

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



**PLANEJAMENTO DE RECURSOS PARA JOGOS DE
ESTRATÉGIA EM TEMPO REAL**

AUGUSTO AFONSO BORGES BRANQUINHO

Uberlândia - Minas Gerais

2009

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



AUGUSTO AFONSO BORGES BRANQUINHO

PLANEJAMENTO DE RECURSOS PARA JOGOS DE ESTRATÉGIA EM TEMPO REAL

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

2009

Dados Internacionais de Catalogação na Publicação (CIP)

B821p Branquinho, Augusto Afonso Borges, 1984-
Planejamento de recursos para jogos de estratégia em tempo real /
Augusto Afonso Borges Branquinho. - 2009.
133 f. : il.

Orientador: Carlos Roberto Lopes.
Dissertação (mestrado) – Universidade Federal de Uberlândia, Pro-
grama de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Inteligência artificial - Teses. 2. Jogos estratégicos (Matemática) -
Teses. I. Lopes, Carlos Roberto. II. Universidade Federal de Uberlândia.
Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3:007.52

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 “**Planejamento de Recursos para Jogos de Estratégia em Tempo Real**” por **Augusto Afonso Borges Brancinho** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 09 de Novembro de 2009

Orientador:

Prof. Dr. Carlos Roberto Lopes
Universidade Federal de Uberlândia

Banca Examinadora:

Prof. Dr. Fernando Santos Osório
Universidade de São Paulo (USP) - São Carlos

Prof^a. Dr^a. Gina Maira Barbosa de Oliveira
Universidade Federal de Uberlândia

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

Data: Novembro de 2009

Autor: **Augusto Afonso Borges Branquinho**
Título: **Planejamento de Recursos para Jogos de Estratégia em Tempo Real**
Faculdade: **Faculdade de Computação**
Grau: **Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

Autor

O AUTOR RESERVA PARA SI QUALQUER OUTRO DIREITO DE PUBLICAÇÃO DESTE DOCUMENTO, NÃO PODENDO O MESMO SER IMPRESSO OU REPRODUZIDO, SEJA NA TOTALIDADE OU EM PARTES, SEM A PERMISSÃO ESCRITA DO AUTOR.

Dedicatória

Dedico este trabalho a Deus, pois sempre colocou pessoas especiais no meu caminho.

Agradecimentos

Agradeço ...

Aos meus amados pais Jair Carolino Branquinho e Terezinha Maria Borges Carolino pela dedicação, amor incondicional, confiança, apoio e carinho em todos os momentos. Obrigado por me ajudarem a ser uma pessoa feliz, honesta, verdadeira e humilde. Com muito amor eu agradeço a vocês por tudo que são na minha vida.

A Mariel Caroline Pereira Queiroz por me mostrar que nem tudo são algoritmos e como é importante ter pessoas especiais ao nosso lado. Obrigado por me ajudar a encontrar o meu melhor e ter mudado a minha vida.

Ao meu professor e orientador Carlos Roberto Lopes que sempre acreditou no meu trabalho, me ajudou e confiou em mim. O senhor me mostrou como é importante o trabalho de pesquisa e educação, muito obrigado por tudo.

Aos meus tios Edivino José da Costa e Maria de Fátima Costa Peres que sempre se preocuparam comigo e estiveram sempre ao meu lado. Obrigado por serem a base de nossa família.

Ao meu amigo e professor Bruno Nepomuceno Luiz que sempre acreditou no meu trabalho, por ter me ajudado na graduação e apoiado durante o início do mestrado. Sempre lembrarei de como você foi importante em vários momentos da minha vida.

As professoras Márcia Aparecida Fernandes e Gina Maira Barbosa de Oliveira que sempre me ajudaram e demonstraram verdadeira paixão pela educação e pesquisa. Professoras como vocês são os meus exemplos de dedicação e espelho para a minha jornada após o mestrado.

Ao professor Ary dos Santos Rocha Júnior que me ajudou no processo de seleção do mestrado.

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

Ao meu tio Jairo Gomes Carolino que no início do mestrado sempre se dispôs em me ajudar.

A Maria de Lourdes Luiz por ter me ajudado durante muitos momentos de minha vida.

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

Aos amigos Marcos Luiz de Paula Bueno, José dos Reis Mota, Lucas Terra Andrade e Bruno Alves Landim, que me ajudaram nos momentos que precisei.

Ao amigo e colega de graduação Cássio Geraldo Freitas.

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.

*“Ser Feliz não é ter um céu sem tempestades, caminhos sem
acidentes, trabalhos sem fadigas, relacionamentos sem decepções.
Ser Feliz é encontrar força no perdão, esperanças nas batalhas,
segurança no palco do medo, amor nos desencontros.*

*Ser Feliz não é apenas valorizar o sorriso, mas refletir sobre
a tristeza. Não é apenas comemorar o sucesso, mas aprender
lições nos fracassos. Não é apenas ter júbilo nos aplausos,
mas encontrar alegria no anonimato.*

*Ser Feliz é reconhecer que vale a pena viver a vida, apesar
de todos os desafios, incompreensões e períodos de crise.
Ser Feliz não é uma fatalidade do destino, mas uma conquista
de quem sabe viajar para dentro do seu próprio ser.*

*Ser Feliz é deixar de ser vítima dos problemas e se tornar
autor da própria história. É atravessar desertos fora de si,
mas ser capaz de encontrar um oásis no recôndito da sua alma.
É agradecer a Deus a cada manhã pelo milagre da vida.*

*Ser Feliz é não ter medo dos próprios sentimentos.
É saber falar de si mesmo. É ter coragem para ouvir um não.
É ter segurança para receber uma crítica, mesmo que injusta.
É curtir os pais e ter momentos poéticos com os amigos,
mesmo que eles nos magoem.*

*Ser Feliz não é ter uma vida perfeita. Mas usar as lágrimas
para irrigar a tolerância. Usar as pedras para refinar a
paciência. Usar as falhas para esculpir a serenidade. Usar
a dor para lapidar o prazer. Usar os obstáculos para abrir
as janelas da inteligência.”*

(Augusto Cury)

Resumo

Geralmente, um jogo de estratégia em tempo real é caracterizado por duas fases. Inicialmente, é necessária a coleta e a produção de recursos. O próximo passo está relacionado a batalha, levando em conta os recursos coletados. O estágio de produção de recursos é um fator chave para vencer o jogo.

Sendo assim, neste estudo é proposto um mecanismo de produção de recursos baseado em planejamento, suportado por inteligência artificial usando análises de meios-fim e escalonamento. O objetivo consiste em produzir planos para jogos de estratégia em tempo real. Ênfase é dada para o escalonamento usando um algoritmo de busca e aprendizado em tempo real.

São apresentados os resultados do sistema proposto comparando com trabalhos relacionados. Os resultados demonstraram que o sistema proposto apresentou uma melhor performance que as abordagens correlatas.

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

Abstract

Generally, a real-time strategy game is characterized by two stages. Initially, it is necessary to collect and produce resources. The next step is related to battles, taking into account the resources that were collected. The resources production stage is a key factor for winning the game.

In this study the authors propose a mechanism for producing resources based on planning, supported by artificial intelligence using means-end analysis and scheduling. Emphasis is given to scheduling that uses an algorithm of real-time search and learning.

The results show that the proposed system presents a better performance compared to related approaches.

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

Sumário

Lista de Figuras	xix
Lista de Tabelas	xxiii
Lista de Algoritmos	xxv
Lista de Abreviaturas e Siglas	xxvii
1 Introdução	29
1.1 Contribuição	30
1.2 Organização da Dissertação	30
2 Fundamentos Teóricos e Trabalhos Correlatos	31
2.1 Jogos RTS	31
2.2 Algoritmos de Busca	36
2.2.1 A*	38
2.2.2 LRTA*	39
2.2.3 SLA*	41
2.3 Planejamento	48
2.3.1 Conceitos Relacionados	48
2.3.2 Planejamento com Busca	52
2.3.3 Planejador STRIPS	55
2.3.4 Planejamento de Ordem Parcial	59
2.4 Planejamento de Recursos para Jogos RTS	63
2.4.1 Planejador Seqüencial	65
2.4.2 Escalonamento	69
2.4.3 Produção de Recursos Adicionais	73
3 Arquitetura de Planejamento	77
4 Estabelecimento de Prioridade para Atividades	85

5	Escalonamento	93
5.1	SLA* Aplicado no Escalonamento de Recursos em Jogos RTS	94
5.1.1	Exemplo	97
5.1.2	Estratégias de Eficiência	102
5.1.3	Detalhes de Implementação	106
6	Experimentos e Discussão dos Resultados	111
7	Conclusão e Trabalhos Futuros	121
	Referências Bibliográficas	123
A	Somatórios	127
B	Prova da Solução Ótima do SLA*	129

Lista de Figuras

2.1	<i>Warcraft II</i> e <i>Warcraft III</i>	32
2.2	<i>Starcraft I</i> e <i>Starcraft II</i>	32
2.3	<i>Townhalls</i> do jogo <i>Wargus</i>	33
2.4	<i>Peasants</i> do jogo <i>Wargus</i>	34
2.5	<i>Gold Mine</i> e <i>Forest</i> do jogo <i>Wargus</i>	34
2.6	<i>Barracks</i> e <i>Farms</i> do jogo <i>Wargus</i>	34
2.7	<i>Footmans</i> do jogo <i>Wargus</i>	34
2.8	Domínio simplificado do jogo <i>Wargus</i> [Chan et al. 2007a].	36
2.9	Exemplo de aplicação do LRTA*.	40
2.10	Exemplo de um Projeto de Atividades com apenas um recurso constante.	43
2.11	Resultado do escalonamento do projeto de atividades.	46
2.12	Primeiros estados da busca para o exemplo de projeto de atividades.	46
2.13	Resultado completo da busca com todos os estados.	47
2.14	Problema “ <i>Mundo dos Blocos</i> ” representado pela linguagem STRIPS.	50
2.15	Primeiro passo de uma busca progressiva para o problema da Figura 2.14.	53
2.16	Primeiro passo da busca regressiva para o problema da Figura 2.14.	54
2.17	Problema “Mundo de Shakey”.	56
2.18	Problema “Mover a Caixa”.	57
2.19	Operadores do problema “Mover a Caixa”.	58
2.20	Estado Inicial e Meta do problema “Mover a Caixa”.	58
2.21	Resultado do planejamento do problema “Mover a Caixa”.	59
2.22	Comparação entre o planejamento de ordem parcial e total.	60
2.23	Descrição do problema “ <i>Pneu Furado</i> ”.	62
2.24	Planejamento de ordem parcial para o problema da Figura 2.23.	63
2.25	Primeiro passo do problema de planejamento seqüencial.	66
2.26	Segundo passo do problema de planejamento seqüencial.	67
2.27	Terceiro passo do problema de planejamento seqüencial.	67
2.28	Quarto passo do problema de planejamento seqüencial.	68
2.29	Quinto passo do problema de planejamento seqüencial.	68
2.30	Plano seqüencial obtido no exemplo tratado.	69

2.31	Tentativa de escalonar a ação <i>collect-gold</i> no tempo 0.	70
2.32	A ação <i>collect-gold</i> é escalonada no tempo 0.	71
2.33	Tentativa de escalonar a ação <i>collect-wood</i> no tempo 510.	71
2.34	Tentativa de escalonar a ação <i>collect-wood</i> no tempo 0.	71
2.35	A ação <i>collect-wood</i> é escalonada no tempo 0.	72
2.36	Tentativa de escalonar a ação <i>build-supply</i> no tempo 1570.	72
2.37	Tentativa de escalonar a ação <i>build-supply</i> no tempo 510.	72
2.38	A ação <i>build-supply</i> é escalonada no tempo 1570.	73
2.39	Resultado do escalonamento do plano da Figura 2.30.	73
2.40	Produção de recursos adicionais através de metas intermediárias.	75
3.1	Planos usados como motivação da arquitetura de planejamento.	78
3.2	Exemplo de execução do Algoritmo 7 em um problema de planejamento. . .	82
3.3	Exemplo de execução do Algoritmo 7 em um problema de planejamento (continuação).	83
3.4	Plano de Ordem Parcial criado com o Algoritmo 7 para o exemplo tratado. .	84
4.1	Plano usado como exemplo no estabelecimento de prioridade.	86
4.2	Subgrafos da Figura 4.1.	86
4.3	Grafo usado na análise do melhor caso do Algoritmo 11.	89
4.4	Grafo usado na análise do pior caso do Algoritmo 11.	90
4.5	Número de vértices e arestas visitados em um grafo com 6 vértices.	91
5.1	Plano de ordem parcial usado como exemplo no escalonamento utilizando o SLA*.	94
5.2	Exemplo de estados que são criados em um passo do escalonamento.	97
5.3	Primeiro passo da busca do SLA* para o plano da Figura 5.1.	99
5.4	Segundo passo da busca do SLA* para o plano da Figura 5.1.	100
5.5	Planos obtidos ao usar o planejamento de [Chan et al. 2007b] (A), <i>MeaPop</i> (B) e SLA* (C).	101
5.6	Estados que são criados ao considerar a técnica “Máximo de Ações”.	103
5.7	Estados criados ao usar a técnica “Menor Tempo”.	103
5.8	Exemplo de um plano de ordem parcial usado na estratégia “Uso de Prio- ridade”.	104
5.9	Primeiro e segundo passos da busca ao usar a técnica “Uso de Prioridade”. .	105
5.10	Estrutura de representação do <i>Hashtable</i>	108
6.1	Resumo dos procedimentos usados nos experimentos.	111
6.2	Gráficos do tempo e <i>makespan</i> para o Experimento 2.	113
6.3	Gráficos do tempo e <i>makespan</i> para o Experimento 3.	115
6.4	Gráficos do tempo e <i>makespan</i> para o Experimento 4.	115

6.5	Gráficos do tempo e <i>makespan</i> para o Experimento 6.	119
-----	---	-----

Lista de Tabelas

2.1	Resultado completo da busca para o exemplo de projeto de atividades. . .	45
5.1	Resultado resumido do processo de busca para o exemplo utilizado no SLA *.100	
6.1	Resultados do Experimento 1.	112
6.2	Resultados do experimento 2.	113
6.3	Resultados do experimento 2 (continuação).	114
6.4	Resultados do experimento 3.	114
6.5	Resultados do experimento 3 (continuação).	114
6.6	Resultados do experimento 4.	116
6.7	Resultados do experimento 5.	116
6.8	Resultados do experimento 5 (continuação).	117
6.9	Resultados do experimento 6.	118
6.10	Resultados do experimento 7.	119
6.11	Resultados do experimento 8.	120

List of Algorithms

1	$A^*(s_{inicio}, s_{meta})$	38
2	$LRTA^*(s_{inicio}, s_{meta})$	39
3	$SLA^*(s_{inicio}, s_{meta})$	41
4	$MEA(S, G)$	65
5	$Schedule(Plano)$	69
6	$OnlinePlanner()$	74
7	$MeaPop(S_{inicio}, G_{meta})$	79
8	$PlanejamentoOrdemParcial(S, G)$	79
9	$CriarAtividade(s_i, r_i, g_i, S, G)$	80
10	$Prioridade(G)$	87
11	$PrioridadeVisita(G, u, p)$	87
12	$EstadosSucessores(Recursos, Atividades)$	95
13	$AdicionarEstados(A_j, S, Recursos, Atividades)$	96
14	$MaximoAcoes(A_j, S, Recursos, Atividades)$	102

Lista de Abreviaturas e Siglas

ADL	Action Description Language
GPS	General Problem Solver
IA	Inteligência Artificial
LRTA*	Learning Real-time A*
MEA	Means-end Analysis
ORTS	Online Real-time Strategy
PDDL	Planning Domain Definition Language
POP	Partial Order Planning
PSRC	Project Scheduling with Resource Constraints
RTS	Real-time Strategy
SGP	Sensory Graphplan
SLA*	Search and Learn A*
SLA*T	Search and Learn A* Threshold
STRIPS	STanford Research Institute Problem Solver
UFU	Universidade Federal de Uberlândia

Capítulo 1

Introdução

Os jogos de estratégia em tempo real (ou jogos RTS, um acrônimo para *Real-time Strategy*) são uma categoria muito apreciada de jogos de entretenimento. *Warcraft* e *Starcraft* são exemplos de jogos muito conhecidos que se enquadram nesta categoria. Normalmente são gêneros caracterizados por serem de guerra e o objetivo é alcançar um domínio militar ou territorial sobre o adversário. O papel do adversário pode ser desempenhado por outro jogador ou pelo computador. As decisões são destinadas para produção de recursos e táticas de batalha, com o objetivo de superar os oponentes. Para isso, é preciso um conjunto de ações, como coleta de recursos, construção de edifícios, treinamento de unidades, desenvolvimento de tecnologia e combate com o exército inimigo. Para todas as ações, é necessário decidir o melhor momento e a ordem que serão executadas. Além disso, o mapa inicial é desconhecido e com o decorrer do jogo pode sofrer alterações.

Normalmente, estes jogos envolvem um período inicial em que o jogador deve rapidamente construir sua economia através da produção de recursos, seguida pela campanha militar em que esses recursos são explorados para as ações ofensivas e defensivas. Assim, vencer a corrida na produção de recursos é muitas vezes um fator chave para o sucesso.

Os recursos dos jogos RTS incluem todas as matérias-primas, construções, civilização e unidades militares. Para construção de armamentos e edifícios, além de necessitar de matéria-prima, é preciso determinar a ordem e a quantidade que serão produzidos. Para essa tarefa [Chan et al. 2007b] utiliza de análises de meios-fim (*Means-end Analysis* - MEA) e escalonamento.

A partir destas considerações, este trabalho pretende melhorar os resultados de [Chan et al. 2007b] obtendo um menor *makespan* (tempo total do plano gerado). Ênfase é destinada ao processo de produção de recursos, sendo proposto um planejamento de ordem parcial. Juntamente com este planejamento de ordem parcial é realizado um processo que estabelece prioridade para as ações serem executadas. A prioridade em executar ações permite que durante o escalonamento os recursos produzidos sejam aproveitados para a execução das próximas ações. Este plano de ordem parcial é usado como entrada para os algoritmos de escalonamento.

Para realizar a tarefa de escalonamento são usados algoritmos de busca em tempo real. Tais algoritmos são usados de modo que as soluções sejam melhoradas com o decorrer do jogo. Além disso, algumas estratégias são usadas para tornar este procedimento mais eficiente.

Em síntese, o objetivo do trabalho é desenvolver um mecanismo de seleção de ações que consiga satisfazer rapidamente a uma meta de produção de recursos com o menor tempo possível. Este trabalho pode ser útil, por exemplo, como uma opção de interface para jogadores humanos, em que o jogador necessita especificar somente as metas que deseja atingir. Além disto, constitui-se numa área interessante de planejamento pois apresenta uma série de aspectos desafiadores tais como ações caracterizadas por tempo de duração e efeitos numéricos, concorrência de ações e sua execução num ambiente altamente dinâmico.

1.1 Contribuição

As principais contribuições deste trabalho são:

1. Desenvolvimento do Algoritmo **MeaPop**. Este algoritmo cria planos de ordem parcial. A partir destes planos é possível aplicar algoritmos de busca em tempo real que realizam o escalonamento das ações.
2. Proposta do Algoritmo **Prioridade** que define quais ações são mais importantes em um plano de ordem parcial.
3. Desenvolvimento de estratégias para aumentar a eficiência do escalonamento de ações.
4. Proposta do uso de estruturas de dados *hashtable* e *heap* para a implementação do algoritmo de escalonamento. Tal procedimento torna a busca mais rápida e diminui o número de estados que são criados.

1.2 Organização da Dissertação

A dissertação está estruturada nos seguintes capítulos especificados a seguir. No Capítulo 2 são apresentados os fundamentos teóricos e trabalhos correlatos. O Capítulo 3 descreve o algoritmo de planejamento de ordem parcial. A tarefa de estabelecer prioridade para atividades é especificada no Capítulo 4. No Capítulo 5 é apresentado o procedimento de escalonamento. Os resultados são apresentados e discutidos no Capítulo 6. Ao final, no Capítulo 7, são apresentadas as conclusões e perspectivas para trabalhos futuros.

Capítulo 2

Fundamentos Teóricos e Trabalhos Correlatos

Neste capítulo são discutidos os fundamentos teóricos e trabalhos correlatos objetivando propiciar uma melhor compreensão do trabalho desenvolvido.

Este capítulo está estruturado da seguinte forma. Na Seção 2.1 são descritos os conceitos de jogos RTS. A seguir, na Seção 2.2 são apresentados conceitos e algoritmos de busca. A Seção 2.3 descreve os conceitos de planejamento. Finalmente, na Seção 2.4 são descritas abordagens envolvendo uso de planejamento para satisfazer metas de obtenção/produção de recursos em jogos RTS.

2.1 Jogos RTS

Os jogos de estratégia em tempo real (ou jogos RTS, um acrônimo para *Real-time Strategy*) são jogos de estratégia militar caracterizados por um ambiente em que as ações e eventos ocorrem em tempo real [Adams 2006].

