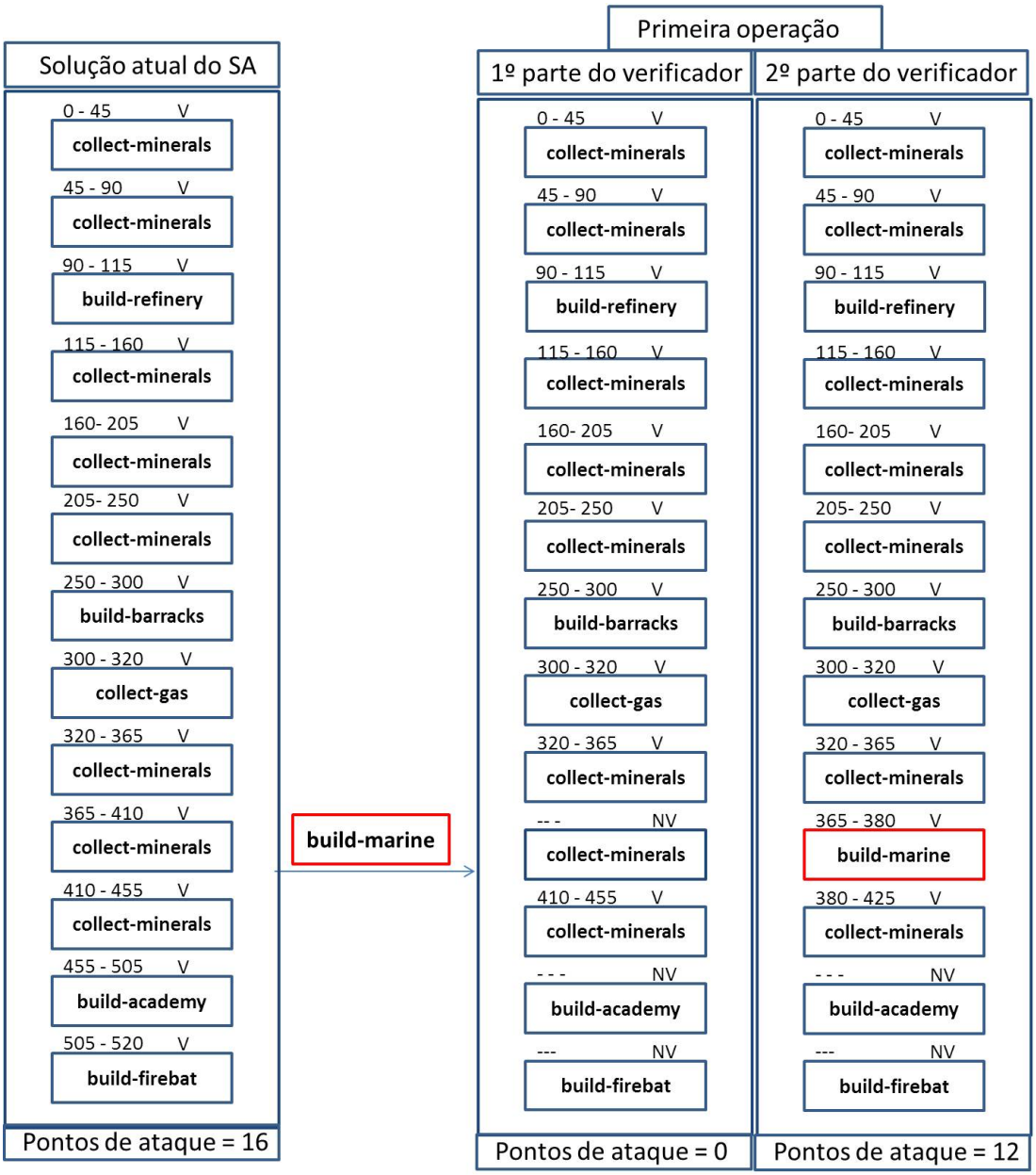

Figura 5.3: Primeira iteração do verificador de consistência sequencial.

ação *build-marine* (número 8). Nessa operação, ambas as ações mantêm seus status de viável/inviável em suas novas posições, não sendo necessário verificar se alguma outra ação do plano irá ficar inviável. O estado plano é (13, 4, 86, 0, 5, 12, 380). Após efetuar a troca, o algoritmo detecta que ação *build-marine* que agora está em uma nova posição e com novos tempos de execução tem todas as suas pré-condições satisfeitas, já que seus predecessores estão todos sendo executados antes dele. Essa ação volta a ser viável então. O estado plano agora é (13, 3, 86, 0, 6, 24, 420). A Figura 5.5 ilustra a terceira iteração do verificador. Mesmo com duas ações inviáveis o plano possui seu valor de ataque em 24. A nova solução gerada nesta iteração é melhor do que as solução inicial e as demais que foram geradas. O *makespan* dessa também é menor em relação ao das outras.

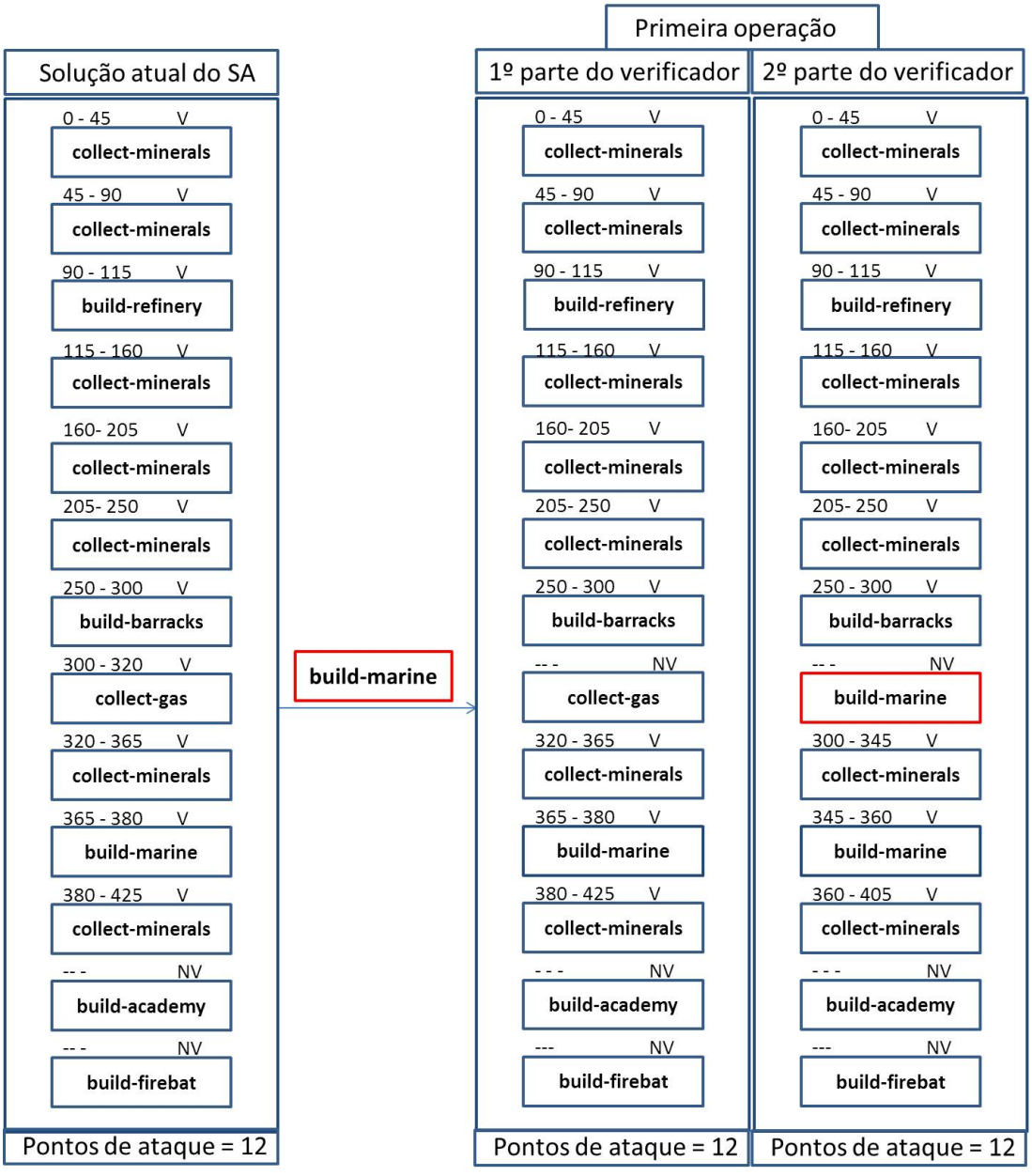

Figura 5.4: Segunda iteração do verificador de consistência sequencial.

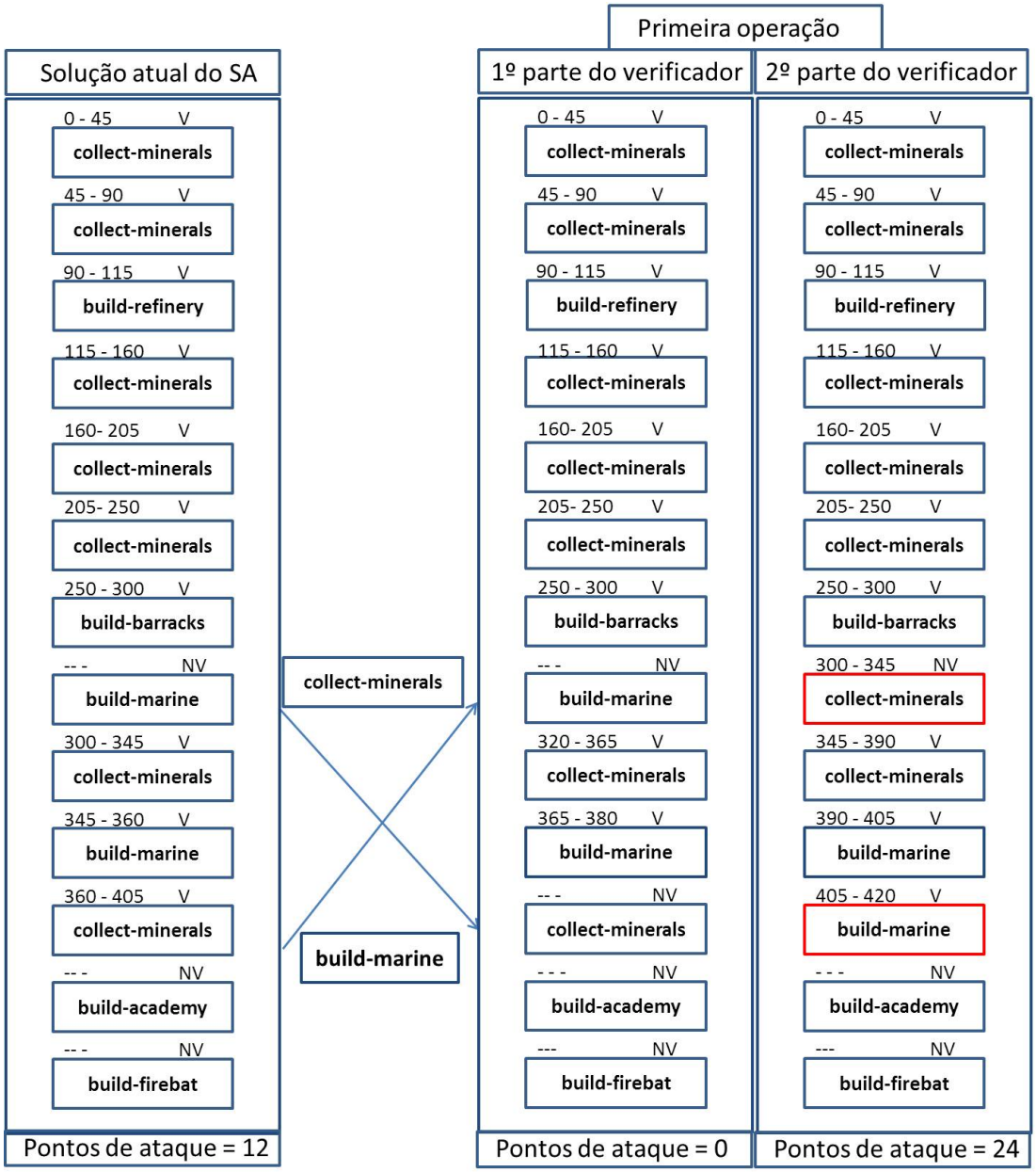

Figura 5.5: Terceira iteração do verificador de consistência sequencial.

Capítulo 6

Arquitetura com Escalonamento para Escolha de Metas

Neste capítulo será apresentada a arquitetura com escalonamento de ações desenvolvida para a abordagem de escolha de metas, através da maximização da produção de recursos em jogos RTS. Com base nos resultados obtidos na arquitetura do Capítulo 5, foi possível prosseguir com a pesquisa e desenvolver essa segunda arquitetura, que foca em metas com paralelismo de ações. Tal arquitetura, permite obter resultados muito próximos dos que são obtidos por jogadores experientes de jogos RTS, em específico de *StarCraft*. Jogadores experientes tendem a paralelizar ao máximo as ações executadas durante uma partida, uma vez que é possível executar mais ações em um intervalo de tempo, gerando assim mais recursos.