Um dos primeiros e mais influentes jogos RTS foi o *Dune II: The Building of a Dynasty* [Adams 2006]. Neste jogo existe a coleta de recursos, construção de bases, produção de unidades e batalhas travadas até alcançar a vitória. A partir deste título foram desenvolvidos jogos famosos como *Warcraft*, *Starcraft*¹ e *Command & Conquer*². Existem tantos jogos RTS que é difícil destacar os mais importantes da história. Nas Figuras 2.1 e 2.2 são exibidas, respectivamente, imagens dos jogos *Warcraft* e *Starcraft*. Neste trabalho ênfase será dada ao jogo *Wargus* que será detalhado mais adiante.

O objetivo dos jogos RTS é alcançar um domínio militar ou territorial sobre o adversário. O papel do adversário pode ser desempenhado por outro jogador ou pelo computador. As decisões são destinadas para produção de recursos e táticas de batalha, com o

¹ *Warcraft* e *Starcraft* são produzidos pela *Blizzard Entertainment*: <http://www.blizzard.com/>

² *Command & Conquer* foi criado pela empresa *Westwood Studios*. Atualmente é desenvolvido pela *Electronic Arts*: <http://www.ea.com/>

Figura 2.1: *Warcraft II* e *Warcraft III*.Figura 2.2: *Starcraft I* e *Starcraft II*.

objetivo de superar os oponentes. Para isso, é preciso de um conjunto de ações, como coleta de recursos, construção de edifícios, treinamento de unidades, desenvolvimento de tecnologia e combate com o exército inimigo. Para todas as ações, é necessário decidir o melhor momento e a ordem que serão executadas. Além disso, o mapa inicial é desconhecido e com o decorrer do jogo pode sofrer alterações.

Normalmente, estes jogos envolvem um período inicial em que o jogador deve rapidamente construir sua economia através da produção de recursos, seguido pela campanha militar em que esses recursos são explorados para as ações ofensivas e defensivas. Este período inicial depende do jogo e da estratégia do jogador. Assim, vencer a corrida na produção de recursos é muitas vezes um fator chave para o sucesso.

Os recursos dos jogos RTS incluem todas as matérias-primas, construções, civilizações e unidades militares. Para construção de armamentos e edifícios, além de necessitar de matéria-prima, é preciso determinar a ordem e a quantidade que serão produzidos. O processo de planejamento consiste em determinar a melhor ordem que as ações serão

executadas.

Freqüentemente é usado o jogo *Wargus*³ ou o ambiente ORTS⁴ para testar algoritmos de planejamento, estratégias e táticas de batalha. *Wargus* é um módulo do jogo *Warcraft II* com código fonte aberto. ORTS é um ambiente para estudos de problemas de IA, como planejamento, escalonamento, busca de caminho e táticas de formação.

Wargus possui diversos recursos e ações. Os recursos materiais são destinados a construção de edifícios e desenvolvimento da civilização. Estes recursos são destinados ao combate. É evidente que a estratégia e a tática de batalha são fatores importantes para a vitória, contudo, quanto antes os recursos estejam disponíveis maiores são as chances de sucesso.

No *Wargus* existem duas raças: humanos e *orcs*. Cada raça possui diferentes guerreiros e construções. O combate é realizado entre humanos contra humanos, *orcs* contra *orcs* e/ou *orcs* contra humanos. A principal estrutura é a *townhall* (prefeitura). A *townhall* é responsável por criar *peasant* (peão) e servir como local de depósito de *gold* (ouro) e *wood* (madeira). Com o decorrer do jogo são realizadas atualizações de tecnologia. Cada atualização permite melhorar a *townhall*. Atualizar a *townhall* possibilita criar novas estruturas e incrementar a coleta de *gold*. O incremento de *gold* ocorre do seguinte modo, com a *townhall* mais simples cada coleta corresponde a 100 *gold*. Após a atualização da *townhall* cada coleta de *gold* tem como resultado 110 *gold*. Uma posterior atualização da *townhall* altera a coleta de *gold* para 125.

A Figura 2.3 exibe os tipos de *townhalls* do *Wargus*. As *townhalls* (A), (B) e (C), são os três tipos de *townhalls* dos humanos. (D), (E) e (F), são as *townhalls* dos *orcs*.

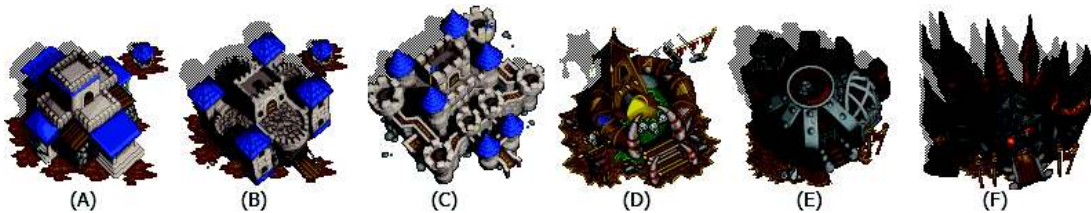


Figura 2.3: *Townhalls* do jogo *Wargus*.

O *peasant* é responsável por realizar as tarefas de coleta de recursos e construção de estruturas. Na Figura 2.4 (A) e (B), são exibidos, respectivamente, os *peasants* dos humanos e *orcs*. Para criar um *peasant* é necessário consumir *gold* e *supply* (suprimento). Além disso, é preciso que uma *townhall* seja usada, ficando ocupada enquanto o *peasant* estiver sendo produzido. Em outras palavras, a *townhall* ocupada não pode ser utilizada por outra ação que possa a ocupar enquanto não estiver livre.

O recurso *gold* é obtido através de sua coleta pelo *peasant* na *gold mine* (mina de ouro). Cada *gold mine* possui uma quantidade específica de *gold*. O *gold* extraído é

³Para maiores informações: <http://wargus.sourceforge.net/>

⁴Para maiores informações: <http://web.cs.ualberta.ca/~mburo/orts/>

armazenado na *townhall*. A Figura 2.5(A) mostra o desenho da *gold mine*.

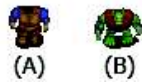


Figura 2.4: *Peasants* do jogo *Wargus*.

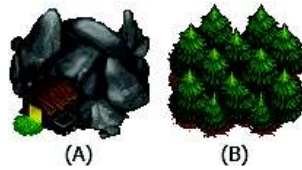


Figura 2.5: *Gold Mine* e *Forest* do jogo *Wargus*.

Outro recurso coletado pelo *peasant* é a *wood*. A coleta de *wood* é realizada na *forest* (floresta). Com o decorrer do jogo a *forest* vai acabando. A *wood* extraída é armazenada na *townhall*. A Figura 2.5 (B) mostra o desenho da *forest*.

O recurso *supply* é representado como sendo uma estrutura chamada *farm* (fazenda). Cada *farm* corresponde a 4 *supply*, ou seja, no momento que uma *farm* é criada é disponibilizado 4 *supply*. A Figura 2.6 (B) e (D), exibe respectivamente, as *farms* dos humanos e *orcs*.



Figura 2.6: *Barracks* e *Farms* do jogo *Wargus*.

Outro recurso é o *footman* (soldado). Os *footmans* são responsáveis por destruir o(s) oponente(s). Existem diversos tipos de *footmans*, cada um deles possuem suas propriedades específicas.

Para criar um *footman* é preciso consumir *gold*, *wood* e/ou *supply*. O que é consumido pode variar de acordo com o *footman*. Além destes recursos é necessário ocupar uma *barracks* (barraca) durante o tempo que o *footman* está sendo criado. A Figura 2.6 (A) e (C) mostra as *barracks* do *Wargus*. Alguns dos *footmans* do *Wargus* são mostrados na Figura 2.7.



Figura 2.7: *Footmans* do jogo *Wargus*.

O *Wargus* possui outros recursos e estruturas⁵. A coleta de recursos e construção de estruturas são representadas por ações. Cada ação é composta por um conjunto de

⁵Para maiores detalhes do jogo *Warcraft II*: http://www.wowwiki.com/Warcraft_II

pré-condições e efeitos. Na Figura 2.8 é descrito um domínio simplificado de ações do *Wargus*, sendo este domínio o mesmo utilizado por [Chan et al. 2007a]. As ações estão associadas há quatro tipos de recursos: *borrow*, *consume*, *require* e *produce*.

A ação associada ao recurso *Require* é uma ação que requer certa quantidade de um recurso que precisa estar presente durante toda a execução da ação. Por exemplo, a ação *collect-gold* requer a presença de uma *townhall*. Neste caso, a *townhall* pode ser utilizada para a coleta simultânea de *gold* por outras ações, já que a *townhall* não está “ocupada” pelas ações *collect-gold*. Assim, o tipo *require* permite compartilhar recursos mas exige que esteja presente durante toda a execução da ação.

A ação associada ao recurso *Borrow* é uma ação que necessita de uma determinada quantidade de recursos. Exige que o montante de recursos seja “ocupado/trancado” durante toda a execução da ação, de modo que nenhuma outra ação pode tomar emprestado esses recursos durante sua execução. Após a conclusão da ação o montante de recursos é liberado para o uso por outras ações. Por exemplo, a ação *collect-gold* ocupa um *peasant*. Durante a execução da ação *collect-gold*, o *peasant* que foi ocupado não pode ser emprestado para qualquer outra ação. Depois da ação *collect-gold* finalizar, o *peasant* se torna novamente disponível e pode ser usado por outras ações. Portanto, para permitir execuções simultâneas da ação *collect-gold* é necessário múltiplos *peasants*.

A ação associada ao recurso *Consume* é uma ação que consome uma certa quantidade de um recurso no início da sua execução. Este montante de recurso é deduzido do estado do jogo. O estado do jogo deve obedecer a restrição de recursos, de modo que a quantidade de recursos é sempre positiva. Essa condição obriga a ação a possuir aquele montante de recursos no estado de início da ação. Por exemplo, a ação *build-barracks* consome 700 unidades de *gold* e 450 unidades de *wood*. Sendo assim, no momento que tal ação for executada é necessário que o estado do jogo tenha pelo menos 700 *gold* e 450 *wood*.

A ação associada ao recurso *Produce* é uma ação que produz uma certa quantidade de um recurso. Este montante de recurso é adicionado ao estado do jogo após o término da ação.

As ações de produção de recursos possuem um tempo de duração, um conjunto de pré-condições e efeitos. O tempo de duração no *Wargus* é medido em ciclos. Um ciclo corresponde a uma fração de tempo. Por exemplo, 20 ciclos podem corresponder a 1 segundo. Cada 0,05 segundos é igual a um ciclo. A cada ciclo é possível executar um processamento, ou seja, uma ação de planejamento.

Na especificação das ações têm-se que as pré-condições referem-se a recursos necessários para a execução da ação, os quais estão associados aos seguintes tipos: *Require*, *Borrow* e *Consume*. Já os efeitos de uma ação especificam os recursos *Produce*. Tais ações encontram-se presentes no *Warcraft II* e *Wargus*. Por exemplo, para executar a ação *collect-gold* é requerido que exista 1 *townhall* e seja ocupado 1 *peasant*. Após a duração de 510, a ação finaliza e produz 100 *gold*.

```
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 2.8: Domínio simplificado do jogo *Wargus* [Chan et al. 2007a].

O domínio e a linguagem de representação são especificados em função do jogo. É importante que a especificação do domínio possua as informações importantes para o objetivo proposto.

2.2 Algoritmos de Busca

Algoritmos de busca são fundamentais para a solução de muitos problemas de IA. A solução de um problema de busca é composta por uma seqüência de estados. Cada estado representa parte de uma solução. Estes estados são divididos em: estado inicial, estados intermediários e estado meta. Para o conjunto de estados dado como solução de um problema, a partir do estado inicial, os estados intermediários são percorridos até alcançar o estado meta. Entre cada estado está associado um custo/distância. Normalmente, a melhor solução consiste no somatório máximo ou mínimo deste parâmetro. Os algoritmos são divididos em dois grupos: *offline* e *online* [Furcy 2004].

Os algoritmos de busca *offline* dispõem de tempo para encontrar a melhor solução,

sendo que nenhuma ação é executada antes que a solução seja encontrada. Uma ação corresponde a mudança entre estados. Para melhorar a convergência destes procedimentos estratégias são utilizadas envolvendo o uso de heurísticas. A heurística é uma regra, simplificação ou aproximação, que reduz ou limita a busca por soluções em domínios difíceis e/ou pouco compreendidos. Uma heurística pode ser aplicada para encontrar soluções ótimas ou aproximadas.

Os algoritmos de busca *online* intercalam escolhas e execuções de ações. Geralmente a escolha da melhor ação é baseada no valor de heurística, tendo como base informações locais. Neste tipo de busca não é possível achar a melhor solução em apenas um passo de busca, pois existe uma limitação de tempo para as ações serem executadas. Segundo [Russell e Norvig 2003], “O termo *online* é de uso comum em ciência da computação para fazer referência a algoritmos que devem processar dados de entrada à medida que eles são recebidos, em vez de esperar que o conjunto de dados de entrada inteiro se torne disponível.”.

Neste trabalho o problema de busca é definido formalmente pela Definição 2.1.

Definição 2.1. S denota um conjunto finito de estados. $s_{início} \in S$ é o estado inicial. $s_{meta} \in S$ é o estado meta⁶. Um conjunto de estados (solução) normalmente é dito como sendo um caminho $(s_1, s_2, \dots, s_{n-1}, s_n)$, ou seja, uma seqüência de estados, tal que $s_1 = s_{início}$, $s_n = s_{meta}$ e n = número de estados. $succ(s) \subseteq S$ denota o conjunto de sucessores do estado s (os sucessores de s são aqueles estados que podem ser alcançados diretamente a partir de s). $c(s, s') > 0$ denota o custo de mover do estado s para $s' \in succ(s)$. Os estados em S são associados as funções $h(s)$, $g(s)$ e $f(s)$. $h(s)$ é uma estimativa da distância entre s e a meta s_{meta} . $g(s)$ corresponde a distância entre s e o estado inicial $s_{início}$. Finalmente, temos que $f(s) = h(s) + g(s)$, ou seja, $f(s)$ é igual a soma de $h(s)$ mais $g(s)$. O valor ótimo de $h(s)$, $g(s)$ e $f(s)$, é definido respectivamente por $h^*(s)$, $g^*(s)$ e $f^*(s)$. Uma heurística h é admissível se $\forall s \in S, h(s) \leq h^*(s)$ e consistente se $\forall s \in S, h(s_{meta}) = 0$ e $0 \leq h(s) \leq c(s, s') + h(s')$ para todo estado onde $s \neq s_{meta}$ e $s' \in succ(s)$ [Furcy e Koenig 2000].

Outras conceitos são considerados dependendo do método e do problema tratado. Entre os algoritmos de busca *offline* temos o A^* [Hart et al. 1968], IDA^* [Korf 1985], Dijkstra, Busca em Largura e Busca em Profundidade [Cormen et al. 2001]. Entre os algoritmos de busca *online* temos o RTA^* , $LRTA^*$ [Korf 1990], FALCONS [Furcy 2004], SLA^* [Zamani e Shue 1998], SLA^*T [Zamani e Shue 2003] e $LRTA^*-LS$ [Hernández e Meseguer 2007].

Nas Seções 2.2.1 e 2.2.2 são descritos, respectivamente, os algoritmos A^* e $LRTA^*$. Estes algoritmos são muito importantes, pois são a base para diversos métodos. Na Seção

⁶Como simplificação assumimos que existe apenas uma meta. Em alguns problemas o número de metas é maior que um.

2.2.3 é apresentado o método SLA*. Ênfase é dada a tais algoritmos pois os mesmos são importantes para o trabalho que foi desenvolvido.

2.2.1 A*

O algoritmo A* (pronunciado “A-estrela”) é um método heurístico que combina os procedimentos *Best-first Search* [Pearl 1984, Russell e Norvig 2003] e Dijkstra [Cormen et al. 2001] para a busca de caminhos. Este algoritmo foi primeiramente descrito por [Hart et al. 1968], sendo chamado inicialmente de algoritmo A. Quando usado com uma heurística apropriada (admissível) atinge uma solução ótima, passando a se chamar A*. Para achar uma solução consideramos o estado inicial $s_{início}$ e a partir dele um conjunto de estados são percorridos até se atingir uma meta s_{meta} . No Algoritmo 1 é apresentado o A*.

Algorithm 1 $A^*(s_{início}, s_{meta})$

1. Adicione o estado inicial $s_{início}$ à lista *aberto*.
 2. Repetir o seguinte:
 - (a) Buscar o estado que tenha o menor valor de f na lista *aberto*. Referimos este estado como sendo o estado corrente $s_{corrente}$.
 - (b) Mova $s_{corrente}$ para a lista *fechado*.
 - (c) Para cada estado s' adjacente ao estado corrente, $s' \in succ(s_{corrente})$, faça:
 - i. Se s' estiver na lista *fechado*, ignore. Caso contrário:
 - A. Se s' não estiver na lista *aberto*, acrescente s' à lista *aberto*. Faça $s_{corrente}$ o pai de s' . Grave os custos f , g e h , do estado s' .
 - B. Se s' já estiver na lista *aberto*, confere para ver se este caminho para s' é melhor, usando o custo g como medida. Um valor g mais baixo mostra que este é um caminho melhor. Nesse caso, mude o estado pai de s' para o estado $s_{corrente}$, e recalcule os valores g e f de s' .
 - (d) Parar quando:
 - i. Acrescentar s_{meta} à lista *fechado*, ou
 - ii. Não achar s_{meta} e a lista *aberto* está vazia. Neste caso, não há nenhum caminho/solução.
 3. Caminhando para trás a partir do estado meta s_{meta} , vá a cada estado até seu estado pai até alcançar o estado inicial $s_{início}$. Este caminho é a solução.
-

O A* utiliza as funções h , g e f , como descrito na Definição 2.1, para determinar a ordem em que os estados (também chamado de nós) são computados. Para que uma solução seja ótima, a função heurística $h(s)$ tem de ser admissível. Além destas funções, são usadas duas listas, uma chamada “Lista Aberta” (*aberto*) e a outra “Lista Fechada” (*fechado*). A lista *aberto* é responsável por guardar os estados que ainda não foram

computados, normalmente armazenados utilizando uma fila de prioridade [Cormen et al. 2001]. A lista *fechado* guarda os estados que podem fazer parte da solução.

Se existir uma solução, o caminho entre $s_{início}$ e s_{meta} é computado do seguinte modo: a partir de s_{meta} os estados pai são determinados sucessivamente até alcançar $s_{início}$. A solução consiste no caminho percorrido da meta até o início através dos estados pai.

2.2.2 LRTA*

O algoritmo de busca em tempo real mais popular é o *Learning Real-time A** (LRTA*) [Korf 1990]. Este método possui *Lookahead*⁷ de um e para cada estado s é associado um valor de heurística $h(s)$. Para problemas de busca de caminho, os valores de heurísticas normalmente são computados usando “Distância de Manhattan” ou “Distância Euclidiana”. O Algoritmo 2 apresenta o pseudo-código do LRTA*.

Algorithm 2 LRTA*($s_{início}, s_{meta}$)

```

1:  $s_{corrente} \leftarrow s_{início}$ 
2: loop
3:    $s' \leftarrow \arg \min_{s'' \in \text{succ}(s_{corrente})} (c(s_{corrente}, s'') + h(s''))$ 
4:   if  $s_{corrente} = s_{meta}$  then
5:      $h(s_{corrente}) \leftarrow h_0(s_{corrente})$ 
6:   else
7:      $h(s_{corrente}) \leftarrow \max(h(s_{corrente}), \min_{s'' \in \text{succ}(s_{corrente})} (c(s_{corrente}, s'') + h(s'')))$ 
8:   end if
9:   if  $s_{corrente} = s_{meta}$  then
10:    return sucesso
11:  end if
12:   $s_{corrente} \leftarrow s'$ 
13: end loop

```

Um valor h é associado a cada estado. O valor $h_0(s)$ corresponde ao valor inicial de h para o estado s . Primeiro o estado inicial $s_{início}$ é definido como sendo o estado corrente $s_{corrente}$. O LRTA* seleciona o estado para o qual deve se mover. Isso é feito avaliando o somatório mínimo $c(s_{corrente}, s'') + h(s'')$ dos estados adjacentes ao estado corrente. Se ocorrer a igualdade desta soma entre alguns estados, é escolhido o estado através de um sorteio (linha 3). A seguir o valor de heurística do estado corrente $h(s_{corrente})$ é atualizado. A atualização é realizada com a melhor estimativa de heurística, de modo que aproxime o estado até a meta (linhas 4-8). Posteriormente, o movimento é realizado para o estado selecionado (linha 12) e o procedimento realiza uma nova iteração. Ao final, quando a meta é encontrada o LRTA* termina sua execução com sucesso.

A Figura 2.9 apresenta um exemplo de aplicação do LRTA* sobre um problema de busca pelo melhor caminho. No mapa (a) é apresentado o número que representa cada

⁷Corresponde ao número de estados em profundidade que o algoritmo pode observar (coletar informações) a partir do estado atual.

estado. O objetivo é, a partir do estado 2 (marcado com um quadrado) alcançar o estado 50 (marcado com um círculo). No mapa (b) temos o custo de se locomover entre os estados. Todos os custos são iguais a 1, exceto o custo entre o estado 16 e 32, que é igual a 3. Nos mapas seguintes é apresentado o valor h de cada estado, sendo que a cada passo é realizada uma iteração do LRTA*. Em (c) temos o mapa inicial. O primeiro movimento é realizado, executando a ação de se locomover do estado 2 para o 18. Isso ocorre porque entre os valores $h(1)$, $h(3)$ e $h(18)$, $h(18)$ é o menor. O estado corrente deixa de ser 2 e passa a ser 18, como é apresentado no mapa (d). Ao executar uma iteração sobre o mapa (e), o valor de heurística do estado 34 é atualizado de 1 para 3. Isso ocorre pois o menor valor h dentre os estados adjacentes mais o custo é superior a $h(34)$. Em (f) é apresentado o mapa após a atualização de heurística do estado 34. Nos passos seguintes são apresentados alguns movimentos realizados durante a busca e os valores de h atualizados. No último mapa (r) é apresentado o resultado final da busca. O caminho consiste no conjunto de estados (2, 3, 4, 20, 36, 52, 51, 50).

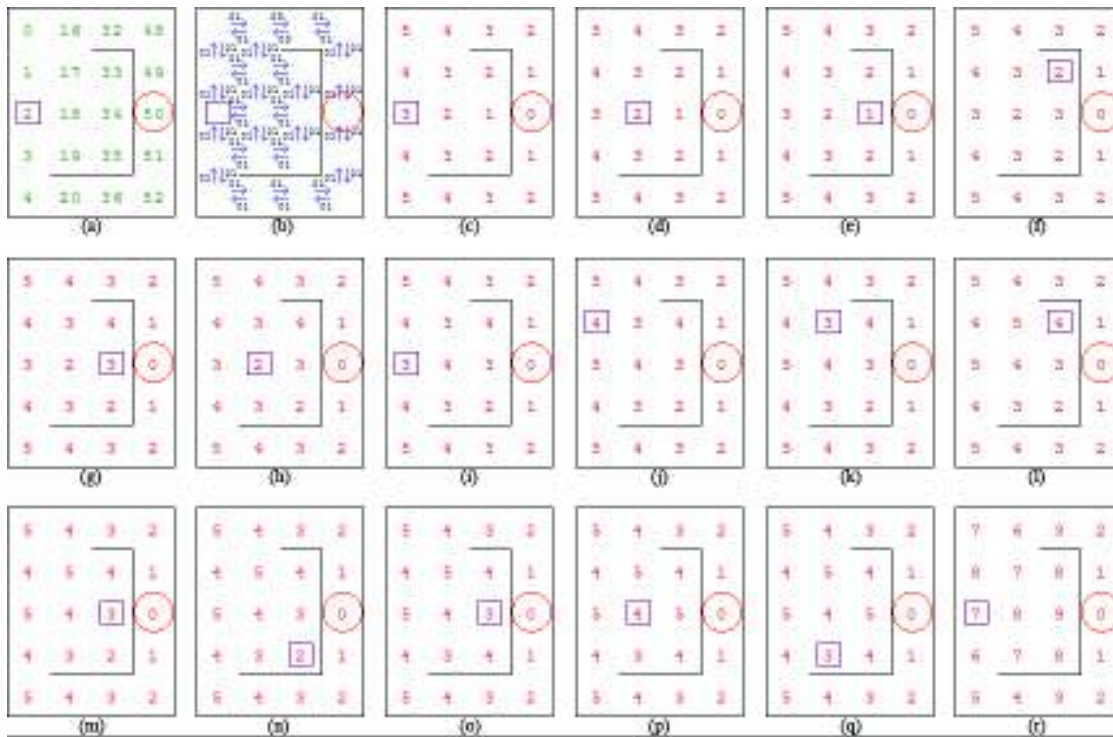


Figura 2.9: Exemplo de aplicação do LRTA*.

Algumas características do LRTA* são [Furcy 2004]:

1. O valor de h nunca decresce e se mantém admissível.
2. Se existe um caminho do estado inicial até a meta, a execução termina. Uma iteração do LRTA* consiste em percorrer o caminho do estado inicial até a meta.
3. Se reiniciada a busca e mantém-se o valor de heurística, o menor custo para os

estados vai sendo construído. Ao final de um número finito de iterações o melhor caminho é encontrado.

Apesar da garantia de convergência do LRTA*, é necessário observar o custo para encontrar tal solução. Dependendo do problema é preciso um grande número de iterações. Sendo assim, existem diversos algoritmos de busca em tempo real baseados no LRTA* que pretendem diminuir o número de iterações que são gastos para encontrar a solução [Bulitko e Lee 2006].

2.2.3 SLA*

Assim como o LRTA*, o algoritmo *Search and Learn A** (SLA*), também chamado busca e aprendizado A*, é usado para problemas com restrição de tempo. O SLA* foi projetado para incrementar a performance do LRTA*, de modo que a solução ótima seja encontrada em apenas uma iteração. Sua aplicação é baseada em uma estrutura de pilha para guardar as soluções e realizar um procedimento de retrocesso que atualiza o valor de heurística dos estados precedentes. O aprendizado é composto exatamente por essa atualização.

O SLA* pode ser aplicado em diversos tipos de problemas, como *pathfinder* e escalonamento. Este trabalho destaca o uso do SLA* em decorrência de sua aplicação em [Zamani e Shue 1998, Shue e Li 2008] para o escalonamento de recursos. É importante salientar que nestes trabalhos os problemas são compostos por apenas um recurso constante. O SLA* é apresentado no Algoritmo 3.

Algorithm 3 SLA*($s_{início}$, s_{meta})