Para que o SA consiga encontrar uma meta que maximize a força de ataque do exército gerado e essa também esteja com suas ações escalonadas, é preciso que as duas técnicas principais que o compõe estejam adaptadas para tal finalidade. Com isso, foi desenvolvido um sistema utilizando planejamento de ordem parcial que é capaz de construir um plano de ações inicial e ao mesmo tempo escalonar suas ações, chamado de POPlan. Também foi proposto um algoritmo de escalonamento integrado para ser usado no planejador e no verificador de consistência com escalonamemto. O verificador de consistência que utiliza escalonamento de ações denominado SHELRChecker, é responsável por gerar e validar novas soluções que também mantém o paralelismo das ações.

Nos trabalhos de [Chan et al. 2007] e [Branquinho et al. 2011b] a arquitetura de escalonamento consiste em um processo independente dentro de suas abordagens, onde os algoritmos propostos são utilizados de forma separada dos outros processos. Neste trabalho aqui apresentado, o processo de escalonamento deve ser feito de forma acoplada acompanhando tanto o planejador quanto o verificador, uma vez que várias soluções são geradas durante a busca. Assim, a arquitetura com escalonamento torna a abordagem mais eficiente gerando resultados compatíveis com os de jogadores experientes, o que possibilita o uso desta pesquisa em outras áreas onde a abordagem possa trazer uma nova

gama de soluções.

O restante desse capítulo está organizado da seguinte forma: A Seção 6.1 apresenta o planejador POPlan junto com suas principais técnicas e algoritmos. Na Seção 6.2 é feita a apresentação do verificador de consistência com escalonamento SHELRChecker mostrando seu funcionamento junto ao SA.

6.1 POPlan - Planejador de Ordem Parcial com Escalonamento

O POPlan é um planejador que utiliza o conceito de planejamento de ordem parcial [Minton et al. 1994] para gerar um plano de ações e executar a tarefa de escalonamento durante esse processo. Seu desenvolvimento foi apoiado na necessidade de um planejador que pudesse gerar um plano de ações como entrada para o SA, habilitando a busca por soluções com ações paralelizadas. Outra necessidade é que o tempo gasto durante esse processo não fosse elevado. De fato, a arquitetura com escalonamento teve como um dos maiores desafios retornar resultados com qualidade dentro de um intervalo de tempo aceitável para uma abordagem de tempo real.

A principal motivação para a construção do POPlan, está no ganho significativo que a produção de recursos tem em relação ao planejamento sequencial, uma vez que a escolha de uma meta com qualidade se dá pela maximização dessa produção. A Figura 6.1 ilustra essa vantagem na produção de recursos. Na figura, o plano (A) ilustra as ações necessárias para construir o recurso *Firebat*, onde essas tiveram os tempos de execução atribuídos e são executadas seguindo a ordem linear indicada pelas setas. Esse plano leva 520 seg para executar as ações. Já no plano (B), o POPlan determina as ações necessárias para construir os recursos e as escalona. Neste plano as setas representam os *links causais* entre as ações. Esses indicam a ordem de restrição e de recursos que serão produzidos e utilizados entre as ações. Tal ordem especifíca apenas que uma ação deve ser executada antes de uma outra ação da qual ela é pré-condição, mas o quão antes é indiferente. Assim, a ordem de execução das ações não segue uma ordem fixa e linear como do planejador sequencial. Para construir o recurso *Firebat* o plano de ações encontrado pelo POPlan representa o melhor resultado em termos de *makespan*, apesar do algoritmo não garantir solução ótima sempre. A Figura 6.2 demonstra a ordem de execução das ações.

A arquitetura do POPlan é composta de dois níveis. No primeiro, os processos de satisfação das restrições do problema para composição do plano são feitos e no segundo as ações definidas por esse processo são submetidas ao escalonamento. Esses dois níveis intercalam suas execuções e são descritos pela Figura 6.3. Com essa arquitetura, o algoritmo possui o conceito de acoplamento forte [Fink 2003], onde os problemas de planejamento e escalonamento são reduzidos a uma representação uniforme. Caso alguma inconsistência

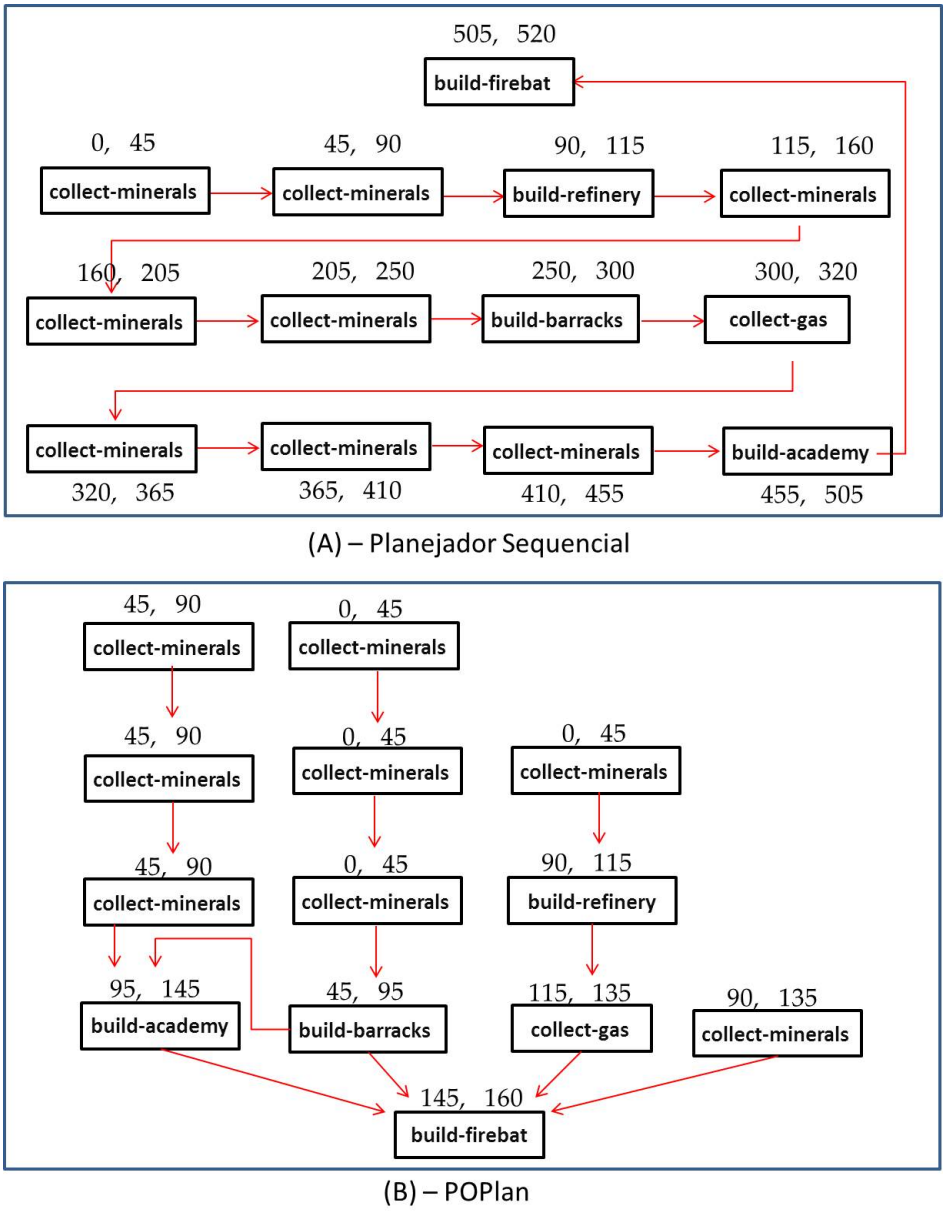

Figura 6.1: Comparação entre os planos obtidos usando POPlan e o planejador sequencial.

seja encontrada na etapa do escalonamento, é possível verificá-la e satisfazê-la na etapa do planejamento que está sendo executada intercalada com essa. Assim, não é preciso interromper todo o processo para tal verificação. Neste acoplamento, o POP é essencial para conseguir atingir essa representação, pois os *links causais* definidos entre as ações na etapa de planejamento possuem a ordem temporal das restrições entre as ações, que auxilia também na produção e uso dos recursos. Com isso, a etapa de escalonamento apenas encontra o melhor tempo para a ação ser executada, deixando as tarefas de planejamento onde as informações de recursos e restrições entre as ações são verificadas e definidas, para a outra etapa.

O POP permite que o acoplamento em um nível mais intrínseco seja atingido, pois se fosse utilizado, por exemplo, o planejador sequencial, essa representação seria decomposta em dois problemas sequenciais distintos. Esse tipo de representação é denominada

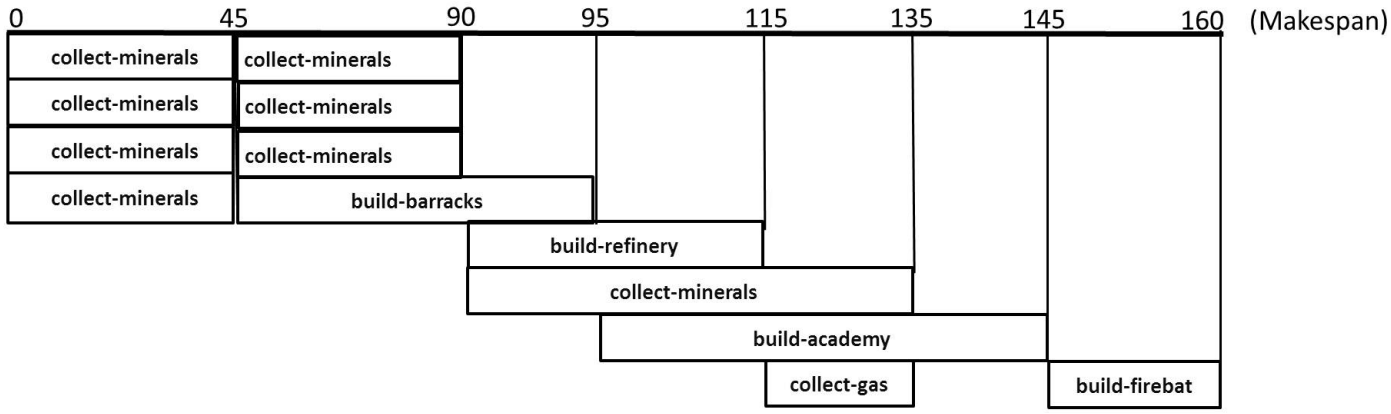

Figura 6.2: Ordem de execução do plano de ações gerado na Figura 6.1.

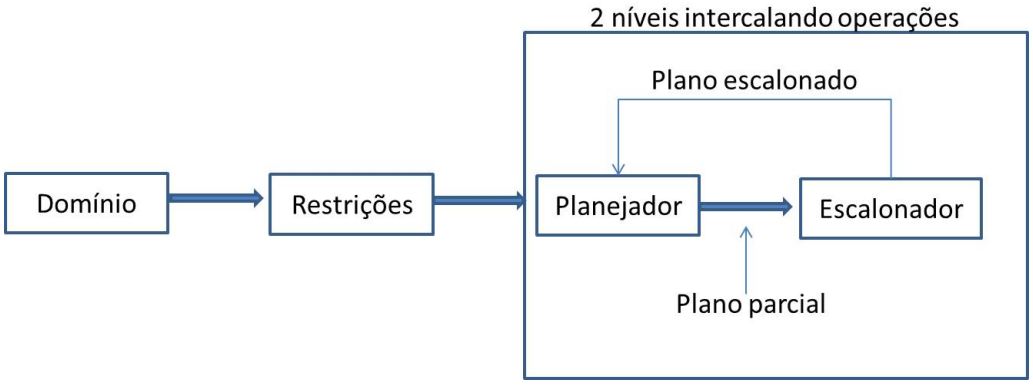

Figura 6.3: Representação do acoplamento forte intercalando POP e escalonamento.

acoplamento fraco [Fink 2003] e é descrita na Figura 6.4. Com o acoplamento fraco não é possível utilizar informações de uma etapa para melhorar o desempenho da outra. O planejador sequencial gera um plano atemporal com restrições de ordem linear e sem nenhuma informação relativa a ordem de produção de recursos. Assim, a etapa de planejamento é executada uma única vez antes do processo de escalonamento ser iniciado e determinar o plano de ações final. Com o POPlan, o processo de planejamento e escalonamento estão integrados de modo que essas duas etapas contribuem mutuamente para um processo mais eficiente.

Foram considerados alguns trabalhos que abordam a questão de integração entre planejamento e escalonamento. Entre esses, destaca-se [Fink 2003] e [Wilkins et al. 1995], onde esses buscam resolver problemas de planejamento mantendo as restrições temporais. No entanto, esses trabalhos não consideram restrições de recursos durante o planejamento, um requisito presente no problema aqui abordado. Nessas abordagens, não considerar a restrição de recursos é uma forma de diminuir o espaço de busca. Assim, o POPlan diferente das abordagens citadas, explora todas as restrições presentes no domínio de jogos RTS, mantendo o foco em obter tempo de resposta compatível com as restrições deste

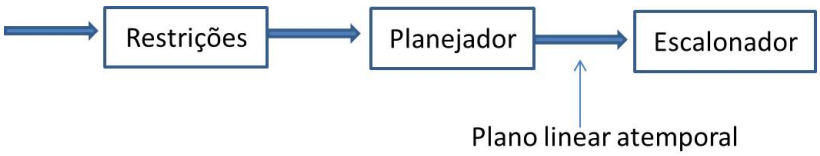

Figura 6.4: Representação do acoplamento fraco com a decomposição dos processos de planejamento sequencial e escalonamento.

ambiente. O Algoritmo 12 descreve o pseudocódigo do planejador.

Algoritmo 12 POPlan($Plan$, R_{meta}, T_{limite}))

```
1:  Plan ← ConstroiLink(Plan, R_meta)
2:  for each Ação Act ∈ Plan do
3:     vezes ← 1
4:     i ← 0
5:     if Act.unidade = false then
6:        vezes ← Quantidade(Act)
7:     end if
8:     while i < vezes do
9:        tmpInicio ← BuscaTMinimo(Act)
10:       melhorTempo ← MAX
11:       for each Action ActRs ∈ Plan do
12:          if ActRs = Act.rbase then
13:             melhorTempo ← Escalona(Plan, ActRs, Act, tmpInicio, melhorTempo)
14:          end if
15:       end for
16:       contr ← Constroi(Plan, Act, melhorTempo)
17:       if contr = false then
18:          Sair()
19:       end if
20:       i ← i + 1
21:    end while
22: end for
23: return Plan
```

O POPlan recebe como parâmetros a lista onde as ações que irão compor o plano serão inseridas $Plan$, o recurso que é a meta atual R_{meta} e o tempo limite para a execução das ações do plano T_limite. Assim como o Planejador Sequencial (Seção 5.1), o POPlan desenvolve um plano de ações que atinge uma quantidade aleatória de recursos limitada ao tempo (*makespan*) limite definido para o desenvolvimento desse. Com o POPlan, um plano de ações que atinge uma quantidade de recursos em um intervalo de, por exemplo, 3 minutos, consegue produzir muito mais recursos que o planejador sequencial. Em seu início de funcionamento, o POPlan chama o método $ConstroiLink(Plan, R_{meta})$ (linha 1), responsável por definir todas as ações que irão compor o plano e seus respectivos *links*

causais.

O POPlan foi construído baseado no conceito de planejamento de ordem parcial. Todas as ações que compõe o plano possuem *links causais*. É possível definir os *links* em dois tipos. O primeiro é o *link* de precedência, onde uma ação que possui tal *link* é necessária para a execução de outra ação que recebe o segundo tipo de *link*, que é o link de condição. Por exemplo, uma ação *collect-minerals* com um *link* para uma ação *build-factory* significa que a ação de coletar recursos minerais precede a execução da ação que irá construir uma *Factory*. Essa ação por sua vez possui um *link* que indica que a ação de coletar minerais é uma condição para sua execução. Assim, quando uma ação alterar algum de seus atributos, fica mais fácil descobrir quais outras ações podem sofrer alterações devido a essa. Esses dois tipos de *links* são definidos em cada ação como duas listas, onde a primeira é a *link* (*link* de precedência) e a segunda é a *linkReb* (*link* de condição) no Algoritmo 13.

O algoritmo $ConstroiLink(Plan, R_{meta})$ estabelece todas as ações do plano para alcançar o recurso meta e define todos os *links causais* necessários. Ele recebe como parâmetros o plano de ações e o recurso meta. O Algoritmo 13 mostra o pseudocódigo do método. A ordem de como as ações são colocadas no plano juntamente com seus *links* é uma das estratégias de eficiência para o POPlan conseguir melhores resultados.

Algoritmo 13 ConstroiLink($Plan$, R_{meta})

```
1:  Plan ← Extrair(R_meta)
2:  for each Ação Act ∈ Plan do
3:    if Act.finalizou = false then
4:      for each Recurso Rsc ∈ Act.Predecessores do
5:        Action Actn
6:        existe ← ChecaPredecessores(Act, Rsc, Actn)
7:        if existe = false then
8:          Act.linkReb.push(Acnt)
9:          Actn.link.push(Act)
10:         Plan.push(Acnt)
11:         Plan ← ConstroiLink(Plan, R_meta)
12:       else
13:         Act.linkReb.push(Acnt)
14:         Actn.link.push(Act)
15:       end if
16:     end for
17:     Act.finalizou = true
18:   end if
19: end for
20: return Plan
```

O algoritmo $ConstroiLink(Plan, R_{meta})$ insere a primeira ação no plano e a partir de seus recursos predecessores preenche este com as ações necessárias restantes. Quando uma ação é selecionada, é verificado se ela já não possui todas suas pré-condições atendidas no

plano através do atributo *existe* (linha 3). Essa verificação é necessária, pois quando uma ação é colocada no plano para satisfazer determinada pré-condição, imediatamente são verificados as pré-condições dessa ação que acabou de entrar (linha 11). Então é possível que uma ação do plano que irá ser verificada tenha suas pré-condições já atendidas. A ordem em que as ações são colocadas no plano estabelece um posicionamento entre essas e seus *links* que é capaz de otimizar a tarefa de escalonamento. Tal ordem é mostrada na Figura 6.5.

A Figura 6.5 apresenta dois planos de ações que constroem o recurso *Firebat*. Os planos possuem as mesmas ações, entretanto o plano (B) apresenta menor *makespan* que o plano (A). Isso ocorre, porque no momento em que o método *ConstroiLink*() insere uma nova ação no plano de ações (linha 10), ele imediatamente chama seu próprio método para verificar as pré-condições dessa nova ação (linha 11). Durante essas verificações, a prioridade do algoritmo no plano (B) é sempre de resolver as pré-condições de recursos não renováveis por último. Assim uma ação sempre tem a execução dessas ações num momento anterior a sua própria execução. Com essa estratégia, o plano (B) possui a ação de *collect-gas* e *collect-minerals* sendo executada o mais próximo possível do tempo de execução da ação *build-firebat* que constrói o recurso meta estabelecido. Assim, as demais ações como *build-barracks* e *build-academy* puderam ser realizadas pelo mesmo recurso *Scv*, onde esse consegue agrupar suas tarefas finalizando um ação e imediatamente iniciando outra. No plano (A), as ações não renováveis que produzem os recursos do *Firebat* estão sendo executadas muito antes dele, e isso introduz espaços de tempo devido as execuções que são feitas muito antes da própria ação que as utiliza. Essas execuções não permitem que determinadas ações possam ser agrupadas e realizadas por um mesmo recurso. Ambos os planos (A) e (B) foram obtidos pelo POPlan, mas o plano (A) foi executado sem as estratégias de ordem de execução descritas. No plano (A) em diversos momentos os recursos *Scv* ficam ociosos a espera do término da execução de outras ações que são pré-condições da ação que esse irá executar. Tais espaços não tiram proveito dos tempos de início e término das ações, comportamento oposto de um jogador humano experiente.

O Algoritmo 13 verifica para cada ação do plano (linhas 3 e 4) se seus recursos predecessores estão presentes no plano ou devem ser inseridos nele. Essa verificação é feita pelo *ChecaPredecessores*(*Act*, *Rsc*, *Actn*) (linha 6), que verifica se existe uma ação no plano que pode satisfazer a pré-condição ou se é preciso inserir uma nova ação. Caso seja necessária uma nova ação, é atribuído o valor *false* a variável *existe* e o algoritmo cria os *links* entre a ação e sua pré-condição, coloca a nova ação no plano e chama a si mesmo para inserir as pré-condições da nova ação (linhas 7 a 11). Caso a ação que atende a pré-condição exista, então são criados apenas os *links* necessários entre ela e a ação que ela satisfaz (linhas 12 a 14).

A função *ChecaPredecessores*() recebe a variável *Actn* (criada dentro do algoritmo 13) como parâmetro e atribui a ela a nova ação que será inserida no plano ou a ação que

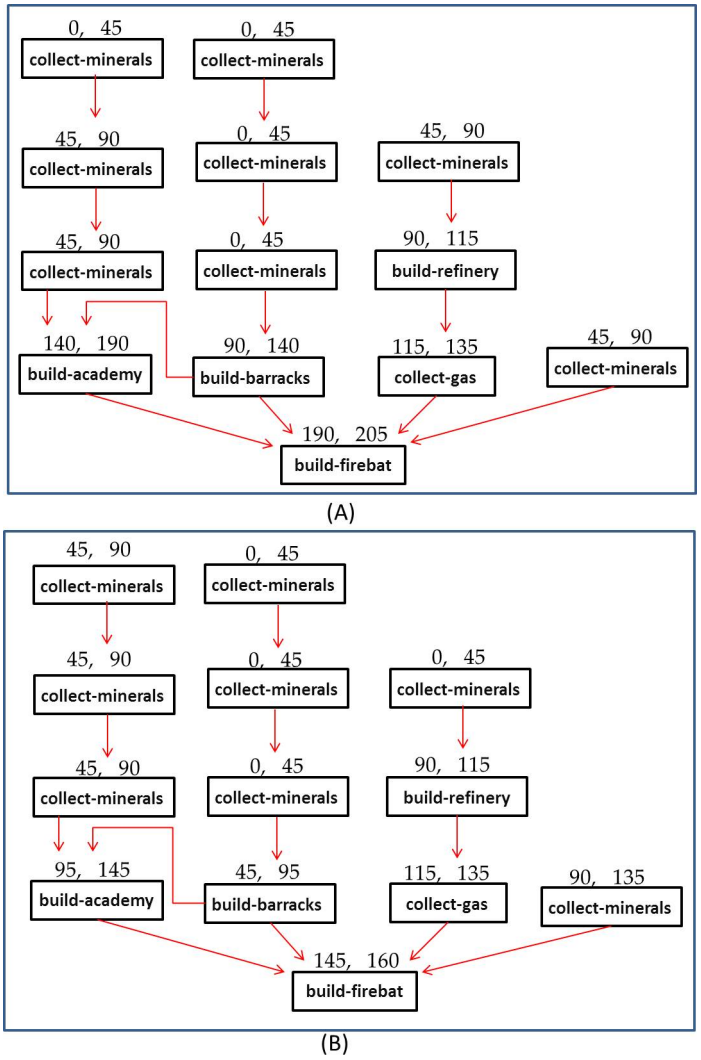

Figura 6.5: Comparação entre dois planos que alcançam o recurso *Firebat* utilizando o POPlan

já está lá e pode satisfazer a pré-condição necessária. O algoritmo valida o atributo *existe* de cada ação que teve todas suas pré-condições satisfeitas. No final, o plano contendo as ações e os *links* entre elas é retornado para o POPlan continuar sua execução. O plano que o Algoritmo 13 constrói e retorna para o POPlan, contém apenas as ações para chegar até o recurso meta e os respectivos *links* entre elas. Este plano permite que o escalonador consiga encontrar o melhor tempo para iniciar a execução de cada ação, sem necessidade de efetuar tarefas específicas de planejamento para obter informações sobre as ações. Informações essas que auxiliam no escalonamento, mas que já estão presentes no plano construído até agora.

O POPlan após configurar o plano com as ações necessárias e os links através do $ConstroiLink()$, começa a percorrer as ações do mesmo (linha 2 do Algoritmo 12) para configurar seus atributos. O algoritmo verifica se a ação não é do tipo unidade de produção, ou seja, se a ação é do tipo *Minerals* ou *Gas*, únicos recursos que não são do tipo unidade. Caso seja, a função $Quantidade(Act)$ (linha 5) é chamada para verificar quantas

vezes a ação deve ser executada. Por exemplo, se a ação *build-minerals* tiver um *link* para uma *build-academy* que necessita de 150 $Minerals$ é necessário três execuções da ação, pois cada execução gera 50 $Minerals$. Essa situação ocorre apenas com ações relativas a recursos não renováveis. O algoritmo $ConstroiLink()$ insere sempre uma ação de recurso não renovável no plano como pré-condição de uma outra ação. Assim, quando é necessária a execução dessa pré-condição mais de uma vez o algoritmo no seu decorrer insere mais ações se necessário baseado na quantidade de recursos existentes e disponíveis no plano. Cada nova ação que é inserida recebe o seu devido tempo de início e fim de execução. Essa inserção de nova ações será melhor descrita a seguir.

A função $BuscaTMinimo(Act)$ (linha 8) encontra o tempo em que uma ação pode ter sua execução iniciada (tempo mínimo), ou seja, verifica qual é o tempo de término do predecessor mais demorado dessa e estabelece esse como sendo o tempo mínimo para ela começar a ser executada. Após isso, as ações do plano são percorridas para que sejam encontrados os recursos base $Act.rbase$ da ação dentro do plano. Toda vez que a ação correspondente a um recurso base é encontrada, o método $Escalona(Plan, ActRs, Act, tmpInicio, melhorTempo)$ (linha 12) é utilizado. Ele percorre a lista que contém as ações executadas pelo recurso base e tenta encontrar um intervalo de tempo entre essas para que a ação atual possa ser executada. A variável $melhorTempo$ (linha 9) é quem armazena o tempo em que a ação vai ser escalonada, ela recebe a constante MAX que inicialmente atribui um alto valor a ela para que seja usada no algoritmo de escalonamento, sendo atualiza com os melhores tempos no decorrer da execução. O método de escalonamento usa uma estratégia para encontrar o menor intervalo de tempo de execução entre os recursos base que estão no plano e que podem executar a ação. Essa estratégia utiliza a variável $tmpInicio$ para informar o tempo mínimo para o início da ação mais o tempo que ela leva para ser executada. Assim, busca-se o melhor escalonamento possível para as ações. O método de escalonamento será descrito no final da seção.