```

1: PILHA.adicionar( $s_{início}$ )
2: loop
3:    $s_{corrente} \leftarrow PILHA.topo()$ 
4:   if  $s_{corrente} = s_{meta}$  then
5:     return PILHA
6:   end if
7:    $s' \leftarrow \arg \min_{s'' \in succ(s_{corrente})} (k(s_{corrente}, s'') + h(s''))$ 
8:   if  $h(s_{corrente}) \geq k(s_{corrente}, s') + h(s')$  then
9:     PILHA.adicionar( $s'$ )
10:  else
11:     $h(s_{corrente}) \leftarrow k(s_{corrente}, s') + h(s')$ 
12:    if  $s_{corrente} \neq s_{início}$  then
13:      PILHA.remover()
14:    end if
15:  end if
16: end loop

```

Como descrito anteriormente, o SLA* consiste no uso de uma pilha (*PILHA*) e na atualização do valor de heurística através de um procedimento de retrocesso. Primeiro, o

estado inicial é adicionado a pilha. A seguir são realizadas iterações na busca da solução. O elemento do topo da pilha é verificado. Se for uma meta o procedimento pára e retorna a solução. Caso contrário, é realizada uma busca, onde todos os estados adjacentes são visitados e o melhor sucessor é escolhido $s' \leftarrow \min_{s'' \in \text{succ}(s_{\text{corrente}})} (k(s_{\text{corrente}}, s'') + h(s''))$. Se o valor de heurística do estado corrente $h(s_{\text{corrente}})$ é menor que $k(s_{\text{corrente}}, s') + h(s')$, é realizada a atualização (aprendizado) de heurística $h(s_{\text{corrente}}) \leftarrow k(s_{\text{corrente}}, s') + h(s')$. Em seguida, é realizado o retrocesso. Caso contrário, este elemento é adicionado ao topo da pilha. Ao final o processo se repete até que a solução ótima seja encontrada.

Segundo [Ahsan e Tsao 2003] o algoritmo SLA* não encontra a solução ótima em decorrência de uma propriedade que este mesmo autor apresenta, chamada *Nil activity schedule*. Entretanto, [Zamani e Shue 2003] argumentam que este ponto levantado é considerado pelo SLA*. Além disso, estes autores demonstram matematicamente que o SLA* encontra a solução ótima. No Apêndice B é apresentada a prova que o SLA* encontra a solução ótima.

Exemplo de Aplicação do SLA*

O problema de Escalonamento de Projetos com Restrição de Recursos (*Project Scheduling with Resource Constraints* - PSRC) envolve o escalonamento de um conjunto de atividades de um projeto com restrição de recursos e restrição de precedência. O objetivo do PSRC é de minimizar o tempo do projeto determinando o tempo de todas as atividades. Quando o SLA* é aplicado aos problemas PSRC, cada estado corresponde a um escalonamento. Os sucessores são criados considerando os recursos do estado atual e as ações disponíveis.

Na Figura 2.10 é mostrado um projeto de atividades onde será demonstrada a aplicação do SLA* [Zamani e Shue 1998]. Cada atividade está associada a uma duração, o recurso necessário para sua execução e a maior duração para completar. A maior duração para completar corresponde ao maior tempo que é necessário para sua execução considerando apenas o tempo de todas as atividades que precedem à atividade.

Uma atividade pode ser executada apenas se as atividades precedentes já foram executadas. Em outras palavras, as atividades 1, 2 e 3, podem ser executadas apenas ao final da atividade 0. A atividade 4 pode ser executada apenas ao final da atividade 1, e assim sucessivamente para as outras atividades.

O máximo de recurso que pode ser usado em um passo do escalonamento é 180. Ou seja, é possível escalonar ao mesmo tempo as atividades 1 e 2, pois o recurso necessário para estas atividades é 120. Entretanto, não é possível escalonar ao mesmo tempo as atividades 1 e 3, pois o recurso necessário é igual a 181, ultrapassando o limite de 180.

Antes de demonstrar a aplicação do SLA* no projeto de atividades da Figura 2.10 são feitas algumas considerações. Posteriormente, será descrito cada passo do escalonamento.

sua execução, passando do conjunto *em progresso* para *completo*. Liberando os recursos antes ocupados. Com isso, qualquer combinação de atividades pode ser feita até o conjunto *completo*.

O custo associado com a transição de um estado para o próximo é o intervalo de tempo entre os dois estados. A transição entre estados é causada por um evento de conclusão de atividade, que devido à liberação dos recursos anteriormente ocupados, levará a um novo escalonamento. O custo de uma transição corresponde ao tempo de duração de uma atividade ao término de sua execução, ou seja, no momento que essa atividade passa do conjunto *em progresso* para *completo*.

Estimativa de Heurística

A estimativa de heurística de um estado, segundo as definições anteriores, é um limite inferior de duração da estimativa para conclusão do restante de atividades do projeto. Para não superestimar o valor de heurística, os métodos podem ser baseados em duas premissas: a precedência de atividades e a restrição de recursos. Estas quando aplicadas simultaneamente irão resultar em uma conclusão com tempo superior ou igual à aquele com apenas uma restrição. Assim, se for ignorada uma das duas restrições, temos a certeza de obter uma solução menor do que a ótima, e será uma solução que satisfaz o limite inferior, não superestimando a heurística.

Quando a restrição de recursos é ignorada, o método *Critical Path Method* (CPM) pode ser aplicado para encontrar a estimativa de heurística de um estado, sendo a duração do maior caminho para completar o restante das ações precedentes. Outro método para o cálculo de heurística consiste em ignorar a restrição de precedência e estimar a partir dos recursos das atividades *não escalonadas*. Em problemas com mais de um recurso é possível aplicar ambas as abordagens [Shue e Li 2008].

Exemplo

Para o exemplo da Figura 2.10 existe apenas um recurso com quantidade igual a 180. São 7 atividades que devem ser escalonadas. A heurística é calculada ignorando a restrição de recurso, sendo exibida na parte inferior de cada atividade. O escalonamento ótimo para este projeto é 03-125-46, encontrado após oito execuções do aprendizado de heurística. Este escalonamento, 03-125-46, é mostrado na Figura 2.11. É importante lembrar que as atividades 0 e 6 não possuem duração nem recurso. Como citado em [Shue e Li 2008], este exemplo foi projetado de modo que a atividade 3 seja escalonada antes de 1 e 2, para que ocorra o efeito acumulativo de aprendizagem.

Na Tabela 2.1 é apresentado o escalonamento para o projeto de atividades da Figura 2.10. A coluna n mostra o aprendizado até que a solução ótima seja encontrada. A coluna t corresponde ao tempo para a próxima seleção de estado.

Tabela 2.1: Resultado completo da busca para o exemplo de projeto de atividades.

n	t	Estado Atual e Estimativa de Heurística	Estados Filhos e Estimativa de Heurística	Atualização de Heurística
1	0	(0,,123456) 21	(0,1,23456) 28 (0,2,12456) 31 (0,3,12456) 29 (0,12,3456) 28 ^s	(0,,123456) 28
2	0	(0,,123456) 28	(0,1,23456) 28 (0,2,13456) 31 (0,3,12456) 29 (0,12,3456) 28 ^s (012,3,456) 29 ^s	nulo (012,,3456) 29
3	0	(0,,123456) 28	(0,1,23456) 28 ^s (0,2,13456) 31 (0,3,12456) 29 (0,12,3456) 29 (01,2,3456) 38 (01,3,2456) 29 ^s (01,4,2356) 39	nulo (01,,23456) 29
4	0	(0,,123456) 28	(0,1,23456) 29 (0,2,13456) 31 (0,3,12456) 29 (0,12,3456) 29 ^s	(0,,123456) 29
5	0	(0,,123456) 29	(0,1,23456) 29 (0,2,13456) 31 (0,3,12456) 29 (0,12,3456) 29 ^s (012,3,456) 29 ^s (012,4,356) 39 (0123,4,56) 39 ^s	nulo nulo (0123,456) 39
6	10	(012,,3456) 29	(012,3,456) 39 ^s (012,4,356) 39	(012,,3456) 39
7	0	(0,,123456) 29	(0,1,23456) 29 ^s (0,2,13456) 31 (0,3,12456) 29 (0,12,3456) 39 (01,2,3456) 38 (01,3,2456) 29 ^s (01,4,2356) 39 (013,2,456) 39 (013,4,256) 39 ^s	nulo nulo (013,,2456) 39
8	10	(01,,23456) 29	(01,2,3456) 38 ^s (01,3,2456) 39 (01,4,2356) 39 (0,1,23456) 38 (0,2,13456) 31 (0,3,12456) 29 ^s (0,12,3456) 39 (03,1,2456) 29 (03,2,1456) 39 (03,5,1246) 39 (03,12,456) 29 (03,15,246) 29 (03,25,146) 39 (03,125,46) 29 ^s (03125,4,6) 29 ^s	(01,,23456) 38 nulo nulo nulo
	0	(0,,123456) 29	(0,1,23456) 38 (0,2,13456) 31 (0,3,12456) 29 ^s (0,12,3456) 39 (03,1,2456) 29 (03,2,1456) 39 (03,5,1246) 39 (03,12,456) 29 (03,15,246) 29 (03,25,146) 39 (03,125,46) 29 ^s (03125,4,6) 29 ^s	nulo nulo nulo
	8	(03,,12456) 29	(03,1,2456) 29 (03,2,1456) 39 (03,5,1246) 39 (03,12,456) 29 (03,15,246) 29 (03,25,146) 39 (03,125,46) 29 ^s (03125,4,6) 29 ^s	nulo nulo nulo
	18	(03125,,46) 29	(03125,4,6) 29 ^s	nulo nulo
	29	(031254,6,) 29		
	29	(0312546,,) 29		

^s Indica o estado selecionado.

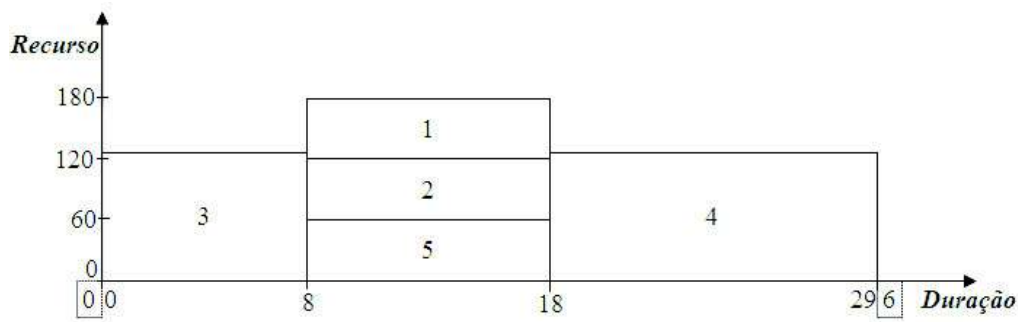


Figura 2.11: Resultado do escalonamento do projeto de atividades.

Para cada seleção de estado, se existirem valores mínimos de heurística iguais entre estados, é escolhido o que possui o maior número de atividades. Mantendo-se a igualdade, é escolhido qualquer um destes estados. A coluna “Estado Atual e Estimativa de Heurística” descreve o estado em um dado momento que está sendo usado na busca e seu valor estimado de heurística. “Estados Filhos e Estimativa de Heurística” apresenta quais são os estados filhos do atual estado da busca, ou seja, são os estados correspondentes ao escalonamento das atividades. Finalmente, “Atualização de Heurística” descreve o valor de aprendizado da heurística durante a busca. Se este valor é **nulo** significa que não ocorreu aprendizado naquele momento. A Figura 2.12 mostra os primeiros estados da busca.

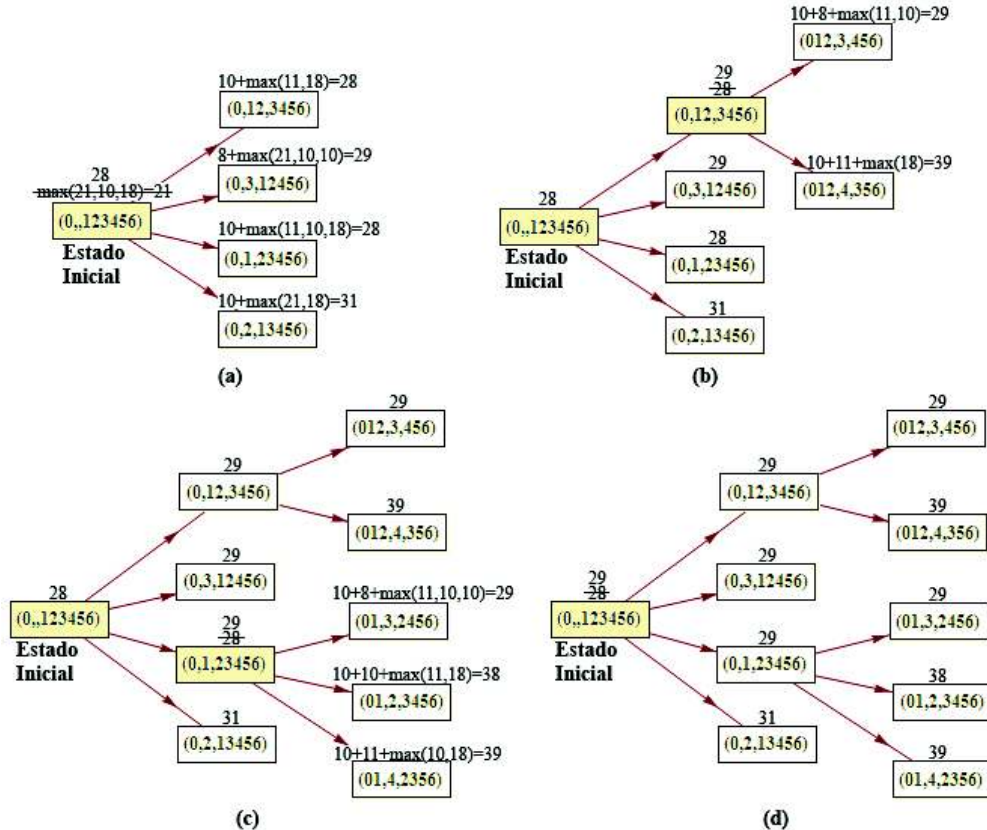


Figura 2.12: Primeiros estados da busca para o exemplo de projeto de atividades.

Inicialmente, para o estado $(0,123456)$ o valor de heurística é $\max(21, 10, 18) = 21$. Os estados filhos com as possíveis combinações de atividades são $(0,1,23456)$, $(0,2,13456)$, $(0,3,12456)$ e $(0,12,3456)$. Os valores de heurística são, respectivamente, $\{10 + \max(11, 10, 18) = 28\}$, $\{10 + \max(21, 18) = 31\}$, $\{8 + \max(21, 10, 10) = 29\}$ e $\{10 + \max(11, 18) = 28\}$. A menor heurística é 28. Entretanto, dois estados possuem este valor. É selecionado o estado que possui a maior quantidade de atividades; neste caso é $(0,12,3456)$. Desta forma para $(0,123456)$ o valor de heurística é atualizado de 21 para 28.

Em seguida a busca se repete para o estado inicial. O estado $(0,12,3456)$ que possui a menor heurística é selecionado e a busca se repete. Este procedimento de busca se repete sucessivamente até encontrar a melhor solução. Todos os estados da busca são exibidos na Figura 2.13.

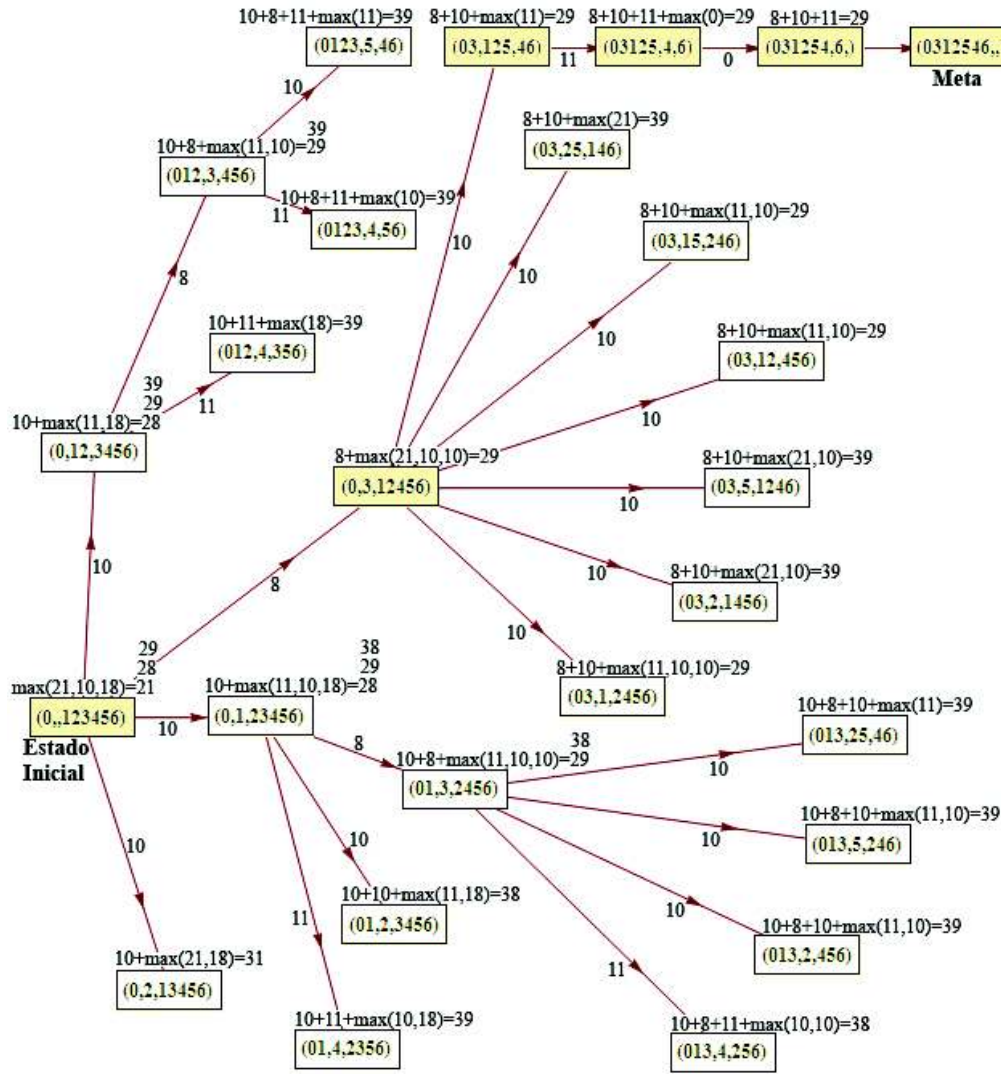


Figura 2.13: Resultado completo da busca com todos os estados.

A heurística do primeiro estado da *PILHA* é atualizada apenas se ocorrer um retrocesso até o estado inicial. Essa busca se repete até encontrar a solução ótima.

2.3 Planejamento

O planejamento em IA envolve determinar um conjunto ordenado de ações que quando executadas por um ou mais agentes a partir de um estado inicial que satisfaça as circunstâncias dadas, resulte num estado final que satisfaça uma dada meta [Russell e Norvig 2003]. Nesta Seção são apresentados os conceitos de planejamento e as abordagens mais comuns para a solução deste problema.

2.3.1 Conceitos Relacionados

O problema de planejamento é definido por um estado inicial do mundo, 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 determinístico⁸ é dado formalmente pela Definição 2.2.

Definição 2.2. O problema de planejamento determinístico é definido formalmente como uma 5-upla $\langle S, i, A, g, P \rangle$, 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. P representa a solução do problema (normalmente chamado **plano**). O plano é formado por um conjunto ordenado e finito de ações.

O componente chave para o planejamento é a linguagem de representação. É preciso que a linguagem tenha 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) [Russell e Norvig 2003].

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, $Rico \wedge Conhecido$ poderia representar o estado de uma pessoa rica e conhecida. Outro modo de representar os estados são com literais de primeira ordem: $Em(Avião_1, Uberlândia) \wedge Em(Avião_2, Uberaba)$

⁸No momento que o plano é executado o resultado final é exatamente o desejado. O ambiente é estático e as ações têm exatamente o comportamento esperado (“Planejamento Clássico”).

poderia representar um estado em um problema de entrega de pacotes. Este problema consiste em enviar pacotes entre cidades utilizando aviões.

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 voar em um avião de um local para outro pode ser descrita da seguinte forma:

Ação(Voar($p, de, para$))

pré-condição: $Em(p, de) \wedge Avião(p) \wedge Aeroporto(de) \wedge Aeroporto(para)$

efeito: $\neg Em(p, de) \wedge Em(p, para)$

Este esquema representa várias possibilidades de ações que podem ser derivadas pela instanciação das variáveis p, de e $para$. Alguns sistemas de planejamento dividem os efeitos em uma **lista de adição** para literais positivos e uma **lista de eliminação** para literais negativos.

Na Figura 2.14 é apresentado um problema de planejamento utilizando STRIPS chamado “*Mundo dos Blocos*”. O problema consiste em mover os blocos usando as ações **Mover** e **MoverParaMesa**. O objetivo é alterar o *Estado Inicial* do mundo 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 **Mover**(b, x, y) move um bloco b que está sobre x para y , se ambos b e y estão livres. Um bloco livre significa que não existe outro bloco sobre ele. Depois que o movimento é feito, x fica livre mas y não. Quando $x = mesa$ a ação **Mover**(b, x, y) 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 **MoverParaMesa**(b, x) move um bloco b que está sobre x para a mesa.

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

Existem diversas soluções para o problema da Figura 2.14. A seguir são apresentados

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 ($Mover(b, x, y)$,
 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 ($MoverParaMesa(b, x)$,
 pré-condição: $Sobre(b, x) \wedge Livre(b) \wedge Bloco(b) \wedge Bloco(x) \wedge (b \neq x)$
 efeito: $Sobre(b, mesa) \wedge Livre(x) \wedge \neg Sobre(b, x)$)

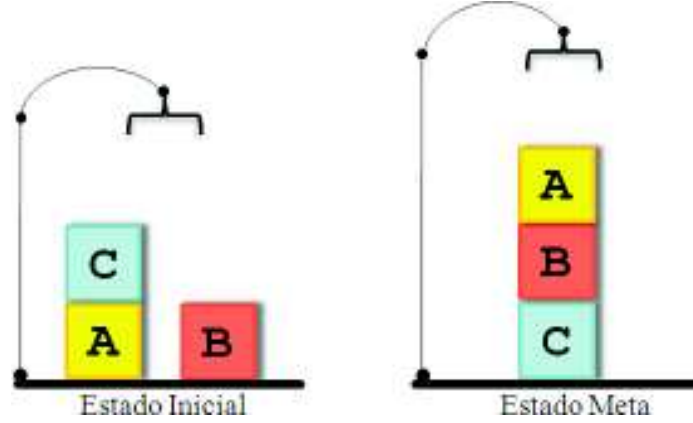


Figura 2.14: Problema “Mundo dos Blocos” representado pela linguagem STRIPS.

dois planos que resolvem este problema ⁹. 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: $MoverParaMesa(C, A) \prec Mover(B, mesa, C) \prec Mover(A, mesa, B)$

- **Estado Inicial:** $Sobre(A, Mesa) \wedge Sobre(B, Mesa) \wedge \textbf{Sobre}(C, A) \wedge \textbf{Bloco}(A) \wedge Bloco(B) \wedge \textbf{Bloco}(C) \wedge Livre(B) \wedge \textbf{Livre}(C) \wedge Livre(mesa) \Leftarrow MoverParaMesa(C, A)$
- $Sobre(A, Mesa) \wedge \textbf{Sobre}(B, Mesa) \wedge Sobre(C, Mesa) \wedge Bloco(A) \wedge \textbf{Bloco}(B) \wedge \textbf{Bloco}(C) \wedge Livre(A) \wedge \textbf{Livre}(B) \wedge \textbf{Livre}(C) \wedge Livre(mesa) \Leftarrow Mover(B, mesa, C)$
- $Sobre(B, C) \wedge \textbf{Sobre}(A, Mesa) \wedge Sobre(C, Mesa) \wedge \textbf{Bloco}(A) \wedge \textbf{Bloco}(B) \wedge Bloco(C) \wedge \textbf{Livre}(A) \wedge \textbf{Livre}(B) \wedge Livre(mesa) \Leftarrow Mover(A, mesa, B)$
- **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: $MoverParaMesa(C, A) \prec Mover(A, mesa, B) \prec MoverParaMesa(A, B) \prec Mover(B, mesa, C) \prec Mover(A, mesa, B)$

- **Estado Inicial:** $Sobre(A, Mesa) \wedge Sobre(B, Mesa) \wedge \textbf{Sobre}(C, A) \wedge \textbf{Bloco}(A) \wedge Bloco(B) \wedge \textbf{Bloco}(C) \wedge Livre(B) \wedge \textbf{Livre}(C) \wedge Livre(mesa) \Leftarrow MoverParaMesa(C, A)$

⁹ $a_1 \prec a_2$ significa que a ação $a_1 \in A$ deve ser executada antes de $a_2 \in A$. $s \Leftarrow a_3$ significa que sobre o estado $s \in S$ será executada a ação $a_3 \in A$.

- $\text{Sobre}(A, \text{Mesa}) \wedge \text{Sobre}(B, \text{Mesa}) \wedge \text{Sobre}(C, \text{Mesa}) \wedge \text{Bloco}(A) \wedge \text{Bloco}(B) \wedge \text{Bloco}(C) \wedge \text{Livre}(A) \wedge \text{Livre}(B) \wedge \text{Livre}(C) \wedge \text{Livre}(\text{mesa}) \Leftarrow \text{Mover}(A, \text{mesa}, B)$
- $\text{Sobre}(A, B) \wedge \text{Sobre}(B, \text{Mesa}) \wedge \text{Sobre}(C, \text{Mesa}) \wedge \text{Bloco}(A) \wedge \text{Bloco}(B) \wedge \text{Bloco}(C) \wedge \text{Livre}(A) \wedge \text{Livre}(C) \wedge \text{Livre}(\text{mesa}) \Leftarrow \text{MoverParaMesa}(A, B)$
- $\text{Sobre}(A, \text{Mesa}) \wedge \text{Sobre}(B, \text{Mesa}) \wedge \text{Sobre}(C, \text{Mesa}) \wedge \text{Bloco}(A) \wedge \text{Bloco}(B) \wedge \text{Bloco}(C) \wedge \text{Livre}(A) \wedge \text{Livre}(B) \wedge \text{Livre}(C) \wedge \text{Livre}(\text{mesa}) \Leftarrow \text{Mover}(B, \text{mesa}, C)$
- $\text{Sobre}(B, C) \wedge \text{Sobre}(A, \text{Mesa}) \wedge \text{Sobre}(C, \text{Mesa}) \wedge \text{Bloco}(A) \wedge \text{Bloco}(B) \wedge \text{Bloco}(C) \wedge \text{Livre}(A) \wedge \text{Livre}(B) \wedge \text{Livre}(\text{mesa}) \Leftarrow \text{Mover}(A, \text{mesa}, B)$
- **Estado Meta:** $\underline{\text{Sobre}(A, B)} \wedge \underline{\text{Sobre}(B, C)} \wedge \underline{\text{Sobre}(C, \text{Mesa})} \wedge \text{Bloco}(A) \wedge \text{Bloco}(B) \wedge \text{Bloco}(C) \wedge \text{Livre}(A) \wedge \text{Livre}(\text{mesa})$

No “Plano 1” observamos que no último estado obtivemos exatamente a meta desejada, ou seja: $\text{Sobre}(A, B) \wedge \text{Sobre}(B, C) \wedge \text{Sobre}(C, \text{Mesa})$. O “Plano 2” não é a melhor solução, já que necessita de mais ações. Mesmo assim ele não deixa ser uma solução.

Apesar de STRIPS possibilitar a especificação de vários problemas reais, ele ainda é insuficiente para representar alguns 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, e possibilita definir tempo de duração para as ações e descrever os efeitos do tempo sobre as ações. Além destas modificações, inclui o tratamento de expressões numéricas e permite especificar uma função de otimização do plano (denominada métrica), a ser descrita na própria representação do problema.

Dependendo de como é definido um problema, podemos executar uma ou mais ações ao mesmo tempo. 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. Geralmente o planejamento é dividido em “Planejamento Clássico” e “Planejamento não-Clássico”.

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

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

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 UCPPOP (*Sensory Graphplan* - SGP). Além destes trabalhos, existem diversos estudos correlatos comparando os métodos de planejamento e propondo melhorias. Nas próximas seções são abordados alguns métodos de planejamento.

2.3.2 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 até uma meta $g \in G$ usando por exemplo um algoritmo de busca do menor caminho. O plano é uma simples seqüência de operadores que correspondem as 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 é muito grande, já que vai criar muitos estados. Em um problema com 20 ou 30 variáveis booleanas 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 a aplicação mais fácil dos algoritmos de busca no uso geral 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.

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, 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, 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 et al. 2005]. Para o problema de planejamento ambos os métodos de busca podem ser aplicados.

Planejamento com Busca Progressiva

A Busca Progressiva começa no estado inicial do problema, considerando seqüências de ações até encontrar uma seqüência que alcance um estado meta. O estado inicial da busca é o estado inicial do problema de planejamento. Em geral, cada estado será um conjunto de literais positivos (literais que não aparecem são falsos).

As ações aplicáveis 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. O teste que verifica se uma meta foi encontrada consiste em determinar se o estado satisfaz a meta do problema de planejamento. A Figura 2.15 apresenta o primeiro passo da busca progressiva sobre o problema da Figura 2.14.

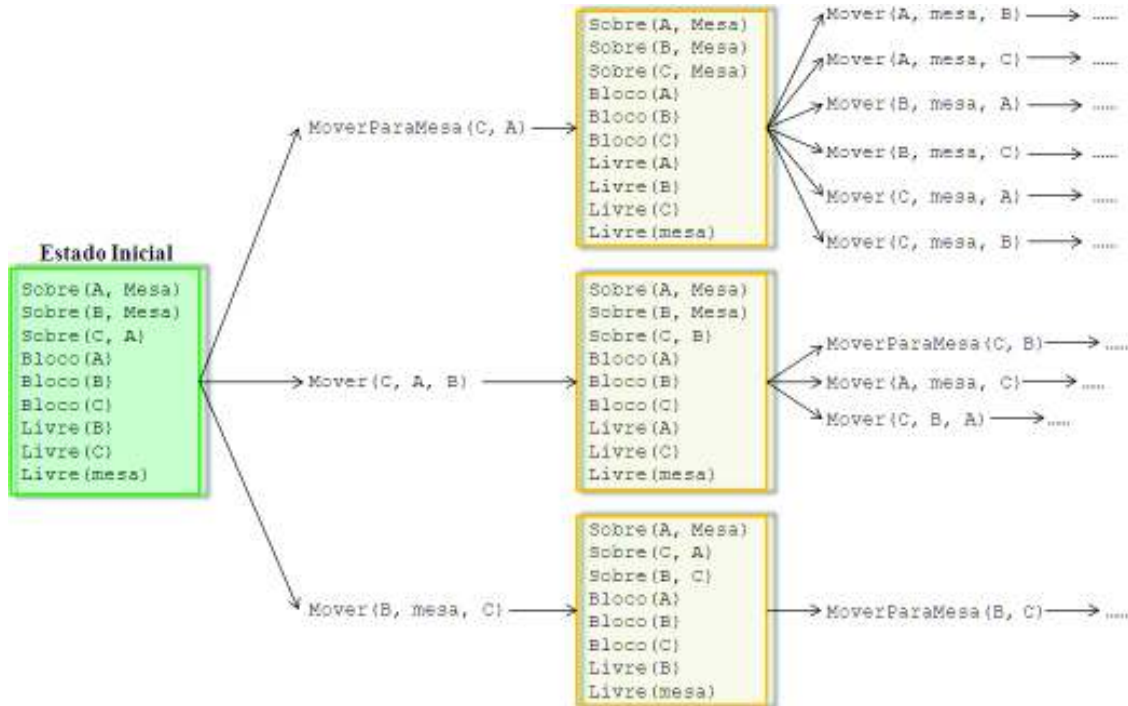


Figura 2.15: Primeiro passo de uma busca progressiva para o problema da Figura 2.14.

O espaço de estados de um problema de planejamento é finito. Portanto, qualquer algoritmo de busca de grafos que seja completo será um algoritmo de planejamento completo. Um bom algoritmo de busca normalmente utiliza um valor de heurística. Por exemplo, o A* é um bom algoritmo, já que utiliza um valor de heurística que otimiza a busca. O problema da busca progressiva é que ela considera todas as ações aplicáveis, incluindo aquelas irrelevantes. Logo, uma boa abordagem fracassa sem uma boa heurística.

No exemplo da Figura 2.15, a ação `Mover(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 `Mover(C, A, B)`. Ao final da busca a solução do problema de planejamento é composto 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 2.2 é igual a P .

Planejamento com Busca Regressiva

A Busca Regressiva aplicada sobre o planejamento considera inicialmente os literais da 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 2.16 é exibido o primeiro passo da busca regressiva sobre o problema da Figura 2.14. São consideradas apenas as ações que possuem algum efeito que satisfaz algum literal da meta.

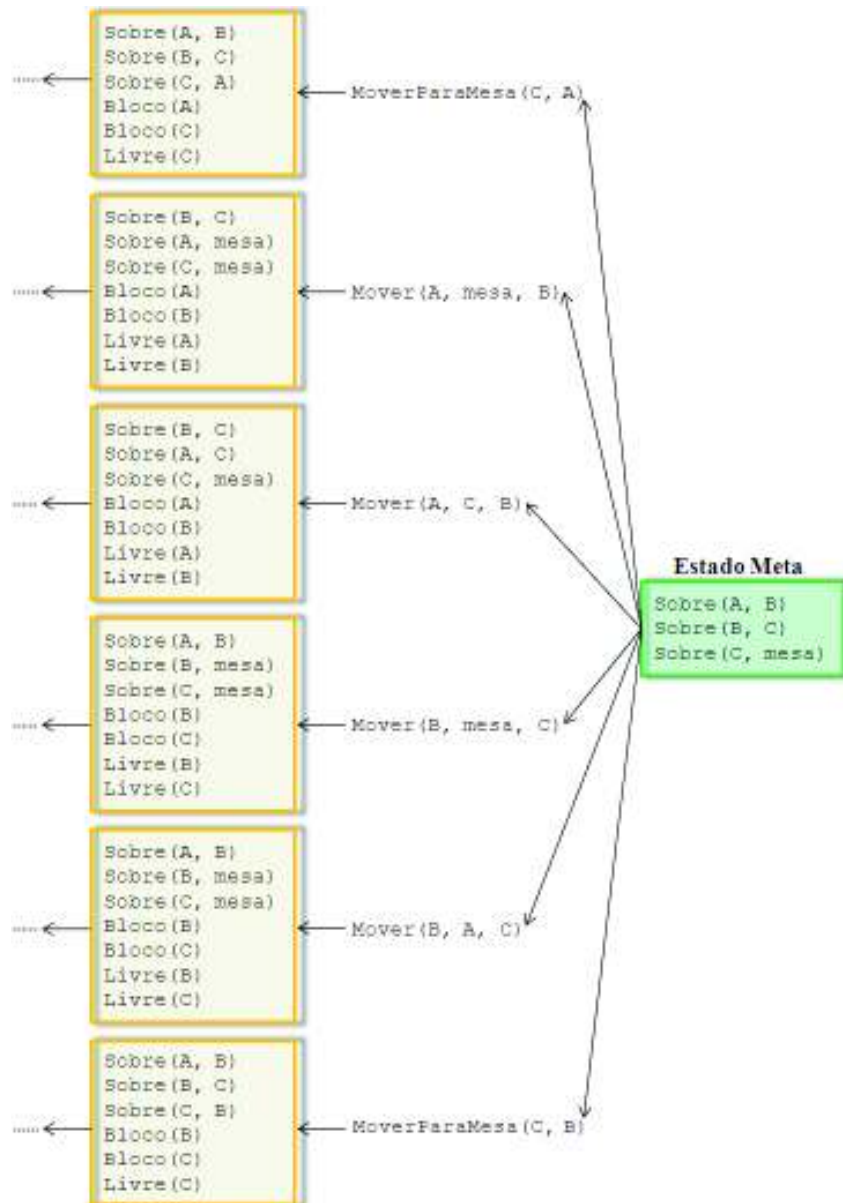


Figura 2.16: Primeiro passo da busca regressiva para o problema da Figura 2.14.

Entretanto, existem muitas ações irrelevantes que também podem levar a um estado meta. Uma busca para trás que permite ações irrelevantes ainda será completa, mas será menos eficiente. Se existir uma solução ela será encontrada e terá apenas ações relevantes. 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 os estados a partir dos quais a aplicação de uma ação leva a meta”. Um novo estado na busca consiste nas pré-condições da ação e os literais que não foram satisfeitos pela ação. Além de inserir apenas ações que alcancem algum literal desejado, nenhum efeito da ação pode negar parte da meta. Se isso ocorrer a ação não faz parte da busca naquele momento. As ações que satisfazem essa restrição são chamadas consistentes. Basicamente, um estado predecessor de uma ação $a \in A$ que satisfaz uma meta $g \in G$ é igual a:

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

Na Figura 2.16 o estado predecessor da ação $\text{Mover}(A, \text{mesa}, B)$ é formado por:

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

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.

2.3.3 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 lhe chamaram de um “solucionador de problemas”, atualmente chamado de “planejador”, sendo uma terminologia mais apropriada hoje na IA [Fikes e Nilsson 1971].

Neste contexto, “resolver problemas” significa encontrar uma seqüência de operadores no espaço de busca do mundo que irá transformar o estado inicial para um outro em que uma determinada fórmula (meta) pode ser provada como verdade. Duas principais técnicas são utilizadas na resolução de problemas: *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 onde são transformados através de operadores, eles 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 consiste em adicionar e deletar do estado do mundo todas aquelas cláusulas especificadas pelas funções de deleção e adição prescritas no 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 a técnica MEA.

Inicialmente STRIPS foi projetado para controlar o robô Shakey. A Figura 2.17 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.

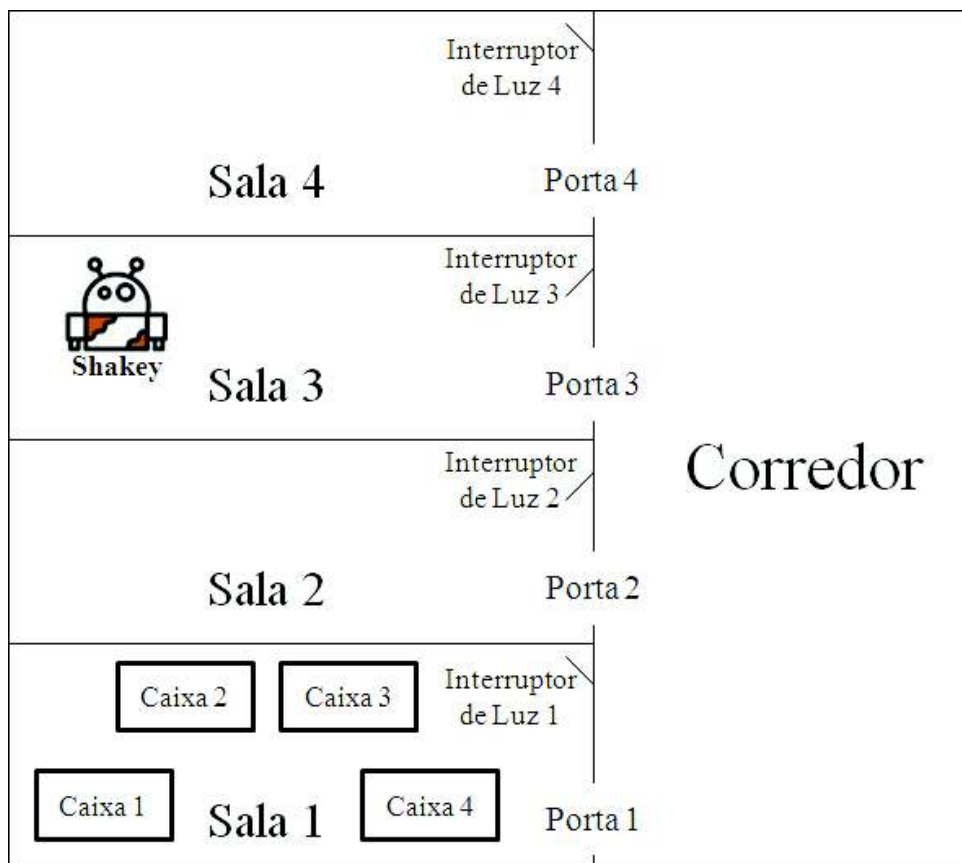


Figura 2.17: Problema “Mundo de Shakey”.

No “Mundo de Shakey” o robô pode se mover entre 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. As ações possíveis são:

- $Ir(x, y)$: 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 2.18 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”.

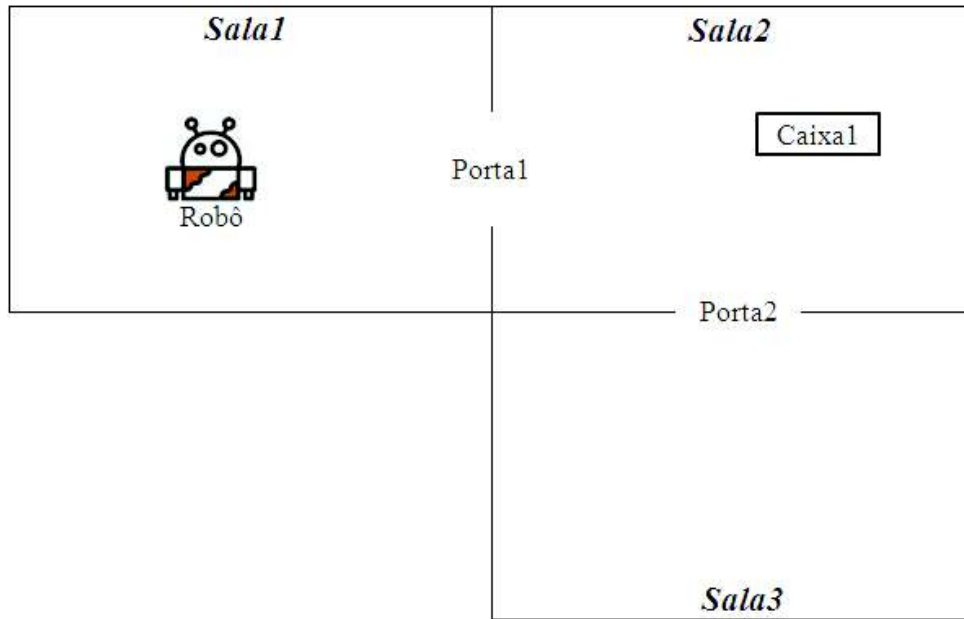


Figura 2.18: Problema “Mover a Caixa”.

Na Figura 2.19 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

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

Figura 2.19: Operadores do problema “Mover a Caixa”.

Estado Inicial

M0: $NASALA(Robô, Sala1)$
 $CONECTADO(Porta1, Sala1, Sala2)$
 $CONECTADO(Porta2, Sala2, Sala3)$
 $CAIXA(Caixa1)$
 $NASALA(Caixa1, Sala2)$
 $(\forall x, \forall y, \forall z) (CONECTADO(x, y, z) \Rightarrow CONECTADO(x, z, y))$

Metas

G0: $(\exists x) (CAIXA(x) \wedge NASALA(x, Sala1))$
 G1: $NASALA(Caixa, r1) \wedge NASALA(Robô, r1) \wedge CONECTADO(d, r1, Sala1)$
 G2: $NASALA(Robô, r1) \wedge CONECTADO(d, r1, Sala2)$

Figura 2.20: Estado Inicial e Meta do problema “Mover a Caixa”.

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 2.20. 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. Conseqüentemente, 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.

Finalmente, em G2 as pré-condições obtidas de **IR** são provadas serem verdadeiras em M0. Conseqüentemente, 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 2.21.

Planejamento