Após definir o tempo de início da ação através da função de escalonamento, o POPlan agora irá finalizar a configuração dos seus atributos e sua atribuição no plano novamente utilizando a etapa de planejamento. O método $Constroi(Plan, Act, melhorTempo)$ (linha 15) é responsável por essa tarefa. Esse configura os tempos iniciais e finais de execução da ação baseado no valor do parâmetro $melhorTempo$, insere novas ações de recursos renováveis caso seja necessário, modifica algum *link* da ação se for necessário devido ao escalonamento que pode alterar tais *links*, confere também se não há alguma ameaça nesses (segundo o conceito de POP), além de verificar se o tempo limite do plano não foi excedido.

Quando a variável $vezes$ requer que uma ação de recursos não renováveis seja executada mais de uma vez, o método $Constroi(Plan, Act, melhorTempo)$ insere a(s) nova(s) ação(ões) necessária(s). O método primeiramente configura a ação que já está no plano, e se necessário em seguida insere uma nova ação que é identica a que já foi configurada,

mas será escalonada em seguida pelo algoritmo de escalonamento que é invocado novamente pelo *loop* de controle (linhas 7 a 20). Essa estratégia permite que seja inserido o mínimo de ações necessárias para satisfazer as pré-condições das ações presentes no plano. Por exemplo, a ação *build-refinery* necessita de duas ações *collect-minerals* para reunir os 100 *Minerals* que são pré-condições para sua execução, mas se em determinado momento existirem 50 *Minerals* disponíveis no plano, será necessário executar apenas uma ação *collect-minerals* para reunir a quantidade necessária.

O método $Constroi()$ também verifica se não há ameaças nos *links* das ações devido ao uso do escalonador. Esse quando utilizado pode ter escolhido outro recurso base para a ação que tenha um intervalo de tempo menor em relação ao recurso que já possuia um *link* para executá-la definido pelo $ConstroiLink()$. Assim, é necessário refazer tal *link*. Após configurar os atributos da ação o algoritmo verifica se o tempo limite não foi excedido. Caso seja excedido, o método finaliza o planejamento com a última ação que foi configurada e retorna o plano de ações final usando a função $Sair()$. Esse plano será passado ao SA como plano inicial. Caso o *makespan* do plano construído para atingir o recurso meta não tenha excedido o tempo limite, um novo recurso escolhido aleatoriamente é passado para o planejador continuar o planejamento repetindo os processos descritos do POPlan.

O plano final que é construído pelo POPlan, contém as ações que atingem uma quantidade de recursos aleatória, além de ter todas as ações escalonadas.

Escalonamento

O método $Escalona(Plan, ActRs, Act, tmpInicio, melhorTempo)$ (linha 12 do Algoritmo 12) é quem faz o escalonamento de todas as ações que estão presentes no plano. O método recebe o plano de ações $Plan$, a ação responsável por produzir o recurso base da ação que vai ser escalonada $ActRs$, a própria ação que vai ser escalonada Act, o tempo mínimo em que ela pode ser iniciada $tmpInicio$ e por último a variável responsável por armazenar o tempo em que ação vai ser iniciada (escalonada) $melhorTempo$. A estratégia do escalonador é de encontrar um intervalo de tempo entre as ações que estão sendo executadas pelo recurso base da ação que será escalonada. Tal intervalo dever ser suficiente para a execução da ação e também deve ser o mais cedo possível. Essa estratégia foi desenvolvida com base na estrutura de execução de ações de jogos RTS e nas características da busca por metas aqui proposta.

Em um jogo RTS, alguns recursos são responsáveis por executar as ações dentro do jogo, por exemplo, o recurso *Scv* é responsável por executar a ação *build-refinery*. Assim, dentro do jogo os recursos mantêm os tempos em que eles estão executando ações. Entretanto, entre essas execuções podem surgir intervalos de tempo em que recursos ficam ociosos ou esperando por outro recurso ser finalizado antes de inicar sua execução. Esses representam o tempo em que o recurso fica ocioso. Com as características dinâmicas

da busca por metas, esses intervalos tendem a surgir com muito mais frequência devido às mudanças que são feitas no plano. Com isso, a busca por intervalos torna-se uma estratégia eficiente, tanto em termos de resultados quanto em tempo de execução (*run-time*). O Algoritmo 14 apresenta o pseudocódigo do escalonador. Esse foi desenvolvido utilizando alguns conceitos e ideias dos escalonadores dos trabalhos de [Chan et al. 2007] e [Branquinho et al. 2011b].

Algoritmo 14 Escalona(*Plan*, *ActRs*, *Act*, *tmpInicio*, *melhorTempo*)

```
1:  intervalo ← tmpInicio
2:  tempoEscalona ← 0
3:  for each Ação ActExe ∈ ActRs.acoesExecutadas do
4:     if intervalo + Act.bTime <= ActExe.iniTime or intervalo >= ActExe.endTime
       then
5:        tempoEscalona ← intervalo
6:     else
7:        intervalo ← ActExe.endTime
8:        tempoEscalona ← intervalo
9:     end if
10: end for
11: if tempoEscalona < melhorTempo and tempoEscalona <= ActRs.endTime then
12:    melhorTempo ← tempoEscalona
13: end if
14: return melhorTempo
```

O algoritmo percorre a lista *acoesExecutadas* (linha 3), que representa as ações executadas pelo recurso base *ActRs* da ação que vai ser escalonada *Act*. Nessa busca, são visitadas todas as ações que o recurso base está executando para que seja encontrado um intervalo de tempo. A condição seguinte ao *loop* (linha 4) é responsável por verificar se existe um intervalo de tempo compatível. Ela verifica se a variável *intervalo* que inicialmente tem o valor mínimo para o a ação começar a ser executada (*tmpInicio*), mais tempo de execução da ação que vai ser escalonada são menores que o tempo inicial de execução da ação que já está sendo executada pelo recurso base. Se for, isso significa que a ação pode ser encaixada em um intervalo que começa no tempo que está armazenado em *tempoEscalona*, e termina exatamente antes do início de execução da ação que já está sendo executada *ActExe*. Caso essa condição falhe é verificada a condição em que o valor de *intervalo* for maior que o tempo final de execução da ação que já está sendo executada *ActExe*. Isto significa que a ação pode ser executada em um intervalo que começa exatamente depois que essa ação(*ActExe*) termina de ser executada.

A condição que verifica qual o melhor intervalo (linha 4), quando não encontra algum intervalo válido antes ou depois da ação que está sendo executada, configura o valor de final de execução dessa para a variável *intervalo*. Assim, quando uma ação não deixa um intervalo válido o seu tempo de término funciona como parâmetro para a busca continuar. A próxima ação verificada passa pelo mesmo processo até que se atinja o fim da lista de

ações executadas e o valor de escalonamento será o valor após a execução da última ação dessa lista. Essa busca mostrou-se eficiente, uma vez que o número de ações executadas não é muito grande e é preciso apenas uma iteração por cada recurso base da ação, entretanto experimentos e análise dos tempos de execução do algoritmo devem ser feitos para que tal eficiência possa ser ratificada. As ações executadas devem estar ordenadas de acordo com o tempo de início de execução. A Figura 6.6 ilustra um exemplo desse processo de busca por intervalos entre as ações executadas por um recurso base.

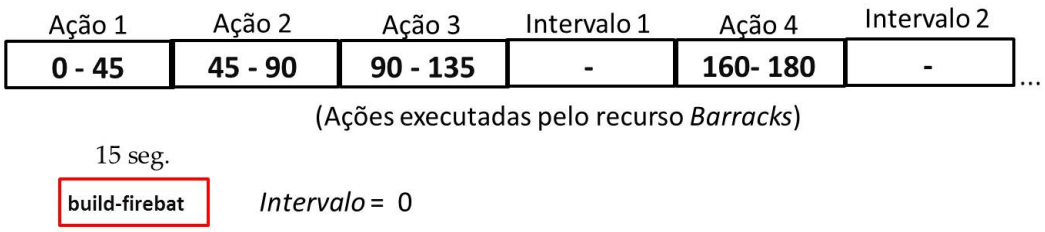

Figura 6.6: Exemplo do sistema de busca por intervalos do escalonador - Passo 1.

Na Figura 6.6, é apresentado um conjunto de ações executadas pelo recurso *Barracks*, a ação a ser escalonada (*buid-firebat*) entre as ações que estão sendo executadas e o valor da variável *intervalo*. A ação precisa de um intervalo de 15 segundos para ser executada, e a variável *intervalo* indica que a ação pode ser executada a partir do tempo 0. A Figura 6.7 mostra as duas primeiras tentativas para escalonar a ação *build-firebat*. Na primeira tentativa o algoritmo percebe que ação não pode se encaixar no intervalo que inicia antes do tempo inicial de execução da *Ação 1* e nem no tempo que se inicia depois do tempo de término dessa. Assim, o valor de *intervalo* assume o valor de término de execução da *Ação 1*, que tenta encontrar um intervalo na próxima ação (*Ação 2*), onde não é possível escalonar novamente.

A Figura 6.8 mostra quando o algoritmo encontra um intervalo válido baseado no valor de *intervalo* que agora é de 135 relativo ao tempo de término de execução da *Ação 3*. Ele verifica que valor de *intervalo* mais o tempo de execução da ação *build-firebat* é menor que o tempo de início de execução da *Ação 4*. A ação pode ser escalonada para terminar antes do tempo de início da *Ação 4*. Esse processo é feito para todas as ações do plano antes que seus atributos sejam configurados, pois o valor de início das ações é feito pelo algoritmo que escalona todo o plano.

De fato, a estratégia de busca por intervalos é interessante uma vez que esses estão em constante mudança de valores e posições durante o planejamento. Com isso, em poucas iterações é possível verificar todos os intervalos e detectar mudanças entre eles caso tenha ocorrido alguma. Contudo, uma análise do algoritmo e experimentos podem confirmar tais expectativas. O algoritmo, entre as linhas 11 e 13 executa uma última verificação para garantir que o melhor intervalo encontrado entre todos os recursos base da ação seja ar-

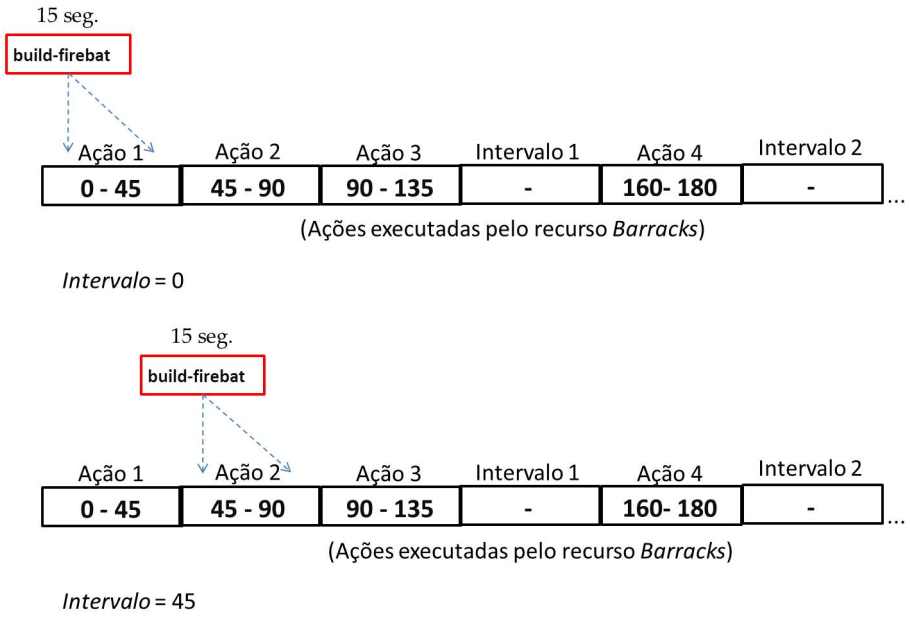

Figura 6.7: Exemplo do sistema de busca por intervalos do escalonador - Passo 2 e 3.