```

M1:  NASALA(Robô, Sala2)
      CONECTADO(Porta1, Sala1, Sala2)
      CONECTADO(Porta2, Sala2, Sala3)
      CAIXA(Caixa1)
      NASALA(Caixa1, Sala2)
      ( $\forall x, \forall y, \forall z$ ) (CONECTADO(x, y, z)  $\Rightarrow$  CONECTADO(x, z, y))
M2:  NASALA(Robô, Sala1)
      CONECTADO(Porta1, Sala1, Sala2)
      CONECTADO(Porta2, Sala2, Sala3)
      CAIXA(Caixa1)
      NASALA(Caixa1, Sala1)
      ( $\forall x, \forall y, \forall z$ ) (CONECTADO(x, y, z)  $\Rightarrow$  CONECTADO(x, z, y))

```

Plano

```

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

```

Figura 2.21: Resultado do planejamento do problema “Mover a Caixa”.

2.3.4 Planejamento de Ordem Parcial

O planejamento linear consiste em uma seqüência de ações. Isso ocorre pois o planejamento geralmente executa uma ação a cada momento. Entretanto, os planos não são necessariamente limitados pela linearidade das ações. Sendo assim, o planejamento de ordem parcial consiste em encontrar planos onde as ações sejam dispostas em paralelo. Em outras palavras, algumas ações podem ser executadas ao mesmo tempo (paralelo).

Tal abordagem apresenta a 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. 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 2.22 apresenta a comparação entre o planejamento de ordem parcial e o planejamento de ordem total (linear).

Originalmente, o planejamento de ordem parcial foi introduzido por [Sacerdoti 1975] como um meio de incrementar 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. Entretanto, o melhor método de planejamento depende da heurística e do procedimento de busca aplicados para solução do problema. Para detalhar estes

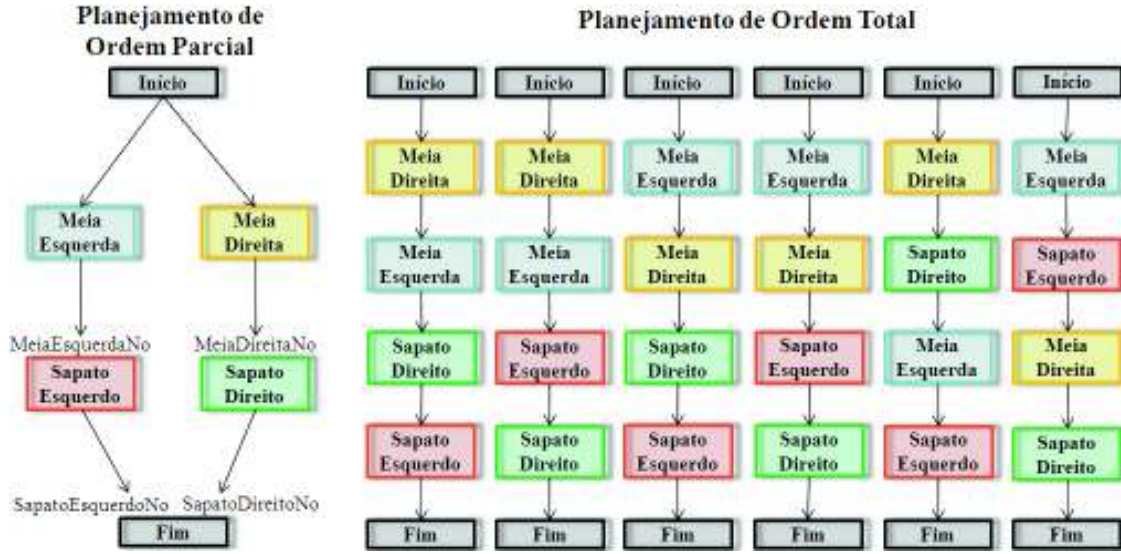


Figura 2.22: Comparação entre o planejamento de ordem parcial e total.

procedimentos, em [Rintanen et al. 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. Ordens que formam ciclos não podem ser adicionados 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 temos $Início \prec Fim$.

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 . Na Figura 2.22 o vínculo $MeiaEsquerda \xrightarrow{MeiaEsquerdaNo} SapatoEsquerdo$ 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 ocorre até encontrar um plano que não possua pré-condições em aberto. Na Figura 2.22 temos os seguintes componentes:

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

Ordens: $MeiaEsquerda \prec SapatoEsquerdo$

$MeiaDireita \prec SapatoDireito$

Vínculos Causais: $MeiaEsquerda \xrightarrow{MeiaEsquerdaNo} SapadoEsquerdo$
 $MeiaDireita \xrightarrow{MeiaDireitaNo} SapatoDireito$
 $SapatoEsquerdo \xrightarrow{SapatoEsquerdaNo} Fim$
 $SapatoDireito \xrightarrow{SapatoDireitoNo} Fim.$

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 contém 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.

Finalmente é demonstrado 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 2.23 utilizando a linguagem ADL (vai além de STRIPS pelo fato de aceitar literais negativos).

O resultado do planejamento é apresentado na Figura 2.24. 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 são descritas suas 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:

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 2.23: Descrição do problema “Pneu Furado”.

1. É realizada 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: $Remover(Sobressalente, PortaMalas)$.
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:

$$Remover(Sobressalente, PortaMalas) \xrightarrow{Em(Sobressalente, Chao)} Montar(Sobressalente, Eixo).$$
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*.

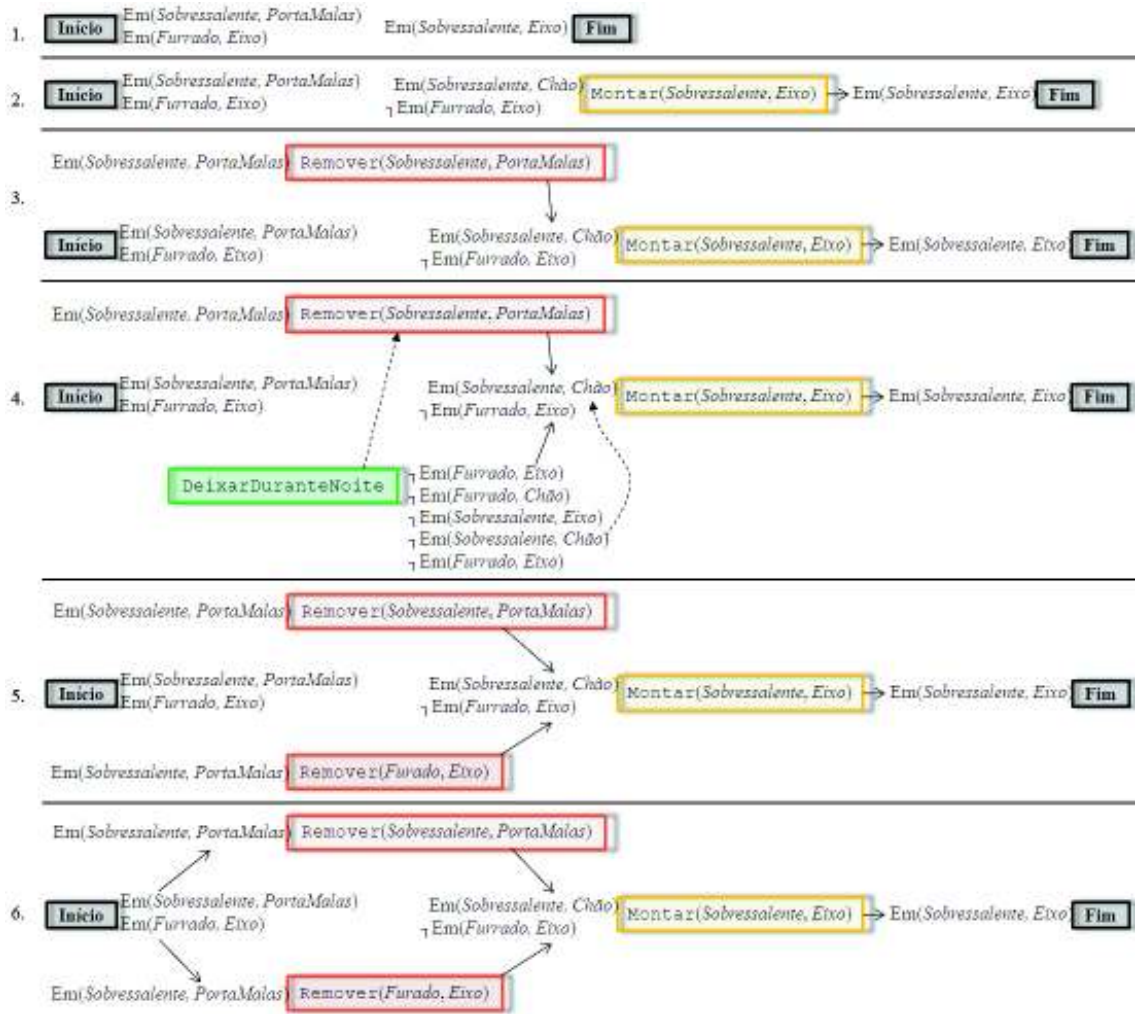


Figura 2.24: Planejamento de ordem parcial para o problema da Figura 2.23.

Em [Russell e Norvig 1995] é apresentado um pseudo-código do algoritmo POP. Este método consiste em uma busca baseada na escolha e conflito de operadores.

2.4 Planejamento de Recursos para Jogos RTS

Uma abordagem que envolve a aplicação de planejamento em jogos do tipo RTS é apresentada em [Chan et al. 2007b, Chan et al. 2007a]. O objetivo é criar planos que resolvam o problema de produção de recursos. 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 por [Chan et al. 2007a] é apresentado na Figura 2.8. Este mesmo domínio de ações é usado para os testes de planejamento deste trabalho. Como citado por [Chan et al. 2007a] seria possível utilizar uma linguagem mais tradicional para representar este domínio, tal como PDDL. Entretanto, a representação usada é fácil e descreve de forma

clara o domínio de ações. Para realizar os testes [Chan et al. 2007b, Chan et al. 2007a] utilizou o jogo *Wargus*.

Um dos problemas do planejamento de recursos para jogos RTS é que com o decorrer do jogo o mapa sofre alterações. Devido a estas alterações o tempo de duração de algumas ações é variável. Essa característica implica na necessidade de realizar novamente o planejamento para metas já computadas. Por exemplo, com o decorrer do jogo a *forest* vai sendo consumida, necessitando que o *peasant* tenha que buscar cada vez mais longe o recurso *wood*. Este problema faz parte dos trabalhos futuros de pesquisa.

A arquitetura de planejamento de [Chan et al. 2007b] é composta por três elementos. O primeiro é o Planejador Seqüencial que é responsável por retornar um plano seqüencial. O segundo elemento da arquitetura estabelece quais ações podem ser executadas em paralelo. Por último temos a Produção Adicional de Recursos que determina quais recursos devem ser criados para decrementar o *makespan*.

O planejador seqüencial determina quais ações devem fazer parte do plano para que a meta seja alcançada. Neste passo, todas as ações são estabelecidas seqüencialmente. O problema de estabelecer uma seqüência de ações é que o *makespan* é muito superior ao *makespan* de um plano com ações em paralelo. Sendo assim, algumas ações podem ser executadas ao mesmo tempo (paralelo), tornando o *makespan* menor.

Por exemplo, se o estado inicial é (*2 peasant, 1 townhall*) e a meta (*200 Gold*), o planejador seqüencial 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.

Para a tarefa de estabelecer quais ações podem ser executadas em paralelo temos o algoritmo de Escalonamento. Este procedimento busca quais ações podem ser executadas no tempo anterior de término ou início de outra ação.

A Produção de Recursos Adicionais utiliza o planejador seqüencial e a heurística de escalonamento para estabelecer se é possível criar novos recursos que decrementem o *makespan*. Por exemplo, para algumas metas é vantajoso gastar um tempo produzindo um *peasant* para ajudar a atingir a meta. Mesmo que sejam gastos recursos e tempo, o recursos adicional em alguns casos permite o decremento do *makespan*. Contudo, a Produção de Recursos Adicionais não foi explorada neste trabalho. Este tópico faz parte dos trabalhos futuros.

As próximas seções descrevem com detalhes estes métodos. Na Seção 2.4.1 é apresentado o Planejador Seqüencial. Na Seção 2.4.2 é descrita a Heurística de Escalonamento. Ao final, na Seção 2.4.3, é apresentada a Produção de Recursos Adicionais.

2.4.1 Planejador Seqüencial

O planejador seqüencial tem como base a técnica MEA. Como citado por [Chan et al. 2007b], devido as características do domínio tratado o planejador seqüencial baseado em MEA é suficiente. Ele opera selecionando uma submeta de modo a decrementar a diferença entre o estado inicial e a meta, e então executa as ações necessárias para resolver essa submeta. Através de um processo recursivo as outras submetas são resolvidas, até que todas as metas estejam satisfeitas.

O Algoritmo 4 apresenta o pseudo-código deste planejador seqüencial. Para simplificar este algoritmo assumimos que não há recurso que seja produzido e consumido ao mesmo tempo por uma mesma ação. A seguir será descrito informalmente este algoritmo com os detalhes do MEA considerando o domínio do *Wargus* (Figura 2.8).

Algorithm 4 $MEA(S, G)$

```

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:  $k \leftarrow \lceil (g_i - r_i) / \alpha \rceil$ 
10:  $Acts \leftarrow$  plano seqüencial 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 pré-condição “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 seqüencialmente  $Plan'$  a partir de  $PS$ 
23: return  $concatenar(Plan', MEA(S', G))$ 

```

Com base nas ações do domínio da Figura 2.8 será caracterizado o comportamento do algoritmo MEA. Em primeiro lugar, o MEA escolhe repetidamente uma submeta insatisfeita $R_i \geq g_i$, e constrói um subplano $Plan'$ que o satisfaça. Segundo [Chan et al. 2007b], “é possível mostrar que, dado um gráfico de dependência de recursos, existe uma ordenação sobre os recursos de tal forma que o MEA não necessitará rever uma meta resolvida. Esta ordem de recursos resolve toda meta de recurso renovável antes de qualquer meta de recurso não renovável (porque as metas de recursos renováveis, uma vez resolvidas, sempre

ficaram resolvidas usando a propriedade monotônica de aumento de recursos renováveis).” Nesta ordenação, cada ação adicionada ao plano é necessária para resolver a meta final. Além disso, se optar por qualquer outra ordenação de recursos que necessita satisfazer novamente metas previamente resolvidas, serão apenas geradas permutações do conjunto de ações produzidas pela ordem. Isto porque, se uma meta for satisfeita novamente, deve ser porque alguma ação utilizada para resolver essa meta foi “usada” pelas condições de outra meta. Assim, o plano encontrado pelo MEA tem o conjunto mínimo de ações, que consome o conjunto mínimo de recursos necessários para atingir a meta. Como o MEA a cada passo acrescenta pelo menos uma ação útil ao plano, sua duração é limitada pelo número mínimo de ações para satisfação da meta.

Se o gráfico de dependência de recursos contém ciclos, é possível que o MEA fique preso em um *loop* infinito para certos estados iniciais. Por exemplo, no *Wargus*, *collect-gold* requer uma *townhall* e um *peasant*, enquanto as ações *build-townhall* e *build-peasant* consomem uma certa quantidade de ouro. A presença destes ciclos significa que há uma possibilidade que não exista um plano para atingir uma determinada meta em alguns casos. No entanto, é possível facilmente estender o algoritmo para detectar tais casos se eles acontecerem. Além disso, como descrito acima, se o estado do jogo inicial contém um *peasant* e uma *townhall*, podemos garantir que há sempre um plano, não importando qual seja a meta. A seguir é apresentado um exemplo de planejamento seqüencial.

Exemplo

Utilizando o Algoritmo 4, tem-se como objetivo encontrar um plano seqüencial para o estado inicial (1 *Townhall*, 3 *Peasant*, 400 *Gold*, 200 *Wood*) e meta (1 *Townhall*, 4 *Peasant*). Nas Figuras 2.25, 2.26, 2.27, 2.28, 2.29 e 2.30, são mostrados os passos da solução deste problema. Cada meta satisfeita é tachada com um traço.

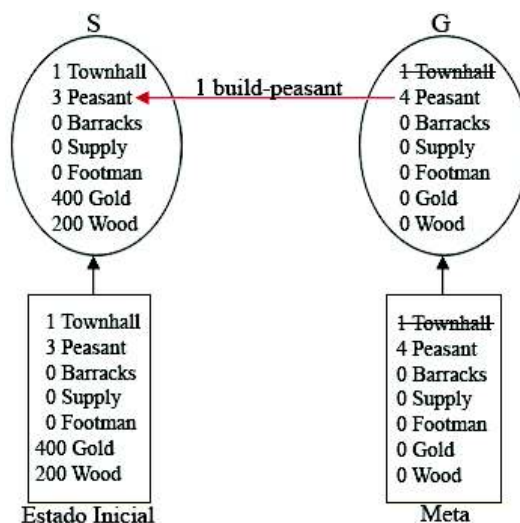


Figura 2.25: Primeiro passo do problema de planejamento seqüencial.

Na Figura 2.25 é apresentado o primeiro passo da solução deste problema. 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*). Para isso, é preciso adicionar uma ação *build-peasant* ao plano. Após incluir esta ação é realizada uma chamada recursiva para o Algoritmo 4, 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.

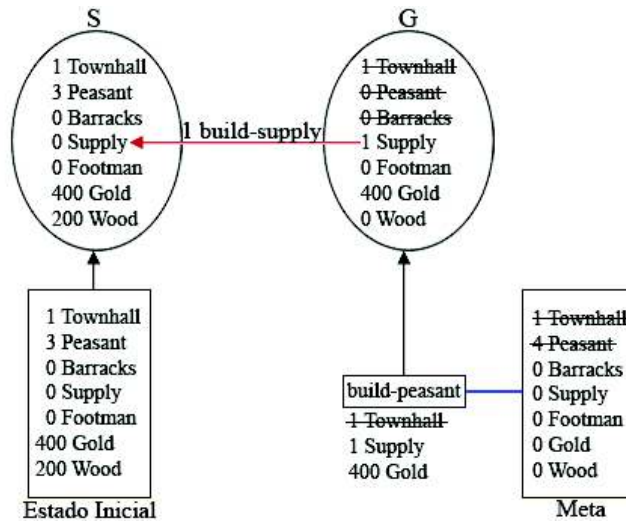


Figura 2.26: Segundo passo do problema de planejamento sequencial.

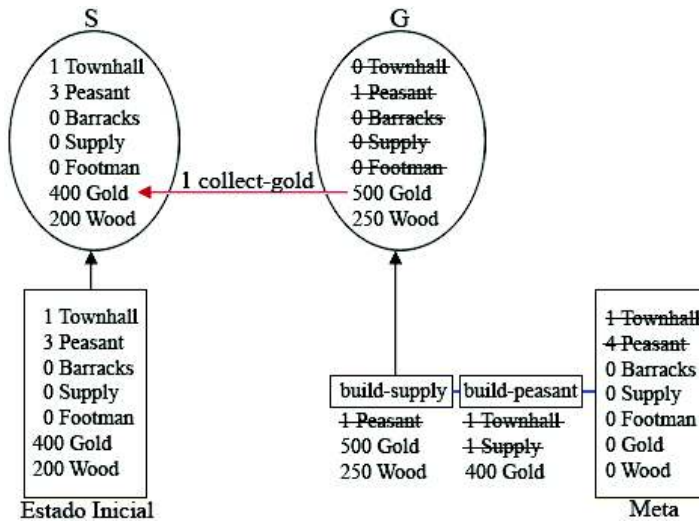


Figura 2.27: Terceiro passo do problema de planejamento sequencial.

A Figura 2.26 mostra o segundo passo do exemplo tratado. O plano encontrado no atual momento é apresentado na parte inferior. Neste caso, consiste apenas na ação *build-peasant*. A meta G corresponde as pré-condições desta ação. Os recursos *1 Supply*

e 400 Gold não são satisfeitos por S . Sendo assim, é determinada a solução da meta 1 *Supply* pela adição da ação *build-supply* no plano. Em seguida, é realizada uma chamada recursiva para satisfação das pré-condições desta nova ação.

Na Figura 2.27 é exibido mais um passo do planejamento. O atual plano é formado sequencialmente pelas ações *build-supply* e *collect-gold*. Novamente, é necessário satisfazer as metas de G . Para isto é incluído ao plano a ação *collect-gold*. Uma nova chamada ao método é realizada para satisfazer as pré-condições de *collect-gold*.

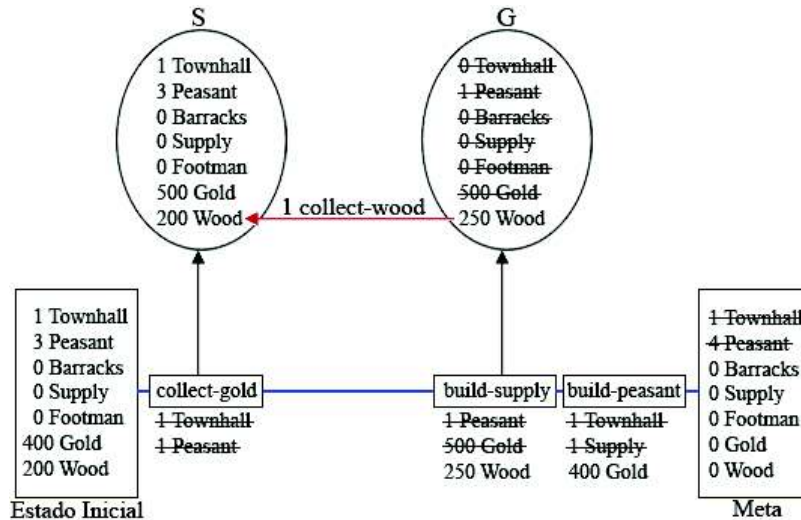


Figura 2.28: Quarto passo do problema de planejamento sequencial.

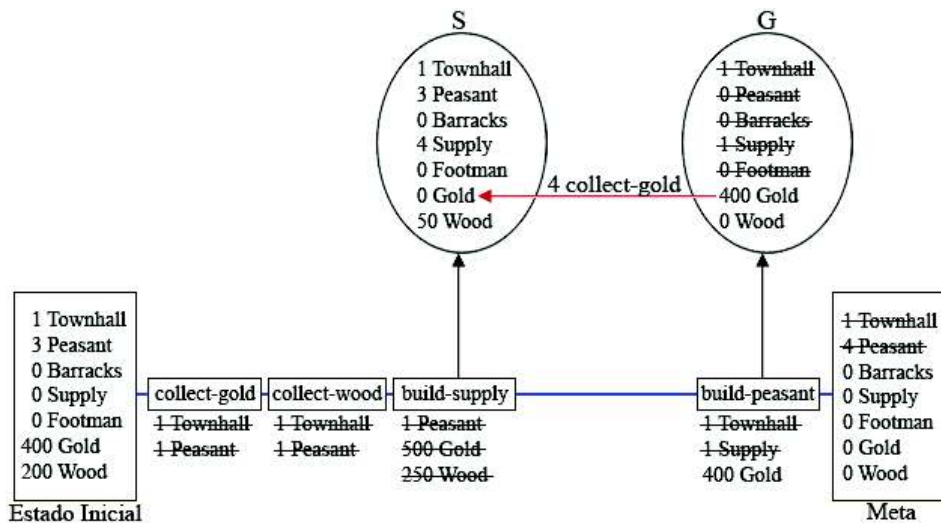


Figura 2.29: Quinto passo do problema de planejamento sequencial.

Após adicionar a ação *collect-gold* no plano, é preciso satisfazer suas pré-condições. Contudo, as metas em G são satisfeitas por S . Deste modo, *collect-gold* é adicionada ao plano sem a necessidade de incluir novas ações para sua satisfação. Posteriormente, é resolvida uma nova meta que estava suspensa no passo da recursão. Nas Figuras 2.28 e 2.29 são apresentados os últimos passos do planejamento.

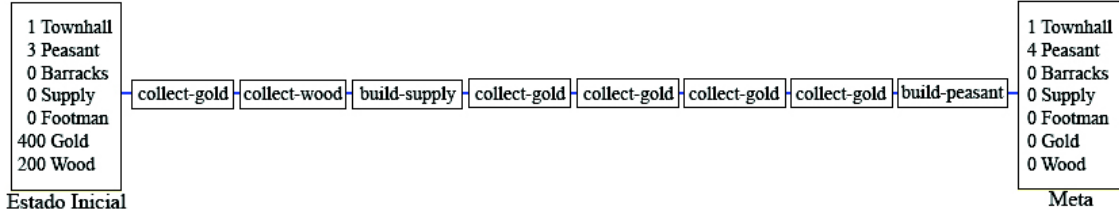


Figura 2.30: Plano sequencial obtido no exemplo tratado.

Ao final temos a solução deste problema exibida na Figura 2.30. O plano sequencial consiste, respectivamente, nas ações *collect-gold*, *collect-wood*, *build-supply*, *collect-gold*, *collect-gold*, *collect-gold*, *collect-gold* e *build-peasant*. O *makespan* deste plano é igual a 4965.

2.4.2 Escalonamento

Para estimar com precisão a utilidade de quaisquer recursos renováveis é preciso reprogramar as ações para permitir a concorrência e diminuir o *makespan* do plano. Fazemos isso utilizando um processo de escalonamento que percorre a seqüência ordenada de ações. Para cada ação A_i , o processo move o tempo de início de A_i 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 , e $R^-(t_i^s)$ é o estado de recursos do jogo antes que os efeitos sejam adicionados. Obviamente, as pré-condições de A_i estão satisfeitas por $R^+(t_i^s)$. Se eles também estão satisfeitos por $R^-(t_i^s)$, isto significa que a satisfação das condições de A_i não são devido a qualquer uma das ações que terminam no tempo t^s , sendo possível mover a ação A_i para começar mais cedo em t^s , a época anterior de decisão (tempo em que uma ação começa ou termina). Isto é repetido até que as condições prévias de A_i sejam satisfeitas por $R^+(t^s)$, mas não $R^-(t^s)$, ou seja, a satisfação das condições de A_i é devido às ações que terminam com tempo t^s . O plano agora é remarcado como a ação A_i que começa no tempo t^s , e podemos continuar a tentar escalonar as próximas ações do plano sequencial. O pseudo-código para este procedimento é dado no Algoritmo 5.

Algorithm 5 *Schedule(Plano)*

```

1: for  $i \leftarrow 1, \dots, k$  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
```

É preciso garantir que ao escalonar uma ação, cada ação entre o novo tempo de início e o velho tempo de término continua sendo executada. Quando cada ação é processada, sempre podemos programá-la antes de uma ação anterior se elas não consomem ou emprestam o mesmo recurso. Considerando um par de ações A e B , sendo A antes de B , em um plano válido com os recursos *consume* e *borrow* em R e assumindo que está prestes a escalonar B . Primeiro, se A e B são adjacentes, o estado antes de A deve ter os recursos suficientes em R para A e B serem executadas. Isto significa que, naquele estado, A e B podem ser executadas simultaneamente, ou se possível, B pode ser escalonada antes de A . Por outro lado, 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. É apresentado a seguir um exemplo de escalonamento utilizando o Algoritmo 5.

Exemplo

Este exemplo consiste no escalonamento do plano seqüencial da Figura 2.30. Este procedimento reduz o *makespan* do plano colocando algumas ações em paralelo. Nas Figuras 2.31, 2.32, 2.33, 2.34, 2.35, 2.36, 2.37 e 2.38, são apresentados alguns passos do escalonamento. Ao final, a Figura 2.39 mostra o plano obtido após o escalonamento.



Figura 2.31: Tentativa de escalonar a ação *collect-gold* no tempo 0.

De modo seqüencial, cada ação do plano é escalonada. O tempo de uma ação é alterado para o menor tempo em que suas pré-condições foram satisfeitas. O primeiro passo do escalonamento, exibido na Figura 2.31, 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. A Figura 2.32 mostra esta ação após ser escalonada.

No passo seguinte, é realizado o escalonamento da ação *collect-wood*. Inicialmente, esta ação é colocado no tempo 510 (Figura 2.33). Neste tempo os recursos disponíveis

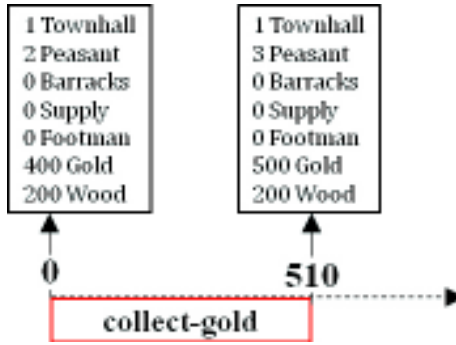


Figura 2.32: A ação *collect-gold* é escalonada no tempo 0.

satisfazem suas pré-condições. Esta ação é posicionada no tempo anterior, igual a 0 (Figura 2.34). Neste caso, *collect-wood* pode ser mantida no tempo 0. Sendo assim, essa ação fica em paralelo com *collect-gold* (Figura 2.35).



Figura 2.33: Tentativa de escalonar a ação *collect-wood* no tempo 510.

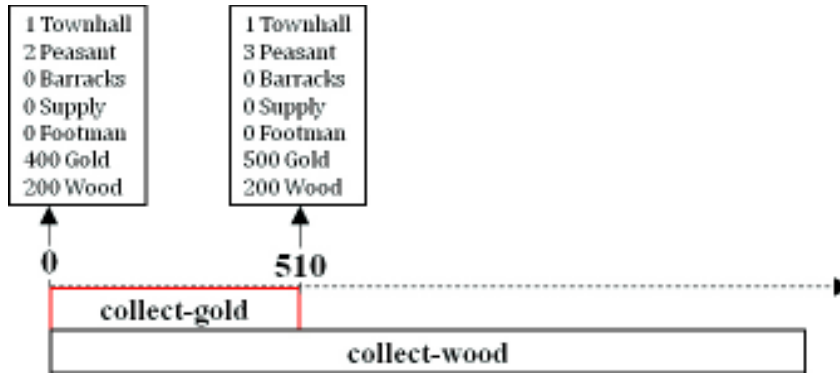


Figura 2.34: Tentativa de escalonar a ação *collect-wood* no tempo 0.

Build-supply é a próxima ação a ser escalonada. As pré-condições desta ação são verificadas se são satisfeitas no tempo 1570 (Figura 2.36). 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 2.37). 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 2.38).

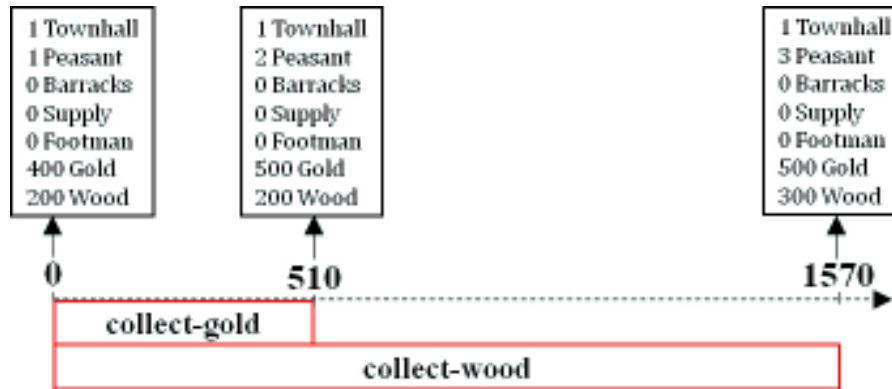


Figura 2.35: A ação *collect-wood* é escalonada no tempo 0.

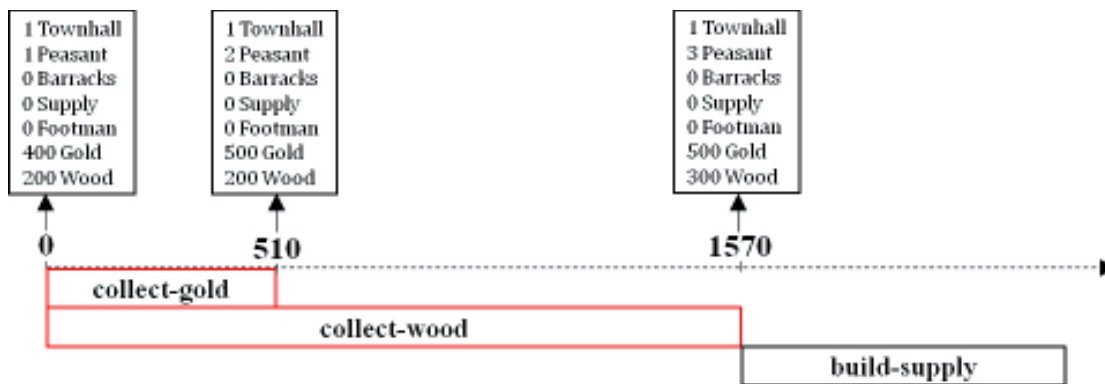


Figura 2.36: Tentativa de escalonar a ação *build-supply* no tempo 1570.

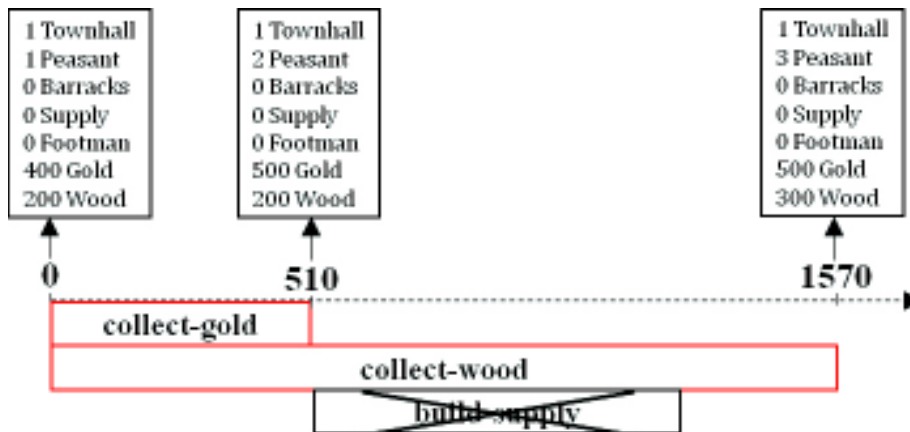


Figura 2.37: Tentativa de escalonar a ação *build-supply* no tempo 510.

Todas as ações restantes são escalonadas utilizando este mesmo procedimento. O resultado final é apresentado na Figura 2.39. É interessante observar que o *makespan* do plano sequencial e do plano escalonado são, respectivamente, 4915 e 2415. Isso demonstra que um melhor plano pode ser encontrado ao executar o procedimento de escalonamento.

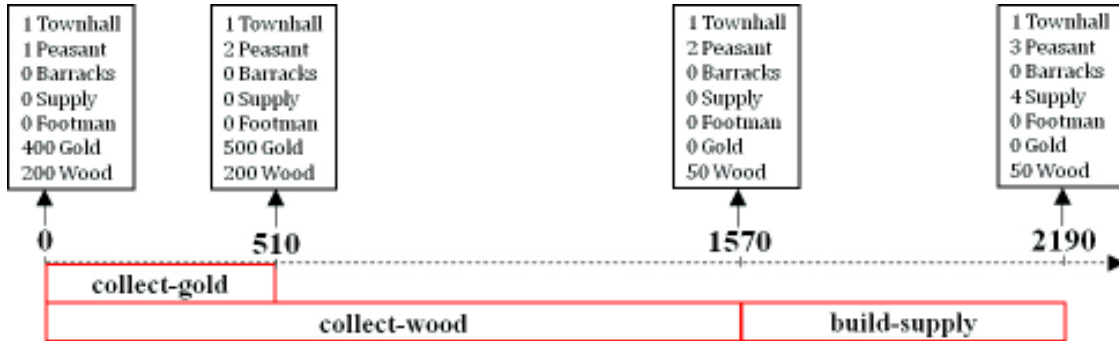


Figura 2.38: A ação *build-supply* é escalonada no tempo 1570.

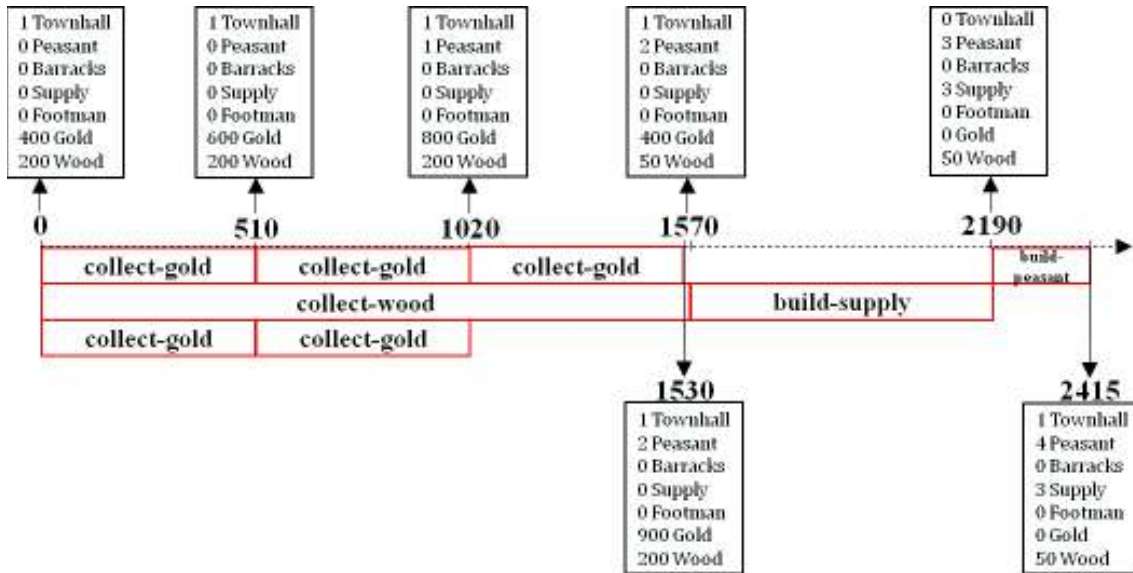


Figura 2.39: Resultado do escalonamento do plano da Figura 2.30.

2.4.3 Produção de Recursos Adicionais

Objetivando minimizar o *makespan* considera-se a inclusão de metas intermediárias de geração de recursos. Uma meta intermediária corresponde a uma meta de produzir um ou mais recursos de um determinado tipo. Para cada meta intermediária o planejador emprega o MEA seguido pelo processo de *Schedule* para gerar um plano. Considerando este plano são aplicados os mesmos procedimentos até atingir a meta global. Para selecionar uma ação a ser executada, dentre os planos gerados, o planejador escolhe o plano com o menor *makespan*. O plano gerado pelo planejador geralmente não é totalmente executado quando o sistema realiza o planejamento utilizando o estado atual do jogo. No entanto, a construção de tais planos são valiosos porque ajudam na seleção da ação na fase atual. Para este cálculo de metas intermediárias é apresentado o Algoritmo 6.

A justificativa para este processo de busca delimitado por metas intermediárias é descrito a seguir. É claro que é possível encontrar um plano que tem um número mínimo

Algorithm 6 OnlinePlanner()