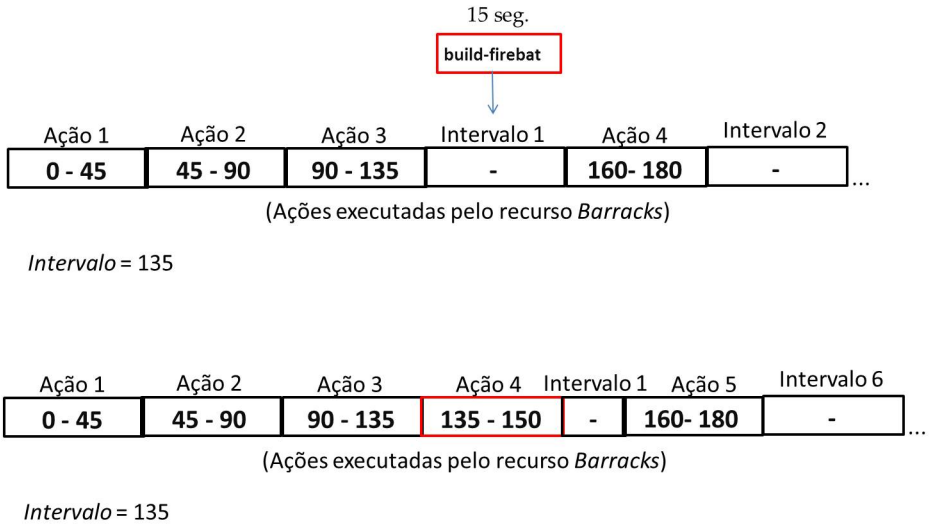

Figura 6.8: Exemplo do sistema de busca por intervalos do escalonador - Etapa final.

mazenado. Nessa, verifica-se o valor do intervalo é melhor do que o que já está armazenado em $melhorTempo$, e também se esse valor não é menor que o tempo de término do próprio recurso base que o forneceu. É possível que existam intervalos de tempo menores que o tempo em que o próprio recurso é construído, os quais devem ser desconsiderados. As ações que são executadas por um recurso estão sempre ordenadas por ordem crescente de tempo inicial de execução, uma vez que essa ordenação valida a busca adotada pelo escalonador. Essa ordenação é feita pelo método $Constroi(Plan, Act, melhorTempo)$ do

POPlan, que utiliza o algoritmo Mergesort [Russell e Norvig 2003] para tal ordenação.

6.2 SHELRChecker - Verificador de Consistência Escalonado

A busca por metas agora vai em direção a planos de ações que tenham qualidade na quantidade de força de ataque e ao mesmo tempo tenham suas ações escalonadas. A etapa de construção e validação de novas soluções tornam-se ainda mais complexas devido à paralelização de ações. O verificador de consistência desenvolvido denominado SHELRChecker é utilizado nesta etapa. O Algoritmo 15 apresenta o pseudocódigo do SHELRChecker.

Algoritmo 15 SHELRChecker($Plan$, $operacao$, T_{limite}, $penalti$)

```
1: Invib(AcoesInv, operacao, penalti)
2: for each Ação Act ∈ AcoesInv do
3:    if Act.viavel = true then
4:       for each Ação Actl ∈ Act.link do
5:          DesfazL(Actl, AcoesInv, penalti)
6:       end for
7:       for each Ação Actl ∈ Act.Clink do
8:          LiberaL(Actl, AcoesInv, penalti)
9:       end for
10:   end if
11: end for
12: Rearranja(Plan, AcoesInv, operacao)
13: continua ← true
14: viavel ← true
15: while continua = true do
16:   tmpInicio ← 0
17:   for each Ação Act ∈ AcoesInv do
18:      viavel ← Reunir(Plan, Act, penalti, tmpInicio)
19:      if viavel = true then
20:         for each Ação ActRs ∈ Plan do
21:            if ActRs = Act.rBase then
22:               mTempo ← Escalona(Plan, ActRs, Act, tmpInicio, mTempo)
23:               continua ← Constroi(Plan, Act, mTempo)
24:            end if
25:         end for
26:      end if
27:   end for
28: end while
29: return Plan
```

O algoritmo recebe como parâmetros a atual solução do SA $Plan$, a operação que vai ser realizada no plano $operacao$, o tempo limite da solução que vai ser gerada T_{limite} e por

último a variável *penalti* para que seja definido o seu valor. Inicialmente é chamado o método $Invib(AcoesInv, operacao, penalti)$ (linha 1) responsável por verificar quais ações vão ficar inviáveis devido a operação. O método coloca as primeiras ações dentro da lista de ações inviáveis $AcoesInv$, que são necessárias para o algoritmo descobrir quais outras vão ficar inviáveis também. Essas primeiras ações que compõe a lista $AcoesInv$ são as que participam diretamente de operação para gerar uma nova solução, seja a nova ação a ser inserida no plano ou as duas que trocaram de posição. O algoritmo sempre coloca ambas como inviáveis, pois é mais eficiente do que verificar ambas individualmente, uma vez que a segunda etapa do algoritmo é dedicada a verificar quais ações podem torna-se viáveis novamente.

O algoritmo entre as linhas 2 e 11 encontra as demais ações que vão ficar inviáveis. Para cada uma que ainda possui o atributo *viavel* válido (linha 3), ou seja, cada ação que acabou de ser inserida na lista, o algoritmo chama os métodos responsáveis por eliminar seus *links* e descobrir quais outras ações ficarão inviáveis também. O método $DesfazL(Actl, AcoesInv, penalti)$ (linha 5) é chamado para colocar cada recurso que recebe um *link* da ação que ficou inviável *Act* dentro da lista de ações inviáveis. Isso é feito uma vez que *Act* que é pré-condição dessas ações não será mais executada. Por exemplo, se uma ação *build-refinery* ficar inviável quer dizer que o recurso *Refinery* não existe mais no jogo, assim todos as ações *collect-gas* que recebem um *link* desse recurso ficaram inviáveis também.

O método $LiberaL(Actl, AcoesInv, penalti)$ (linha 8) é chamado logo depois do $DesfazL()$, ele é responsável por liberar as ações que estavam servindo de pré-condição para a ação que tornou-se inviável, assim eliminando os *links* que chegavam até essa. Isso libera os recursos usados pela ação para que sejam usados por outras ações. No exemplo anterior, se a ação *build-refinery* ao ficar inviável fez com que os recursos do tipo gás ficam inviáveis também; em contrapartida ela liberou os recursos do tipo minerais utilizados para construí-la. Também foi liberado o tempo gasto pelo *Scv* para construir o recurso *Refinery*, que pode ser usado para executar uma outra ação.

Depois que a lista de ações inviáveis já foi totalmente preenchida, o algoritmo chama o método $Rearranja(Plan, AcoesInv, operacao)$ (linha 12) . Ele executa a operação sobre o plano. Ele também coloca a nova ação que entrou no plano ou as ações que foram trocadas de lugar no topo da lista de ações inviáveis. Isso é feito para que elas possam ser verificadas antes das demais ações inviáveis na segunda etapa do algoritmo, uma vez que tem prioridade por serem as ações envolvidas diretamente na construção de uma nova solução.

Da linha 15 em diante, o algoritmo concentra-se em verificar quais ações podem torna-se viáveis novamente, configurar seus atributos e escaloná-las. Após começar a percorrer a lista de ações inviáveis a função $Reunir(Plan, Act, penalti, tmpInicio)$ (linha 18) é chamada. Ela busca pelas pré-condições da ação que está sendo verificada *Act*, se as

ações estão disponíveis para serem usadas para executá-la. A função percorre o plano em busca de ações viáveis, se alguma é viável e não está com *link* para outra ação então será utilizada para viabilizar essa. Quando todas as pré-condições são encontradas e podem ser utilizadas, a variável *penalti* diminui seu valor, pois alguma ação voltará a ser viável. A variável *tmpInicio* verifica o tempo de término da pré-condição que termina de ser executada mais tarde, como parâmetro para o algoritmo de escalonamento. Caso não sejam encontradas as ações de pré-condição, a função retorna o valor negativo que significa que a ação não pode tornar-se viável ainda.

Quando as pré-condições de uma ação são encontradas dentro do plano, o algoritmo escalona e configura os seus atributos, já que ela voltará a ser viável. É feita uma busca pelos recursos base da ação (linha 20) para definir qual deles irá executar a ação. Quando um recurso base é encontrado o algoritmo de escalonamento é chamado $Escalona(Plan$, $ActRs$, Act, $tmpInicio$, $mTempo)$ (linha 22). O Algoritmo 14 é o mesmo utilizado no POPlan. Ele executa o escalonamento para cada ação que volta a ser viável. Neste momento o algoritmo torna-se muito útil, devido ao seu desempenho e tempo de resposta, uma vez que ele deve ser executado diversas vezes em cada uma das iterações presente nas execuções do SA.

Por fim, a função $Constroi(Plan, Act, tmpInicio)$ (linha 23) é chamada para configurar os atributos da ação que voltou a ser viável. Ela seta os tempos de execução das ações, constrói todos os *links* entre ela e suas pré-condições, retira a ação da lista de inviáveis e verifica se o tempo limite não foi atingido. Caso esse tempo seja atingido, a ação não volta a ser viável e o verificador interrompe o processo retornando o plano com a última ação que tornou-se viável, sendo esse a nova solução para o SA.

Capítulo 7

Experimentos e Discussão dos Resultados

Para análise e discussão dos resultados obtidos, será apresentado o ambiente de testes utilizado. Foram realizados testes de performance e comparação com resultados obtidos por jogadores humanos. Ambos os testes foram realizados dentro do próprio ambiente do jogo *StarCraft*. Para isso, foi utilizado um computador *Intel Core i7 1.73 Ghz* com 8Gb de memória *ram*, sendo executado com o sistema operacional *Windows 7*. A maioria dos testes apresenta resultados baseados em uma média de 10 execuções dos mesmos. O tempo dos testes é medido e apresentado em segundos, para que se tenha uma estimativa clara em relação ao tempo utilizado pelos algoritmos dentro do ambiente do *Starcraft*.

Para utilizar os algoritmos dentro do ambiente do jogo foi usado a *BWAPI* (BroodWar API) [Bwapi 2011], capaz de injetar os algoritmos dentro do jogo e coletar os resultados que são obtidos. A *BWAPI*[1] permite gerenciar as unidades e recursos do jogo *StarCraft*, bem como reunir informações e coletar resultados do próprio ambiente do jogo. No decorrer da seção a estratégia para o uso da abordagem de escolha de metas dentro do jogo será apresentada.

Os procedimentos usados nos testes são: $SA(S)$, $SA(E)$, $Jogador(E)$, $Jogador(M)$, $Jogador(I)$. Estes são apresentados na Figura 7.1.

O $SA(S)$ consiste na abordagem de escolha de metas aqui desenvolvida, utilizando a arquitetura sequencial. O método primeiramente executa o planejador sequencial com o Algoritmo 9, logo em seguida o SA (Algoritmo 8) é executado e utiliza o Algoritmo 11.

O $SA(E)$ representa a abordagem aqui desenvolvida utilizando a arquitetura com escalonamento. Inicialmente é executado o POPplan (12), que produz o plano de ações inicial que é utilizado em seguida com o SA (8) juntamente com o SHELRChecker (15).

[1] A *Brood War Application Programming Interface* (*BWAPI*), é uma *API* de código aberto em C++ para criar e testar módulos de IA no jogo *Starcraft:Broodwar*. Usando *BWAPI* é possível recuperar informações dos jogadores, informações das unidades e comandar os recursos desenvolvidos. A *BWAPI* é o sistema de customização e teste de IA oficial da conferência AIIDE.

SA(S)
SA - Simulated Annealing adaptado para jogos RTS (Algoritmo 8).
Planejador Sequencial desenvolvido (Algoritmo 9).
Verificador de Consistência Sequencial (Algoritmo 11).
SA(E)
SA - Simulated Annealing adaptado para jogos RTS (Algoritmo 8).
POPlan - Planejador de Ordem Parcial com Escalonamento (Algoritmo 12).
SHELRChecker - Verificador de Consistência Escalonado (Algoritmo 15).
Jogador(E)
Jogador humano de nível experiente no jogo *StarCraft*
Jogador(M)
Jogador humano de nível médio no jogo *StarCraft*
Jogador(I)
Jogador humano de nível inicial no jogo *StarCraft*

Figura 7.1: Resumo dos procedimentos usados nos experimentos.

Ao fim é retornada a meta a ser atingida juntamente com o plano de ações para alcança-la.

Os procedimentos $Jogador(E)$, $Jogador(M)$ e $Jogador(I)$ correspondem ao jogadores humanos que jogaram o *StarCraft* para que suas performances fossem coletadas e usadas para comparações. $Jogador(E)$ refere-se a um jogador experiente do jogo, $Jogador(M)$ a um jogador de nível médio e $Jogador(I)$ a um jogador iniciante.

No Experimento 1 (7.1) são utilizados o $SA(S)$ e o $Jogador(E)$. O experimento consiste em comparar os resultados obtidos pelo SA utilizando arquitetura sequencial e um jogador experiente de *StarCraft*, em relação a força do exército obtido por cada um. Ambos executaram suas performances no ambiente do *StarCraft*. Neste experimento, são definidos vários tempos limite para ambos (esquerda da tabela), e dentro desses tempos eles devem procurar atingir uma meta através de produção de recursos que maximize seus respectivos exércitos. Abaixo de cada um dos procedimentos ($Jogador(E)$ e $SA(S)$ em negrito está a quantidade de pontos de ataque do exército desenvolvido por cada um deles, a direita na tabela está o valor do *makespan* das metas encontradas pelo $SA(S)$ e o *runtime* (tempo de execução) gasto por esse também. Os pontos de ataque são atributos presente em cada recurso do jogo e são utilizados para quantificar a força do recurso em relação a sua capacidade de ataque.

Tabela 7.1: Resultados do Experimento 1 utilizando a arquitetura sequencial

Tempo limite	Jogador(E)	SA(S)	*Makespan* do SA(S)	Runtime do SA(S)
200 seg.	12	12	198 seg.	35,5 sec.
400 seg.	31	**34**	392 seg.	80,2 seg.
600 seg.	47	**52**	586 seg.	205,4 seg.
800 seg.	**109**	102	800 seg.	386,8 seg.
1000 seg.	141	**146**	990 seg.	555,1 seg.
1200 seg.	**204**	199	1189 seg.	696,3 seg.
1400 seg.	**241**	236	1388 seg.	812,1 seg.