```

1: for para todos os predeterminados ciclos do jogo do
2:    $t \leftarrow$  tempo atual
3:    $S \leftarrow$  estado atual do jogo
4:   if existe alguma ação disponível que pode ser executada no tempo atual then
5:      $Plan \leftarrow Schedule(MEA(S, G))$ 
6:   end if
7:    $(G_1, \dots, G_n) \leftarrow$  uma lista de metas intermediárias que gera mais recursos renováveis.
8:   for all  $i = 1, \dots, n$  do
9:      $P_0 \leftarrow MEA(S, G_i)$ 
10:     $S' \leftarrow$  estado projetado após executar  $P_0$  a partir de  $S$ 
11:     $P_1 \leftarrow MEA(S', G)$ 
12:     $Plan_i \leftarrow Schedule(concatenar(P_0, P_1))$ 
13:    if  $makespan$  de  $Plan_i < makespan$  de  $Plan$  then
14:       $Plan \leftarrow Plan_i$ 
15:    end if
16:  end for
17: end for
18: for all  $(A_j, t_j^s, t_j^e) \in Plan$  onde  $t_j^s = t$  do
19:   executar  $A_j$ 
20: end for

```

de ações e cria o montante mínimo de recursos renováveis, como os *peasant*. No entanto, a criação adicional de recursos renováveis pode diminuir o *makespan* de um plano (mesmo que este novo plano tenha mais ações). Se o tempo despendido na criação desses recursos é compensado pelo tempo poupado pelas ações em paralelo permitidas pelos recursos adicionais renováveis. Contudo, os planos podem se tornar demasiadamente complexos se existe uma busca ilimitada sobre todas as metas intermediárias possíveis. Para contornar este problema, é necessário encontrar explicitamente um plano que permita alcançar a meta intermediária de criar alguma quantidade fixa adicional de recursos renováveis, em seguida, encontrar um plano que permita alcançar a meta global. Os dois planos são então combinados em um único plano, sendo verificado se o novo plano tem um *makespan* menor do que o plano original. Se assim for, é considerado como solução o novo plano, que produz os recursos adicionais renováveis.

Para cada “ciclo” (unidade de tempo do *Wargus*) do jogo, o cliente verifica se existem algumas ações disponíveis que podem ser executadas. Se assim for, ele chama o planejador para encontrar os planos que preenchem a meta a partir do estado atual. Um plano terá como meta satisfazer a meta global de recursos, sem metas intermediárias, enquanto os outros terão por objetivo satisfazer cada uma das metas intermediárias que explicitamente criam recursos renováveis adicionais. O plano com o menor *makespan* é escolhido. Para realizar essas tarefas são utilizados os dois procedimentos apresentado anteriormente: MEA e *Schedule*.

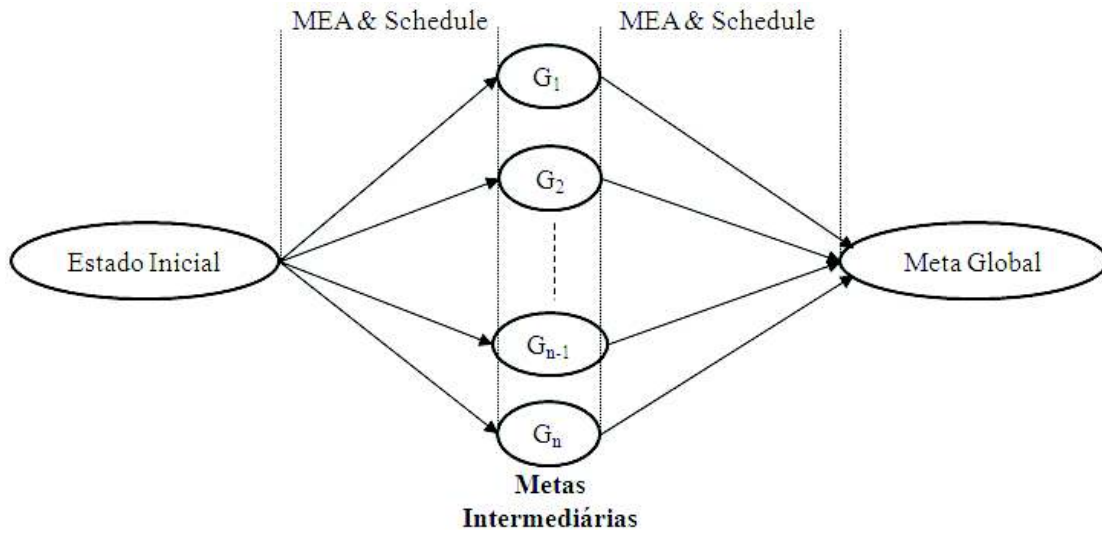


Figura 2.40: Produção de recursos adicionais através de metas intermediárias.

A Figura 2.40 apresenta uma representação geral da busca de recursos adicionais usados para atingir a meta global mais rapidamente. Existem n metas intermediárias, representadas por G_i , onde $1 \leq i \leq n$. Essas metas intermediárias possibilitam encontrar planos com menor *makespan*. Os planos são criados utilizando os algoritmos MEA e *Schedule*. Dentre os planos gerados ao considerar as metas intermediárias é escolhido o que possui o menor *makespan*.

Capítulo 3

Arquitetura de Planejamento

Para gerar um plano, onde a ordem de todas as ações são especificadas completamente, inicialmente é proposto um planejamento de ordem parcial. O planejamento de ordem parcial especifica todas as ações a serem executadas, mas não determina uma ordem específica para a execução das mesmas. As ações são escolhidas conforme a meta em relação ao estado atual. Segundo [Russell e Norvig 1995], 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”.

Como descrito no Capítulo 2 e citado por [Chan et al. 2007b, Chan et al. 2007a], o planejamento para jogos do tipo RTS possui algumas propriedades específicas. Apesar da possibilidade de utilizar outros planejadores, um simples planejamento usando MEA é suficiente para essa tarefa.

A motivação da arquitetura de planejamento deste trabalho tem como base a Figura 3.1. Neste problema o estado inicial é $(1 \text{ Townhall}, 1 \text{ Peasant}, 2 \text{ Supply}, 600 \text{ Gold})$ e a meta é $(1 \text{ Townhall}, 3 \text{ Peasant})$. Temos em A um plano sequencial gerado pelo planejador sequencial de [Chan et al. 2007b]. Este plano é dado como entrada para o algoritmo de escalonamento deste mesmo autor, tendo como resultado o plano B . No plano A existe o problema que as ações *collect-gold* não aproveitam 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. Sendo assim, a arquitetura de planejamento do trabalho tem o objetivo de alterar o plano sequencial dado como entrada para o escalonamento. Isto possibilita aproveitar melhor os recursos que são criados durante o planejamento.

O plano C mostra o plano sequencial que se pretende obter neste trabalho. Como a primeira ação *build-peasant* não necessita dos recursos produzidos pelas ações *collect-gold*, é possível alocar tal ação no início do plano. Para encontrar estes planos, é proposto o uso de um planejamento de ordem parcial em conjunto de um procedimento de estabelecimento de prioridade. Isso permite priorizar as ações mais importantes que podem disponibilizar recursos durante o escalonamento. Quando o plano C é dado como en-

trada para o escalonamento de [Chan et al. 2007b], 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 .

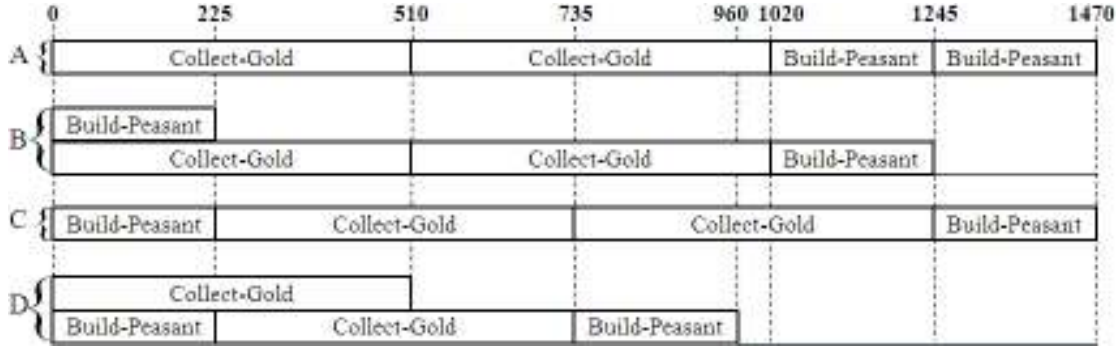


Figura 3.1: Planos usados como motivação da arquitetura de planejamento.

A seguir é descrito o planejamento de ordem parcial proposto neste trabalho. O plano criado é composto por atividades. Uma atividade consiste em uma ou mais ações de um determinado tipo. Este método utiliza a seleção de operadores (ações) para resolver metas. Cada operador compõe um conjunto de pré-condições e efeitos, sendo responsável por decrementar a diferença entre a atividade inicial e a meta. São criadas novas atividades utilizando estes operadores. Para cada nova atividade é aplicada uma recursão para satisfazer suas pré-condições. Este procedimento ocorre até que todas as metas sejam satisfeitas.

O Algoritmo 7 especifica o planejamento de ordem parcial proposto **MeaPop**($S_{início}$, G_{meta}). Os parâmetros de entrada são o estado inicial $S_{início}$ e o estado meta G_{meta} . Estes parâmetros são compostos apenas por recursos. Uma variável global R guarda os recursos que são produzidos por cada atividade. Isso permite que os recursos sejam aproveitados, evitando que atividades redundantes sejam criadas. As atividades S e G são criadas a partir dos estados $S_{início}$ e G_{meta} . A atividade S não está associada a ações e não apresenta pré-condições. Contudo, S tem como efeito a produção dos recursos presentes em $S_{início}$. A atividade G não está associada a ações e efeito. Entretanto, S tem como pré-condição os recursos presentes em G_{meta} . No **MeaPop** um plano é gerado pela chamada da função **PlanejamentoOrdemParcial**(S , G) tendo como parâmetros as atividades S e G . O plano é composto pelas atividades e *links causais* criados pela função **PlanejamentoOrdemParcial**.

O método **PlanejamentoOrdemParcial**, apresentado no Algoritmo 8, é responsável por estabelecer quais atividades e *links causais* devem ser criados. Inicialmente, são verificados se todos os recursos da meta G são satisfeitos por S . Se forem satisfeitos é estabelecido um *link causal* entre S e G . Isso significa que não é preciso criar nenhuma atividade e não é necessário utilizar as atividades presentes em R para satisfazer G . Os

Algorithm 7 MeaPop(S_{inicio}, G_{meta})

1. Variável global $R \leftarrow \emptyset$, instanciada inicialmente sem elementos.
 2. Criar a atividade S sem ações e sem pré-condições. Entretanto, S possui o efeito de produzir os recursos de S_{inicio} .
 3. Criar a atividade G sem ações e sem efeitos. Entretanto, S possui como pré-condições os recursos de G_{meta} .
 4. Executar **PlanejamentoOrdemParcial**(S, G). Este procedimento cria o plano de ordem parcial.
-

recursos do tipo *consume* (*supply*, *gold* e *wood*.) são alocados para a satisfação de G . Alocar um recurso significa que nenhum outra ação pode usar tal recurso. O recurso é alocado apenas se for necessário e esteja disponível.

Algorithm 8 PlanejamentoOrdemParcial(S, G)

1. Se todos os recursos da meta G são satisfeitos por S é criado o *link causal* $S \xrightarrow{recursos} G$. Os recursos do tipo *produce* da atividade S são alocadas para satisfazer G . Apenas os recursos necessários e disponíveis são alocados.
 2. Caso contrário. Para cada recurso meta $g_i \in G$, faça:
 - (a) $s_i \leftarrow$ recurso g_i presente em S .
 - (b) $r_i \leftarrow$ recurso g_i presente em R .
 - (c) Se g_i é do tipo *consume*, é alocada a quantidade de recurso s_i necessária e disponível para satisfazer g_i .
 - (d) Se g_i é satisfeito pelo recurso s_i , retornar ao passo 2.
 - (e) Se $r_i > 0$, é criado o *link causal* $R \xrightarrow{r_i} G$ com a(s) atividade(s) em R que produz(em) o recurso r_i . Essa relação é estabelecida com o menor número de atividades possível. Caso r_i seja do tipo *consume*, é alocado o recurso r_i necessário para satisfazer g_i . É alocado apenas o recurso necessário e disponível.
 - (f) Se g_i não é satisfeito pelos recursos $s_i + r_i$, executar **CriarAtividade**(s_i, r_i, g_i, S, G).
-

Se G não é totalmente satisfeito por S , é iniciada uma busca com o objetivo de satisfazer G . Para isso, novas atividades são criadas, *links causais* são estabelecidos e recursos são alocados. A ordem que os recursos de G são verificados é: *townhall*, *peasant*, *barracks*, *supply*, *footman*, *gold* e *wood*. Essa ordem possibilita que ações mais importantes sejam satisfeitas primeiro [Chan et al. 2007b].

As variáveis s_i , r_i e g_i , correspondem respectivamente aos recursos presentes em S , R e G . O índice i indica que o mesmo recurso é atribuído a cada variável. Ou seja, se a meta é o recurso *gold*, as variáveis s_i , r_i e g_i , referem-se a *gold*.

Se o recurso g_i é do tipo *consume*, são alocados os recursos necessários e disponíveis para satisfazer g_i . Caso s_i satisfaça g_i , a busca passa para o próximo recurso. Se s_i não

satisfazer g_i , o procedimento continua. Se existir recurso disponível em r_i , tal recurso é usado para satisfazer a meta, sendo alocada a quantidade necessária e disponível para satisfazer a meta. Novamente, se r_i é do tipo *consume* a quantidade necessária e disponível do recurso r_i é alocado. Caso os recursos de S mais R não sejam suficientes para satisfazer a meta, uma nova atividade é criada através da função **CriarAtividade**(s_i, r_r, g_i, S, G).

O Algoritmo 9, **CriarAtividade**(s_i, r_r, g_i, S, G), é responsável por criar atividades. Essas atividades produzem recursos, tendo como objetivo satisfazer g_i . Primeiro é estabelecida qual ação A_i produz o recurso g_i . Em seguida, é calculada a quantidade de ações necessárias para satisfazer a diferença de recursos ($g_i - (s_i + r_i)$). É verificado o tipo de recurso, se for *consume* é criada apenas 1 atividade com k ações A_i , caso contrário, são criadas k atividades com 1 ação A_i cada. O quanto antes alguns recursos são disponibilizados no escalonamento melhor pode ser o plano, por isso algumas atividades são criadas com apenas uma ação A_i . Uma nova chamada ao método **PlanejamentoOrdemParcial** é realizada, tendo o objetivo de satisfazer as pré-condições da(s) nova(s) atividade(s). Ao final todo recurso produzido e disponível de uma atividade com todas as pré-condições satisfeitas é adicionado a variável R (Passos 4(d) e 5(d) do Algoritmo **CriarAtividade**).

Algorithm 9 CriarAtividade(s_i, r_i, g_i, S, G)

1. $A_i \leftarrow$ ação que produz o recurso g_i .
 2. $\alpha \leftarrow$ quantidade de recurso produzido por A_i .
 3. $k \leftarrow \text{ceil}((g_i - (s_i + r_i))/\alpha)$. Quantidade de ações necessárias para satisfazer a meta.
 4. Se o recurso s_i é do tipo *consume*.
 - (a) Criar uma atividade A com k ações A_i . Na atividade A são estabelecidas as pré-condições para sua execução. Ou seja, as metas que devem ser satisfeitas para executar A .
 - (b) Criar o *link causal* $A \xrightarrow{\text{produce}} G$.
 - (c) Executar **PlanejamentoOrdemParcial**(S, A).
 - (d) Os recursos produzidos pela atividade A são adicionados a variável R . Alocar os recursos produzidos necessários para satisfazer a meta, exceto a quantidade excedente.
 5. Caso contrário, repetir k vezes os seguintes passos:
 - (a) Criar uma atividade A com 1 ação A_i . Na atividade A são estabelecidas as pré-condições para sua execução. Ou seja, as metas que devem ser satisfeitas para executar A .
 - (b) Criar o *link causal* $A \xrightarrow{\text{produce}} G$.
 - (c) Executar **PlanejamentoOrdemParcial**(S, A).
 - (d) Os recursos produzidos pela atividade A são adicionados a variável R .
-

Nas Figuras 3.2 e 3.3 é exibido um exemplo de aplicação do Algoritmo 7.

Exemplo

Neste exemplo, o estado inicial é $(1 \text{ Townhall}, 1 \text{ Peasant}, 50 \text{ Gold}, 50 \text{ Wood})$ e a meta é $(1 \text{ Townhall}, 2 \text{ Peasant}, 1 \text{ Footman})$. No primeiro momento a variável R não possui nenhum elemento, estando vazia. A cada passo as atividades são criadas e os recursos disponíveis são adicionados a R . São apresentados 18 passos, onde cada um deles corresponde a criação de uma atividade ou um relacionamento de precedência. Na parte superior das atividades é descrito o número da atividade e o tempo de duração. O número da atividade é um parâmetro que representa a atividade e o tempo é usado no escalonamento, sendo explicado com detalhes no decorrer do trabalho. As metas que devem ser satisfeitas são exibidas na parte inferior de cada atividade. Cada meta satisfeita ou recurso alocado é sublinhado.

Inicialmente, no passo A , é verificado cada recurso. A meta 1 Townhall já é satisfeita pelo estado inicial, sendo desconsiderada. A seguir é criada uma atividade de número 1 com uma ação *build-peasant* para satisfazer a meta 2 Peasant , tendo tempo igual a 225. As pré-condições desta atividade são 1 Townhall , 1 Supply e 400 Gold .

No passo B é criada uma nova atividade de número 2 para satisfazer a meta 1 Supply . Essa atividade possui um tempo de 845 e meta 1 Peasant , 500 Gold e 250 Wood . O recurso 1 Peasant já existe no estado inicial, sendo necessário satisfazer 500 Gold . Entretanto, já existe 50 Gold disponível, essa quantidade é destinada a satisfação parcial desta meta. Uma nova atividade com 5 ações *collect-gold* é criada, como é apresentado em C . Para a atividade 3 os recursos do estado inicial satisfazem suas metas (D). Porém, os recursos excedentes da atividade 3 são adicionados em R para que novas atividades possam as utilizar para satisfazer suas metas. Este procedimento ocorre até que todas as metas sejam satisfeitas, como pode ser observado nos passos seguintes das Figuras 3.2 e 3.3.

O resultado final deste exemplo é observado na Figura 3.4. Basicamente, este plano é um “grafo”, onde cada vértice corresponde a uma quantidade de ações e as arestas definem as relações de precedência, ou seja, uma ação pode ser executada somente após todas as precedências. Cada atividade possui alguns parâmetros na parte superior, representando respectivamente, o número da atividade, a maior duração para completar essa atividade sem considerar as restrições de recursos e a prioridade. Uma observação importante é que os recursos de uma atividade são adicionados em R apenas após a satisfação de todas as suas metas. No Capítulo 4, é tratado com detalhes o cálculo de prioridade das atividades.

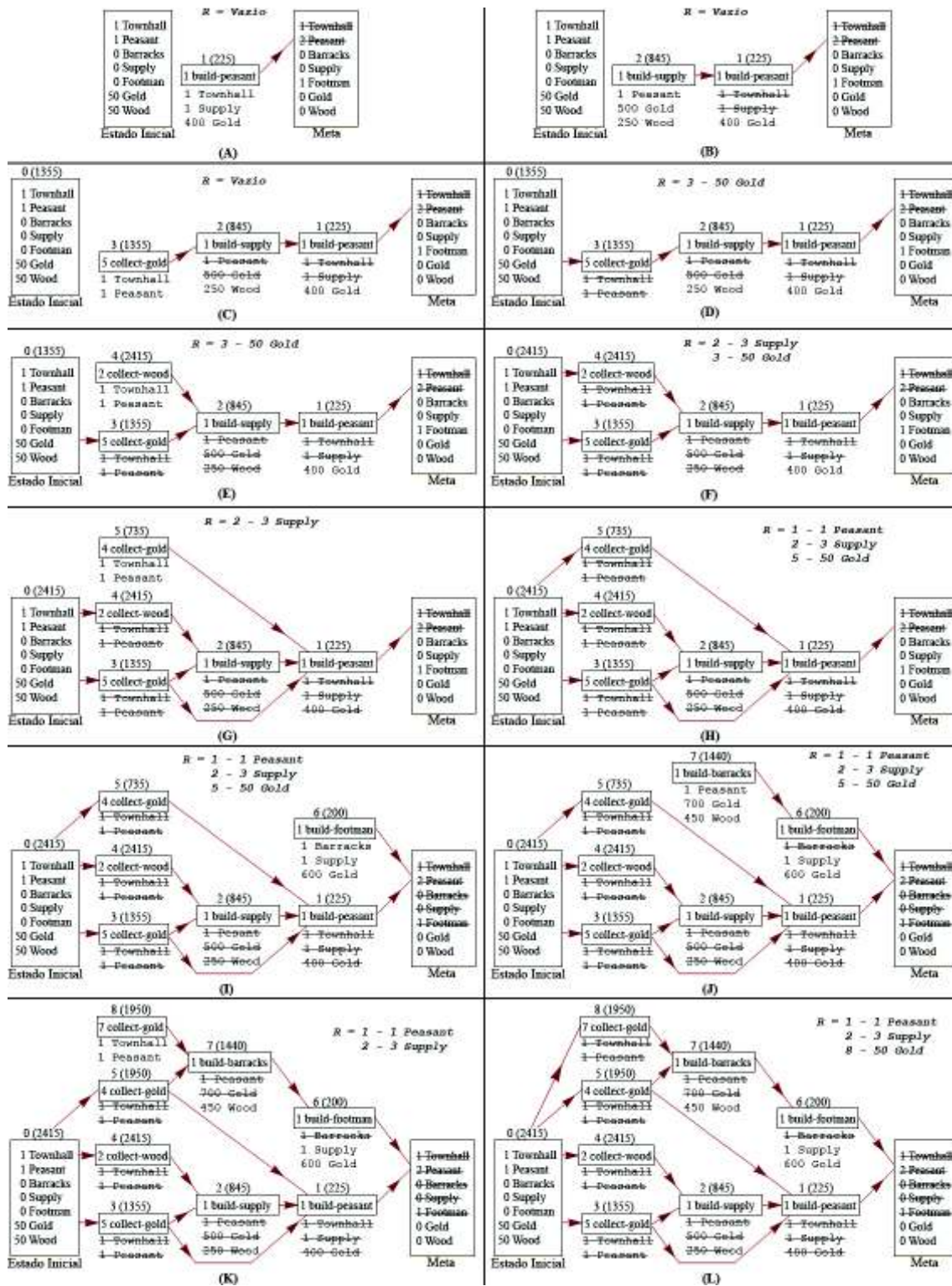


Figura 3.2: Exemplo de execução do Algoritmo 7 em um problema de planejamento.

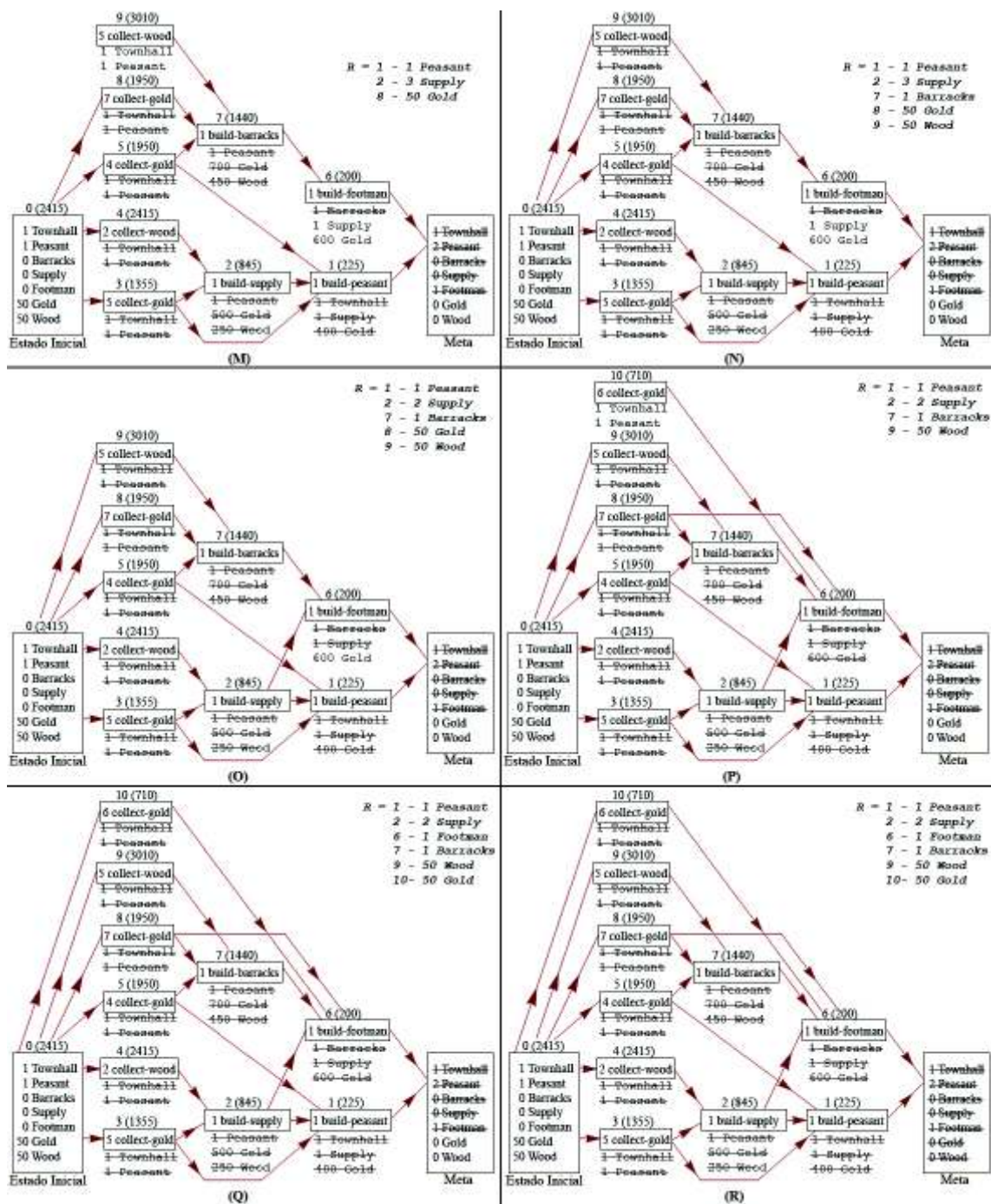


Figura 3.3: Exemplo de execução do Algoritmo 7 em um problema de planejamento (continuação).

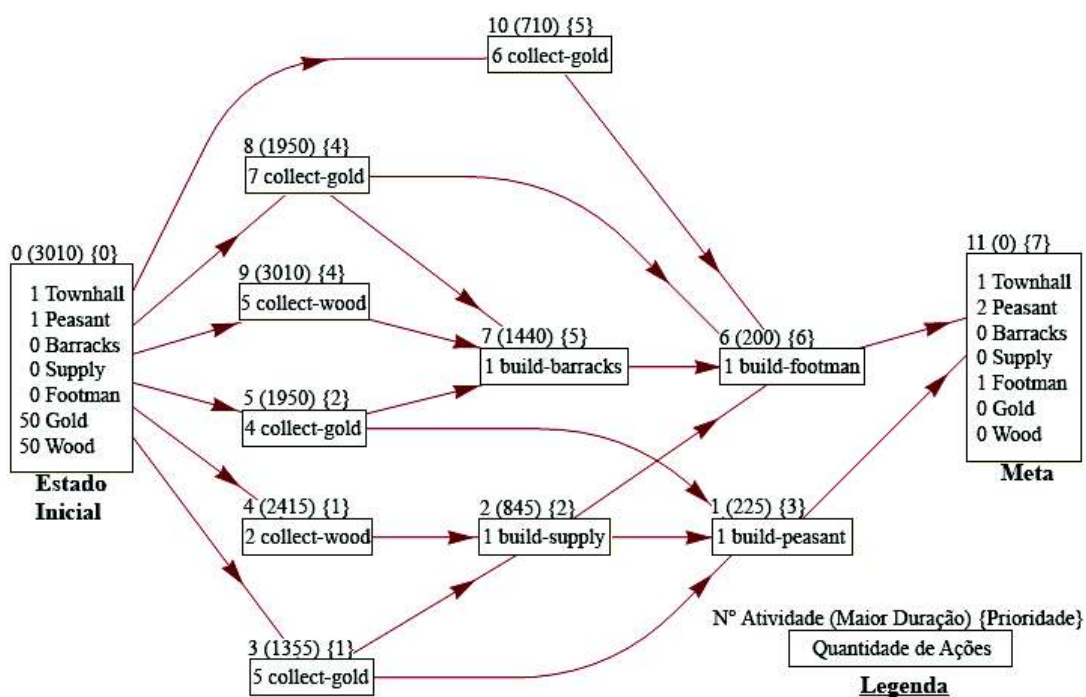


Figura 3.4: Plano de Ordem Parcial criado com o Algoritmo 7 para o exemplo tratado.

Capítulo 4

Estabelecimento de Prioridade para Atividades

Objetivando tornar mais eficiente o processo de escalonamento observou-se que identificar ações mais relevantes era um fator importante. Neste sentido prioridades foram estabelecidas para as atividades.

O mecanismo usado para definir prioridades baseia-se num procedimento similar a uma busca em profundidade [Cormen et al. 2001]. Este método é aplicado sobre as atividades que precedem diretamente a meta. O procedimento obtém um valor numérico que identifica a prioridade de uma atividade. Um valor numérico baixo associado a uma atividade significa que a mesma possui uma prioridade alta e tem preferência sobre as outras atividades que estão associadas a um valor numérico superior.

Os planos de ordem parcial tratados como entrada para o algoritmo de estabelecimento de prioridade para atividades são considerados grafos. Cada precedência corresponde a uma aresta que conecta duas atividades. Supondo duas atividades u e v , se u precede v , então, temos a seguinte aresta $\{v, u\}$. O grafo usado neste procedimento é o “inverso” do plano gerado do Capítulo 3. Na Figura 4.1 é apresentado um exemplo de um grafo com base no plano da Figura 3.4.

Para definir as prioridades são usados os Algoritmos 10 e 11. O Algoritmo 10 é responsável por gerar subgrafos a partir do grafo correspondente ao plano de ordem parcial gerado na Seção 2.3.4. Estes subgrafos são dados como entrada para o Algoritmo 11. Este algoritmo é responsável por estabelecer as prioridades, de modo que as atividades com maior “profundidade” sejam mais importantes. A “profundidade” corresponde a um valor que indica o quanto mais profundo o elemento está na busca.

Inicialmente, cada atividade que precede a meta corresponde a um subgrafo. Cada subgrafo é responsável por estabelecer a prioridade de parte das atividades do plano. No contexto deste trabalho temos algumas ações que são mais importantes. O quanto antes estas ações são executadas melhor é o aproveitamento dos recursos. Os subgrafos são geradores considerando a mesma ordem de importância de recursos do Capítulo 3,

Algorithm 10 Prioridade(G)