Ainda no experimento 1, o $SA(S)$ conseguiu superar o jogador em três ocasiões, empatando uma e perdendo outras três. Para cada um dos intervalos de tempo limite, foram feitas 10 execuções do algoritmo e o jogador também jogou o jogo esse número de vezes. Por fim, foram feitas médias dos pontos de ataque dos exércitos obtidos por ambos em cada uma dessas execuções. É possível notar que o $SA(S)$ em metas com intervalo de tempo médio (200 a 600 segundos) conseguiu melhores resultados, pois com esses tempos estratégias de batalha ainda não são fatores cruciais para vitória no jogo. Nesses intervalos de tempo, uma produção de recursos que aumente a força dos recursos produzidos em geral é um fator chave para vitória. Porém, em intervalos de tempo maiores o $SA(S)$ não conseguiu superar o jogador. Nos experimentos utilizando a arquitetura sequencial, o jogador humano seguiu o princípio de execução de ações em ordem linear, compatível com a arquitetura do $SA(S)$.

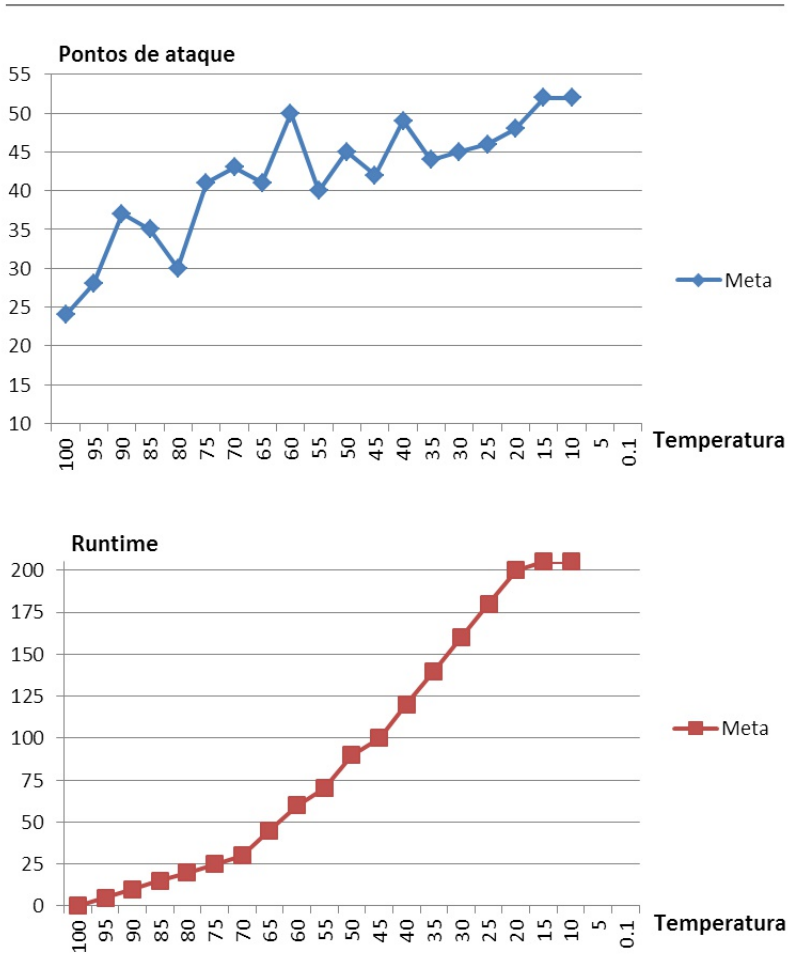

Figura 7.2: Gráficos de pontos de ataque e *runtime* para um dos testes do experimento 1.

A Figura 7.2, mostra os respectivos gráficos de pontos de ataque e *runtime* de uma das 10 execuções do Experimento 1 (7.1), com intervalo de tempo de 600 segundos. No gráfico que ilustra os pontos de ataque, o SA apresenta o comportamento esperado em relação aos valores de temperatura. Com altos valores o algoritmo oscila aceitando diversos

planos. Quando a temperatura começa a diminuir essa oscilação é menor até alcançar sua convergência. Neste exemplo, o algoritmo não chega até a temperatura mínima possível (0.1), pois foi invocado um dos critérios de parada já que o algoritmo atingiu um valor próximo do ótimo e não aceitou mais nenhum plano durante diversas iterações. Em relação ao gráfico de *runtime*, o algoritmo quando está em temperaturas mais baixas tende a gastar mais tempo, devido ao critério de seleção que torna-se mais rígido.

Tabela 7.2: Resultados do Experimento 2 utilizando o $SA(S)$

Tempo Limite	450 sec.	750 sec.	1050 sec.
Valor ótimo de pontos de ataque	41	92	156
Quantidade de execuções que estão 95% do ótimo	60%	50%	40%
Quantidade de execuções que estão 90% do ótimo	30%	40%	35%
Valor médio de *runtime*	89,3 seg.	371,8 seg.	571,2 seg.
Valor médio obtido	34	79	144
Total de execuções	15	15	15

A Tabela 7.2 apresenta os resultados do Experimento 2. Nesse, foram feitas 15 execuções do $SA(S)$ com três intervalos de tempo diferentes. O algoritmo consegue resultados com qualidade em intervalos de tempo de 450 segundos, sendo 60% desses próximos ou sendo o valor ótimo. Esses valoes são considerados ótimos baseados em resultados de jogadores experientes, pois esses compartilham seus *rankings* de produção de recursos em sites especializados na *internet*. Com esses valores, é possível descobrir a quantidade máxima de recursos que podem ser produzidos nesses intervalos. Em intervalos de tempo maiores, o algoritmo tem menor incidência de soluções próximas do valor ótimo, devido à quantidade de recursos que aumentam e necessitam de gerenciamento e validação.

Tabela 7.3: Resultados do Experimento 3 utilizando a arquitetura com escalonamento

Tempo limite	Jogador(E)	SA(E)	*Makespan* do SA(E)	Runtime do SA(E)
200 seg.	44	**46**	195 seg.	32,4 sec.
400 seg.	184	**190**	396 seg.	76,1 seg.
600 seg.	**325**	318	600 seg.	291,4 seg.
800 seg.	544	**546**	797 seg.	535,9 seg.
1000 seg.	**792**	789	998 seg.	777,2 seg.
1200 seg.	**912**	905	1199 seg.	952,8 seg.
1400 seg.	**1248**	1232	1395 seg.	1098,3 seg.

Os Experimentos 3 (7.3) e 4 (7.4), utilizam o $SA(E)$. No Experimento 3, novamente são comparados os valores de um jogador experiente com o algoritmo, entretanto dessa

vez ambos podem utilizar ao máximo os recursos do jogo devido ao paralelismo de ações. Os pontos de ataque obtidos neste experimento são muito superiores aos obtidos em experimentos anteriores, isso salienta a importância do escalonamento nesta abordagem. O algoritmo conseguiu superar o jogador em três ocasiões e foi superado em outras quatro. É interessante salientar que o algoritmo conseguiu os melhores resultados em intervalos de tempo médio (até 600 seg.), assim como a arquitetura sequencial. Nesses resultados o *runtime* do $SA(E)$ foi menor que o do $SA(S)$, mesmo com a complexidade trazida pelo escalonamento. Isso é devido a estrutura de *links causais* da abordagem de POP, que auxilia diminuindo a busca por ações e pré-condições durante o gerenciamento. Além disso, o algoritmo de escalonamento apresenta eficiência nos tempos de execução. Entretanto, com o crescimento dos intervalos de tempo limite, o *runtime* do $SA(E)$ aumenta bastante, superando a arquitetura anterior. Isso ocorre devido ao aumento no número de ações, que começa na geração da solução inicial pelo POPlan. Quando soluções são geradas pelo $SA(E)$ a complexidade na validação das mesmas aumenta junto com o número de ações que estão presentes.

O algoritmo tem sua performance avaliada na média de execuções que estão próximas do valor ótimo. O algoritmo não alcança o valor ótimo sempre, devido às estratégias de eficiência que foram desenvolvidas para que ele operasse sobre as restrições de um sistema de tempo real. Outro motivo, é que a abordagem explora um espaço de soluções muito grande em comparação com as referências citadas, onde se têm apenas as soluções entre um plano inicial e uma meta com quantidade fixa de recursos. Aqui, a solução está entre diversos planos que podem assumir o papel de solução final, onde à meta possui um conjunto de recursos indeterminado que deve ser encontrado dadas as restrições e condições impostas pelo jogo.

Na Figura 7.3, são apresentados os gráficos relativos aos pontos de ataque e *runtime* de um dos testes com intervalo de tempo de 600 segundos do Experimento 3 (7.3). Em relação aos pontos de ataque, o SA oscila bastante entre soluções em baixas temperaturas, tanto que em alguns casos ele chega a considerar planos de ações com valor inferior ao plano inicial. Isso ocorre devido à quantidade de ações que estão nos planos gerados, que é maior em relação ao mesmo intervalo de tempo do $SA(S)$. No $SA(E)$ as operações executadas nos planos de ações também geram mais alterações do que na arquitetura anterior, com isso as oscilações também aumentam. Em baixas temperaturas o algoritmo segue a tendência normal de aceitar em sua maioria soluções melhores que a atual. Em relação ao *runtime*, o algoritmo mantém a tendência de operar mais rápido quando a temperatura encontra-se em quantidades médias/altas. Em temperaturas menores ele despende mais tempo.

No Experimento 4 são apresentados os valores da análise de performance do $SA(E)$. Neste também foram feitas 15 execuções do algoritmo em cada um dos três intervalos de tempo. Foram obtidos bons resultados com os intervalos médios, com limite de 450 seg.

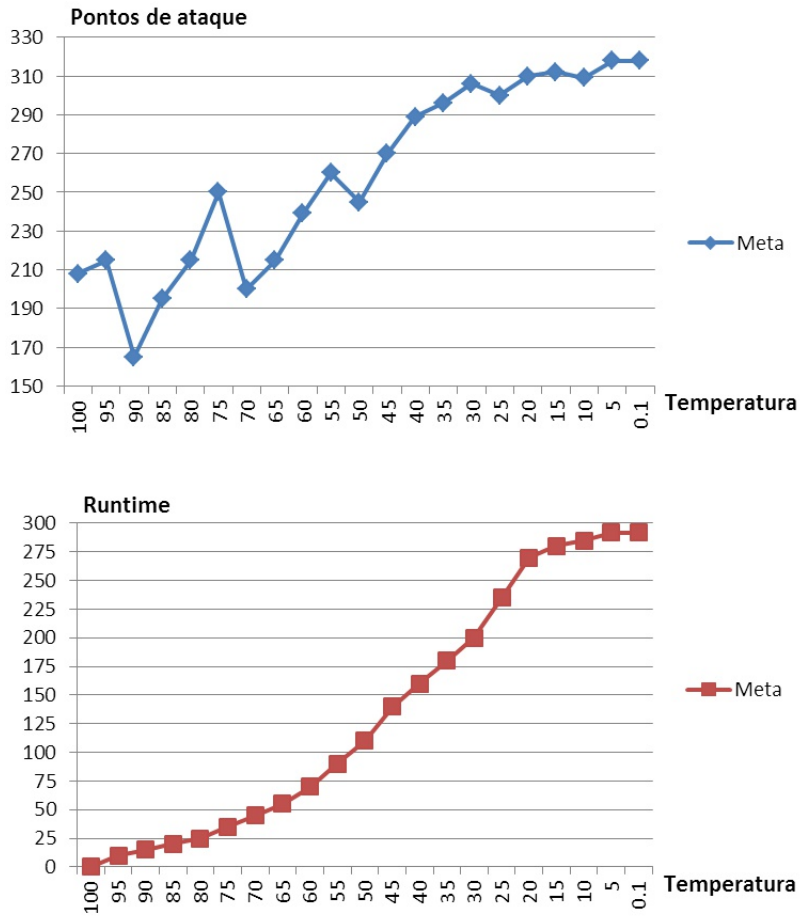

Figura 7.3: Gráficos de pontos de ataque e *runtime* para um dos testes do experimento 3.

foram obtidas soluções quase que somente a 90% e 95% do ótimo. Com intervalos de 750 seg., bons resultados foram atingidos também. Já com 1050 seg. o algoritmo teve mais dificuldades em manter os resultados como nos intervalos anteriores, devido ao aumento no número de ações.

Com os resultados mostrados até agora, fica evidente que a abordagem aqui proposta tanto na arquitetura com escalonamento quanto na sequencial consegue obter os melhores resultados em intervalos de tempo médio. Para que a abordagem pudesse ser utilizada dentro do ambiente do jogo e classificada como uma abordagem de tempo real, foi utilizada uma estratégia que tira vantagem dessa característica do algoritmo em relação aos resultados obtidos.

Em todos os testes o SA conseguiu obter uma solução em um *runtime* menor do que o *makespan* da mesma. Com isso, enquanto são executadas as ações de uma meta dentro do jogo, o algoritmo tem o tempo que será gasto para executar essas, disponível para encontrar uma próxima meta que será iniciada assim que essa for finalizada. Com essa estratégia, enquanto uma meta está sendo executada o algoritmo é capaz de encontrar a próxima meta, ou seja, uma meta futura que possui os recursos construídos pela meta

Tabela 7.4: Resultados do Experimento 4 utilizando o $SA(E)$

Tempo Limite	450 sec.	750 sec.	1050 sec.
Valor ótimo de pontos de ataque	216	497	819
Quantidade de execuções que estão 95% do ótimo	48%	39%	31%
Quantidade de execuções que estão 90% do ótimo	46%	41%	33%
Valor médio de *runtime*	79,7 seg.	498,1 seg.	792,5 seg.
Valor médio obtido	67	449	731
Total de execuções	15	15	15

anterior como estado inicial e será executada após essa. Essa estratégia consiste em decompor uma meta em submetas menores, onde essas têm intervalos de tempo em que o algoritmo consegue encontrar os melhores resultados mantendo eficiência no tempo de resposta. De fato, essa estratégia foi usada com sucesso nos testes em que era necessário executar metas acima de 180 segundos. Por exemplo, uma meta de 600 seg. (10 minutos) é decomposta em três metas de 200 seg, assim é necessário encontrar uma meta de 200 seg. e depois com os recursos que serão produzidos por essa é necessário uma próxima meta de 200 seg. e assim por diante. Sempre que uma meta é dividida em submetas, essas possuem intervalos de tempo entre 200 e 250 segundos.

Tabela 7.5: Resultados do Experimento 5

SA(E) utilizando única meta			**SA(E) utilizando submetas**			
Pontos de ataque	*Makespan*	*Runtime*	Pontos de ataque	*Makespan*	*Runtime*	
320	596 seg.	339,6 seg	317	591 seg.	289,1 seg.	**1**
301	594 seg.	344,9 seg	**324**	598 seg.	292,4 seg.	**2**
318	592 seg.	336,5 seg	314	599 seg.	290,8 seg.	**3**
313	590 seg.	348,1 seg	**319**	595 seg.	290,4 seg.	**4**
319	593 seg.	346,5 seg	**322**	597 seg.	291,2 seg.	**5**
324	596 seg.	342,8 seg	318	594 seg.	289,7 seg.	**6**
312	598 seg.	337,9 seg	**322**	600 seg.	290,1 seg.	**7**
309	588 seg.	349,4 seg	**323**	594 seg.	288,3 seg.	**8**
315	594 seg.	341,7 seg	313	596 seg.	293,6 seg.	**9**
302	593 seg.	347,2 seg	**318**	590 seg.	291,1 seg.	**10**

O Experimento 5 (7.5), demonstra a performance da estratégia de divisão de meta em submetas. Neste experimento, o $SA(E)$ foi executado dez vezes para encontrar uma meta com tempo limite de 600 seg. Para a comparação, a esquerda da tabela o algoritmo foi executado utilizando como parâmetro uma meta única de 600 segundos, o algoritmo parte de um plano inicial com esse *makespan*. Já na parte direita da tabela, foi utilizada

a estratégia de dividir a meta principal em submetas, o algoritmo dividiu a meta de 600 seg. em 3 submetas de 200 seg.

Analisando os resultados do experimento 5, o algoritmo utilizando submetas conseguiu encontrar na maioria das vezes soluções melhores e em todos as execuções em menor tempo de execução. Isso ocorre devido a submetas que contém quantidades menores de ações, o que facilita as operações do algoritmo evitando que durante o gerenciamento dos planos várias ações possam tornar-se inviáveis devido as mudanças no plano. Utilizando uma meta única, a quantidade de ações dentro do plano é alta e faz com que as soluções geradas fiquem complexas para o gerenciamento acarretando mais tempo de execução e maior dificuldade em convergência. Na estratégia de submetas, quando uma submeta tem sua execução finalizada suas ações são transformadas em recursos dentro do jogo, e fazem parte da próxima submeta que conterá apenas ações necessárias para desenvolver seus recursos. Assim, é garantido que cada submeta contém uma média próxima de ações ao invés de uma quantidade exagerada que pode dificultar o planejamento.

Tabela 7.6: Resultados do Experimento 6

Jogador(E)		**SA(E)**		
Pontos de ataque	Quantidade de recursos militares	Pontos de ataque	Quantidade de recursos militares	Vitórias em confronto
188	15	**190**	16	**SA(E)**
190	15	188	19	**SA(E)**
187	16	**188**	17	**Jogador(E)**
188	15	182	15	**Jogador(E)**
190	17	187	16	**Jogador(E)**
186	14	**191**	17	**SA(E)**
191	16	188	20	**SA(E)**
187	16	185	18	**Jogador(E)**
185	15	184	16	**Jogador(E)**
186	16	**191**	18	**SA(E)**

A Tabela 7.6 apresenta os resultados do Experimento 6. Nesse é feita novamente uma comparação entre o $Jogador(E)$ e o $SA(E)$. Foram feitas dez execuções do algoritmo com tempo limite de 400 segundos. Ao fim de cada execução, os recursos produzidos por ambos são colocados em uma batalha, sem interferência ou controle de algum dos jogadores. O $Jogador(E)$ obteve seis vitória sobre o $SA(E)$ em relação a força de exército. Nos combates diretos os exércitos produzidos pela $SA(E)$ conseguiram vencer mais vezes. Isso ocorreu nas vezes em que o algoritmo conseguiu produzir mais recursos militares que o jogador humano, pois a abordagem de busca por metas muitas vezes gera soluções que possuem maior quantidade de recursos com menor custo de produção, para preencher espaços de tempo no plano e utilizar recursos ociosos ou com pouco valor disponível dentro do jogo. Já o $Jogador(E)$, muitas vezes se concentrou em produzir recursos mais

poderosos, entretanto sua produção era feita em pequena escala de quantidade devido ao limite de tempo que foi imposto.

Tabela 7.7: Resultados do Experimento 7

Jogador(M)		**SA(E)**		
Pontos de ataque	Quantidade de recursos militares	Pontos de ataque	Quantidade de recursos militares	Vitórias em confronto
127	10	**188**	17	**SA(E)**
139	12	**191**	20	**SA(E)**
135	12	**189**	18	**SA(E)**
129	10	**184**	15	**SA(E)**
140	13	**172**	14	**Jogador(M)**
136	11	**185**	13	**SA(E)**
126	09	**181**	18	**SA(E)**
139	12	**182**	17	**SA(E)**
122	11	**187**	18	**SA(E)**
128	10	**191**	19	**SA(E)**

A Tabela 7.7 apresenta o Experimento 7, que envolve a comparação do $SA(E)$ com o $Jogador(M)$. Como no experimento anterior, foram feitas 10 execuções do algoritmo e o jogador de nível médio jogou o jogo essa mesma quantidade de vezes, ambos com tempo limite de 400 segundos. O $SA(E)$ conseguiu superar o $Jogador(M)$ em todas as execuções, obtendo metas com poder superior em termos de força de ataque. Nas batalhas travadas pelos exércitos produzidos em ambas as metas, destaque para o SA que obteve apenas uma derrota, mesmo com um exército mais poderoso. Essa se deu devido um recurso militar aéreo produzido pelo $Jogador(M)$ que não foi abatido pelo exército do $SA(E)$, pois nenhum tipo de recurso necessário para confrontar esse foi produzido. Com isso, a abordagem de escolha de metas consegue produção de recursos superior a um jogador de nível médio.

O Experimento 8 cujos resultados aparecem descritos na Tabela 7.8 mostram um comparativo com um jogador de nível iniciante. Neste, o $SA(E)$ conseguiu superar com facilidade o $Jogador(I)$ na produção de recursos com qualidade na força de ataque do exército. A quantidade de recursos produzida pelo algoritmo é muito superior, refletindo nas batalhas que foram travadas, onde os exércitos desenvolvidos pelo algoritmo conseguiram obter vitória em todos os testes.

No experimento 9 é feita novamente a análise da performance do $SA(E)$, mas agora com ênfase na geração de metas com intervalos de tempo médio. Na Tabela 7.9, são apresentados os resultados médios de 15 execuções com três diferentes intervalos de tempo. Para metas com intervalo de 250 segundos, a quantidade de soluções próximas do ótimo é grande, tal resultado é salientado pela média da pontuação de ataque (64) cujo valor é próximo do ótimo. Para intervalos de 400 segundos são obtidos bons resultados. Já no

Tabela 7.8: Resultados do Experimento 8

Jogador(I)		**SA(E)**		
Pontos de ataque	Quantidade de recursos militares	Pontos de ataque	Quantidade de recursos militares	Vitórias em confronto
75	7	**185**	14	**SA(E)**
81	8	**187**	16	**SA(E)**
71	7	**183**	15	**SA(E)**
68	7	**190**	20	**SA(E)**
79	6	**192**	19	**SA(E)**
82	9	**186**	19	**SA(E)**
77	8	**188**	17	**SA(E)**
65	6	**182**	18	**SA(E)**
78	7	**191**	21	**SA(E)**
80	8	**183**	17	**SA(E)**

Tabela 7.9: Resultados do Experimento 9 utilizando o SA(E)

Tempo Limite	250 sec.	400 sec.	600 sec.
Valor ótimo de pontos de ataque	71	191	327
Quantidade de execuções que estão 95% do ótimo	68%	40%	37%
Quantidade de execuções que estão 90% do ótimo	32%	47%	40%
Valor médio de *runtime*	47,2 seg.	77,6 seg.	289,1 seg.
Valor médio obtido	64	181	310
Total de execuções	15	15	15

intervalo de 600 segundos, começa ser apresentada uma queda no desempenho. Os resultados salientam a eficácia desta abordagem na busca por metas através de produção de recursos em intervalos de tempo menores, onde o algoritmo consegue melhores resultados. Tal característica também reforça o uso da estratégia de submetas.

A Figura 7.4 mostra o gráfico das 15 execuções do Experimento 9 para metas com tempo limite de 250 segundos. O gráfico mostra a quantidade de pontos de ataque da produção de recursos em relação ao *makespan* de cada meta encontrada. É possível notar novamente, que para esse intervalo de tempo foram encontrados diversos planos com valores próximos do ótimo.

No experimento 10 novamente é feito uma comparação entre o $SA(E)$ e o $Jogador(E)$ na escolha de metas através da maximização da produção de recursos. Entretanto, na Tabela 7.10 o tempo limite para a execução das ações dentro do jogo em cada um dos testes é de 250 segundos. Neste intervalo, o algoritmo consegue um dos seus melhores de-

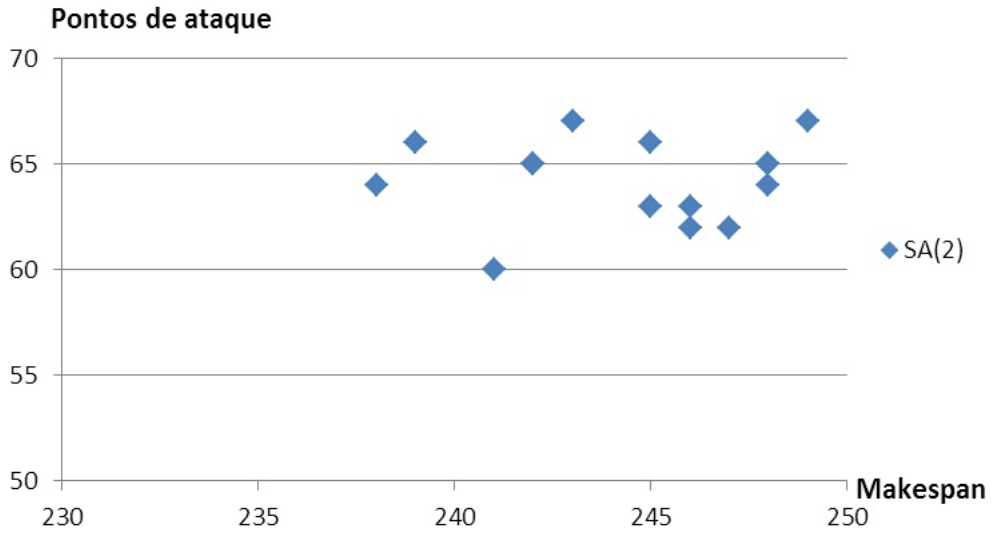

Figura 7.4: Gráficos de pontos de ataque e *makespan* das metas encontradas experimento 9.

Tabela 7.10: Resultados do Experimento 10

Jogador(E)		**SA(E)**		
Pontos de ataque	Quantidade de recursos militares	Pontos de ataque	Quantidade de recursos militares	Vitórias em confronto
76	7	72	7	**Jogador(E)**
75	8	**79**	7	**SA(E)**
76	8	**80**	8	**SA(E)**
79	8	**81**	9	**SA(E)**
73	6	**76**	6	**SA(E))**
80	7	73	6	**Jogador(E)**
81	9	71	6	**Jogador(E))**
76	6	**77**	8	**SA(E)**
74	6	74	7	**SA(E)**
81	8	75	6	**Jogador(E)**

sempenhos. Nos resultados, o algoritmo foi capaz de superar o jogador de nível experiente, conseguindo encontrar metas com maior quantidade de recursos e unidades militares. Em todos os confrontos entre os recursos, o exército do $SA(E)$ quando encontrou uma meta melhor conseguiu sair vitorioso. Entretanto, nos resultados onde os pontos de ataque das metas encontradas pelo $Jogador(E)$ eram maiores o exército do jogador saiu vitorioso.

Não foram considerados experimentos de comparação com as referências citadas neste trabalho. As referências mencionadas aqui possuem foco diferente da abordagem que foi proposta. Além disso, as técnicas utilizadas aqui e que possuem parte do desenvolvimento baseado nas referências mencionadas, tiveram sua estrutura e funcionamento alterados para adaptar-se a esta pesquisa. Tais adaptações fazem com que os critérios

para comparação sejam difíceis de obter. Assim, os testes focaram na análise e desempenho da abordagem proposta, juntamente com as comparações com jogadores humanos, que realizam tarefas de planejamento dentro do jogo que são semelhantes a proposta do trabalho.

Capítulo 8

Conclusão e Trabalhos Futuros

Nesta dissertação foi proposta uma abordagem para o problema de escolha de metas em jogos RTS. Essa representa uma proposta nova cujo foco não é explorado nas referências de planejamento para jogos RTS. Assim, existe pouco material para auxiliar em sua construção. O trabalho visa preencher uma lacuna identificada na maior parte das pesquisas citadas durante a dissertação, que não consideram de forma criteriosa a escolha de quais recursos devem ser produzidos, visando um melhor planejamento.

Foi proposto o uso do algoritmo de busca estocástica *Simulated Annealing*, uso e adaptação de planejadores baseados em técnicas tais como planejamento de ordem parcial e verificadores de consistência. Com isso, é construído um plano de ações inicial que atinge uma meta de recursos aleatórios baseado em um tempo limite para a execução das ações dentro do jogo. Em seguida, o plano é submetido ao processo de busca que altera a quantidade e ordem das ações que produzem os recursos para encontrar um plano final que maximize a força dos recursos produzidos. Esse plano final é então a meta a ser atingida junto com as ações necessárias para produzir os recursos dentro do intervalo de tempo estabelecido.

Com base nos resultados, a abordagem mostrou-se eficiente na busca por uma meta juntamente com seu plano de ações, sendo executada dentro do ambiente do jogo *StarCraft*. Nos testes realizados, o algoritmo foi capaz de competir com um jogador experiente na busca pela melhor meta dentro do jogo, superando-o em algumas ocasiões e principalmente em testes com tempo limite específicos. Em batalhas utilizando os exércitos produzidos, o algoritmo levou vantagem mesmo naqueles em que a força de seu exército era menor, devido à característica de produção de recursos de menor custo em maior escala. Com jogadores médios e iniciantes o algoritmo conseguiu ótimos resultados superando-os em todos os testes. O desempenho do algoritmo indica que utilizando os intervalos de tempo em que o algoritmo consegue seus melhores resultados é possível obter soluções próximas do valor ótimo, atingido até mesmo o ótimo em certas ocasiões.

O tempo de execução do algoritmo apesar de alto em alguns casos não caracterizou-se como um empecilho para o seu uso dentro do jogo. Em função da estratégia adotada, onde

uma meta tem seu intervalo de tempo reduzido em submetas para produção contínua de recursos. Enquanto uma meta era executada a sua sucessora já estava sendo produzida, sendo finalizada antes do término da atual. Essa característica foi um dos principais desafios a ser superado neste trabalho. Como trabalhos futuros para essa pesquisa pode-se destacar os seguintes complementos:

1. Explorar o uso de outros algoritmos de busca local, tal como algoritmo genético para executar mais testes de desempenho visando a melhoria da abordagem.
2. Explorar melhorias na escolha dos recursos que vão adentrar no plano de ações para a geração de novas soluções, de modo a definir prioridades que auxiliem a produção de recursos dependendo daqueles que já existam no plano.
3. Inserir técnicas e estratégias de busca multiobjetivo, onde seria possível maximizar diversas características do jogo além da força de ataque dos recursos, na busca por metas com qualidade.
4. Construir uma interface para auxiliar um jogador humano dentro do jogo. Essa interface pode funcionar como um auxílio para jogadores de todos os níveis, onde esses poderiam consultar quais recursos produzir em um determinado intervalo de tempo ou qual a melhor ordem de execução das ações que eles vão executar para obter um menor tempo de produção.
5. Incrementar a abordagem com técnicas de IA que permitem a exploração de estratégias de ataque e defesa para que seja construída uma inteligência artificial jogadora completa. Essa seria capaz de planejar quais as melhores metas dentro do jogo além de atacar e se defender do inimigo em uma disputa direta.

Referências Bibliográficas

[Aarts e Korst 1989] Aarts, E. e Korst, J. (1989). *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimisation and Neural Computing.* John Wiley e Sons, Inc. New York, NY, USA 1989.

[Adams 2006] Adams, D. (2006). The State of the RTS. IGN PC.com. Disponível em: http://pc.ign.com/articles/700/700747p1.html - Último acesso em 03/07/2012.

[Anderson et al. 1998] Anderson, C. R., Smith, D. E., e Weld, D. S. (1998). Conditional Effects in Graphplan. In *Conference on Artificial Intelligence Planning Systems (AIPS)*, pp. 44–53.

[Andrew Trusty 2008] Andrew Trusty, Santiago Ontanón, A. R. (2008). Stochastic Plan Optimization in Real-Time Strategy Games. *In Artificial Intelligence and Interactive Digital Entertainment (AIIDE-2008), Stanford University, California.*

[Back 1996] Back, T. (1996). *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford Univ. Press.

[Bandyopadhyay et al. 2008] Bandyopadhyay, S., Sriparna, S., Ujjwal, M., e Kalyanmoy, D. (2008). A Simulated Annealing-Based Multiobjective Optimization Algorithm: AMOSA. *IEEE Transactions on Evolutionary Computation.*

[Blum e Furst 1997] Blum, A. L. e Furst, M. L. (1997). Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90(1-2):281–300.

[Bonet e Geffner 2011] Bonet, B. e Geffner, H. (2011). mGPT: A Probabilistic Planner Based on Heuristic Search. *Journal Of Artificial Intelligence Research, Volume 24, pages 933-944.*

[Bouleimen e Lecocq 2003] Bouleimen, K. e Lecocq, H. (2003). A new efficient simulated annealing algorithm for the RCPSP and its multiple mode version. *European Journal of Operational Research*, Volume 149,:pp. 268–281(14).

[Branquinho et al. 2011a] Branquinho, A., Lopes, C. R., e Naves, T. F. (2011a). Developing Strategies for Improving Planning and Scheduling of Actions in RTS Games. *23rd IEEE International Conference on Tools with Artificial Intelligence.*

[Branquinho et al. 2011b] Branquinho, A., Lopes, C. R., e Naves, T. F. (2011b). Using Search and Learning for Production of Resources in RTS Games. *The 2011 International Conference on Artificial Intelligence.*

[Brémaud 1999] Brémaud, P. (1999). *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues.* Springer.

[Buro 2004] Buro, M. (2004). Call for AI Research in RTS Games. *American Association for Artificial Intelligence (aaai)*.

[B.Wahlstrom. 1968] B.Wahlstrom., N. N. J. R. (1968). Application of Intelligent Automata to Reconnaissance. Technical report, Stanford Research Institute.

[Bwapi 2011] Bwapi (2011). BWAPI - An API for interacting with Starcraft : Broodwar. disponível em http://code.google.com/p/bwapi/ - Último acesso em 03/07/2012.

[Chan et al. 2008] Chan, H., Fern, A., Ray, S., Ventura, C., e Wilson, N. (2008). Extending Online Planning for Resource Production in Real-Time Strategy Games with Search. *Workshop on Planning in Games ICAPS*.

[Chan et al. 2007] Chan, H., Fern, A., Ray, S., Wilson, N., e Ventura, C. (2007). Online Planning for Resource Production in Real-Time Strategy Games. In *ICAPS*.

[Churchil e Buro 2011] Churchil, D. e Buro, M. (2011). Build Order Optimization in StarCraft. *AIIDE 2011: AI and Interactive Digital Entertainment Conference*.

[Do e S. 2003] Do, M. B. e S., K. (2003). SAPA: A scalable multi-objective metric temporal planner. *Journal of AI Research*, 20:63–75.

[Edelkamp e Hoffmann 2004] Edelkamp, S. e Hoffmann, J. (2004). PDDL2.2: The language for the Classical Part of the 4th International Planning Competition. Technical Report 195.

[Fayard 2005] Fayard, T. (2005). Using a Planner to Balance Real Time Strategy Video Game. *Workshop on Planning in Games, ICAPS 2007*.

[Fikes e Nilsson 1971] Fikes, R. e Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3/4):189–208.

[Fink 2003] Fink, E. (2003). *Changes of Problem Representation: Theory and Experiments*. Springer.

[Gerevini e Serina 2003] Gerevini, A.; Saetti, A. e Serina, I. (2003). Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research 20:239-230*.

[Hart et al. 1968] Hart, P. E., Nilsson, N. J., e Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107.

[Holland 1975] Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor: Univ. of Michigan Press, 1975.

[Hoos e Stützle 2012] Hoos, H. H. e Stützle, T. (2012). *Stochastic Local Search: Foundations and Applications*. Elsevier.

[Izquierdo 2000] Izquierdo, V. B. (2000). Uma Proposta de Especificação Formal e Fundamentação Teórica para Simulated Annealing. Master's thesis, Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação.

[Kautz e Selman 1992] Kautz, H. e Selman, B. (1992). Planning as Satisfiability. In *ECAI '92: Proceedings of the 10th European Conference on Artificial Intelligence*, pp. 359–363, Vienna, Austria, New York, NY, USA. John Wiley & Sons, Inc.

[Kautz e Selman 1996] Kautz, H. e Selman, B. (1996). Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Conference on Artificial Intelligence*, pp. 1194–1201. AAAI Press.

[Kautz e Walser 1999] Kautz, H. e Walser, J. P. (1999). State-space Planning by Integer Optimization. In *In Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pp. 526–533. AAAI Press.

[Korf 1985] Korf, R. E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109.

[Laarhoven e Aarts 1989] Laarhoven, P. J. M. e Aarts, E. H. L. (1989). *Simulated Annealing: Theory and Applications*. Springer.

[Lecheda 2004] Lecheda, E. M. (2004). Algoritmos Genéticos para Planejamento em Inteligência Artificial. Master's thesis, Universidade Federal do Paraná.

[Mcallester e Rosenblitt 1991] Mcallester, D. e Rosenblitt, D. (1991). Systematic Nonlinear Planning. In *In Proceedings of the Ninth National Conference on Artificial Intelligence*, pp. 634–639.

[Michael Bowman e Labiche 2010] Michael Bowman, L. C. B. e Labiche, Y. (2010). Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms. *IEEE Transactions on Software Engineering*, pp. 817–837.

[Minton et al. 1991] Minton, S., Bresina, J., e Drummond, M. (1991). Commitment Strategies in Planning: A Comparative Analysis. In *In Proceedings of IJCAI-91*, pp. 259–265.

[Minton et al. 1994] Minton, S., Bresina, J., e Drummond, M. (1994). Total-Order and Partial-Order Planning: A Comparative Analysis. *Journal of Artificial Intelligence Research*, 2:227–262.

[Mohan Sridharana 2010] Mohan Sridharana, Jeremy Wyattb, R. D. (2010). Planning to see: A hierarchical approach to planning visual actions on a robot using POMDPs. *Artificial Intelligence*.

[Newell e Simon 1995] Newell, A. e Simon, H. A. (1995). GPS, A Program That Simulates Human Thought. *Computers & thought*, pp. 279–293.

[Pearl 1984] Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Pednault 1989] Pednault, E. P. D. (1989). ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pp. 324–332, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Penberthy e Weld 1992] Penberthy, J. S. e Weld, D. S. (1992). UCPOP: A Sound, Complete, Partial Order Planner for ADL. pp. 103–114. Morgan Kaufmann.

[Ras e Skowron 2010] Ras, Z. e Skowron, A. (2010). A Heuristic for Domain Independent Planning and Its Use in an Enforced Hill-Climbing Algorithm. *10th International Symposium, Ismis '97*.

[Rintanen 2005] Rintanen, J. (2005). *Automated Planning: Algorithms and Complexity*. PhD thesis, University of Freiburg.

[Russell e Norvig 2003] Russell, S. J. e Norvig (2003). *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall.

[Sacerdoti 1975] Sacerdoti, E. D. (1975). The Nonlinear Nature of Plans. In *IJCAI*, pp. 206–214.

[Sussman 1973] Sussman, G. J. (1973). *A Computational Model of Skill Acquisition*. PhD thesis, MIT.

[Weld 1999] Weld, D. S. (1999). Recent Advances in AI Planning. *AI Magazine*, 20:93–123.

[Wilkins et al. 1995] Wilkins, D. E., Myers, K. L., e Lowrance, J. D. (1995). Planning and Reacting in Uncertain and Dynamic Environments. *Journal of Experimental and Theoretical AI*, vol 7:pp. 197–227.

[Wu 2009] Wu, J.-H. (2009). *Probabilistic planning via automatic feature induction and stochastic enforced hill-climbing*. PhD thesis, Purdue University West Lafayette, IN, USA 2009.