```

1: for all vértice  $u \in V[G]$  do
2:    $cor[u] \leftarrow \text{BRANCO}$ 
3:    $w[u] \leftarrow 0$ 
4: end for
5:  $p \leftarrow 1$ 
6: for all vértice  $u \in Adj[Meta]$  do
7:    $G' \leftarrow$  subgrafo a partir de  $u$  (exceto os vértices onde  $cor = \text{PRETO}$ ).
8:   PrioridadeVisita( $G', u, p$ )
9:    $p' \leftarrow$  maior prioridade em  $G'$  mais 1.
10:  for all vértice  $v \in V[G']$  do
11:     $cor[v] \leftarrow \text{PRETO}$ 
12:     $w[v] \leftarrow p + p' - w[v]$ 
13:  end for
14:   $p \leftarrow p'$ 
15: end for
16:  $w[meta] \leftarrow p$ 

```

Algorithm 11 PrioridadeVisita(G, u, p)

```

1:  $w[u] \leftarrow p$ 
2: for all vértice  $v \leftarrow Adj[u]$  do
3:   if  $w[v] \leq w[u]$  then
4:     PrioridadeVisita( $G, v, w[u] + 1$ )
5:   end if
6: end for

```

Inicialmente todo vértice possui cor branca e w igual a 0 (linhas 1 à 4). A profundidade inicial p é igual a 1 (linha 5). Cada vértice u adjacente a meta corresponde a um subgrafo G' (o estado inicial e a meta não fazem parte dos subgrafos). Este subgrafo corresponde aos vértices com $cor = \text{BRANCO}$ e que são alcançáveis a partir de u . As arestas são todas aquelas que conectam estes vértices. Para essa tarefa pode ser usado um simples algoritmo de busca em profundidade ou busca em largura. Após definir G' é aplicado o Algoritmo 11 para definir a prioridade de cada vértice $v \in V[G']$ (linha 8). A seguir é determinado o maior valor de prioridade em G' mais 1. Como o grafo é um retrato do plano de ordem parcial, com arestas no sentido contrário, o valor de profundidade deve ser alterado. Isso é feito para todos os vértices de G' (linha 12). Cada vértice em G' é marcado com a cor PRETO (linha 11), já que acabou de ser analisado. Finalmente é armazenada a profundidade atual em p (linha 14). Por último o valor de p é associado a meta, já que é a última atividade.

O Algoritmo 11 encontra a maior profundidade dos vértices em um grafo. Este valor de profundidade representa exatamente a maior distância entre o primeiro vértice e todos os outros considerando uma distância de 1 para cada aresta. O Lema 4.1 e a Prova 4 demonstram a corretude do Algoritmo 11. Após a prova de corretude é descrito com detalhes a análise de complexidade deste procedimento.

Lema 4.1. *Se o Algoritmo 11 é aplicado em um grafo $G = (V, E)$ orientado, acíclico e vértice inicial $u \in V$, tal que:*

- *Inicialmente $\forall s \in V, w[s] = 0$ e $w[u] = 0$.*
- *A distância entre os vértice $s \in V$ e $t \in V$ é de 1 se existe a aresta $\{s, t\} \in E$.*
- *Existe pelo menos um caminho de u até todos os vértices $s \in V$.*

Então, é encontrada a maior profundidade de todos os vértices.

Prova. Não existem ciclos e as arestas possuem custo constante igual a 1. A prova é feita por contradição para o grafo $G = (V, E)$ após a aplicação do Algoritmo 11 supondo que: $(\exists(s, t))$ tal que $((s \in V) \wedge (t \in V) \wedge (\{s, t\} \in E) \wedge (w[t] \leq w[s]))$. Ou seja, existe uma aresta $\{s, t\}$ que pertence ao conjunto E de modo que a profundidade do vértice adjacente t é menor ou igual ao vértice s (Equação 4.1).

$$w[t] \leq w[s] \quad (4.1)$$

Pela definição não existem ciclos e o vértice t é alcançável diretamente por s , ou seja, t é adjacente ao vértice s . Na linha 3 do Algoritmo 11, se $w[t]$ (vértice adjacente) é menor ou igual a $w[s]$ (vértice “pai”), então, é realizada uma chamada recursiva e consequentemente uma atualização de $w[t]$. Todos os vértices adjacentes ao vértice t sofrem o mesmo processo, de modo recursivo até encontrar a maior profundidade. Após essa atualização, o valor de $w[t]$ é igual a $w[s] + 1$, sendo assim:

$$w[t] > w[s] \quad (4.2)$$

Analisando as Equações 4.1 e 4.2, a contradição ocorre ao assumir que: $(\exists(s, t))$ tal que $((s \in V) \wedge (t \in V) \wedge (\{s, t\} \in E) \wedge (w[t] \leq w[s]))$. Deste modo, provamos que o Algoritmo 11 encontra a solução desejada.

Análise

Analisando o Algoritmo 10, temos a seguinte complexidade. Nas linhas 1 à 4 temos $O(V)$ e as linhas 5 e 16 são $O(1)$. Para o laço de repetição da linha 6 temos no pior caso $O(V^3)$, este valor é devido ao pior caso do Algoritmo 11. Isso ocorre quando o subgrafo possui todos os vértices e arestas de G . Este subgrafo é dado como entrada para o Algoritmo 11, que no pior caso tem complexidade $O(V^3)$. Entretanto, para o problema de planejamento é esperado que seu comportamento seja de $O(V + E)$, pois este valor corresponde ao melhor caso do Algoritmo 11.

Para analisar a complexidade do Algoritmo 11, avaliamos o tempo de execução sobre um grafo $G = (V, E)$, acíclico, com o máximo de arestas possível e um valor de “profundidade” w inicialmente igual a 0 para cada vértice. Analisamos a complexidade utilizando

a análise agregada citada em [Cormen et al. 2001] na Seção 17.1. A análise agregada considera as diferentes operações possuindo o mesmo custo agregado. Neste caso, o mesmo custo é associado para as operações de visitar os vértices e arestas.

Em geral, para o problema de planejamento o grafo G , dado como parâmetro do Algoritmo 11, possui poucos vértices e arestas. Sendo assim, a complexidade deste método durante o planejamento é dada aproximadamente pelo melhor caso. Entretanto, são apresentadas a seguir as análises de pior e melhor caso.

Melhor Caso

O melhor caso do Algoritmo 11 é composto exatamente pela mesma complexidade de uma busca em profundidade. Isso ocorre pois cada vértice e aresta são visitados apenas uma vez, não sendo necessário realizar novas buscas, pois a profundidade final dos vértices é encontrada em apenas um passo. A Figura 4.3 apresenta em (a) o grafo inicial, em (b) temos as primeiras arestas (em negrito) que são visitadas com os valores de profundidade atualizados e finalmente em (c) é exibido o grafo final.

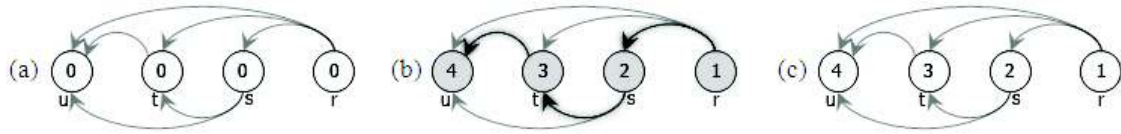


Figura 4.3: Grafo usado na análise do melhor caso do Algoritmo 11.

Segundo [Cormen et al. 2001], a complexidade da busca em profundidade é $O(V + E)$. Sendo assim, o Algoritmo 11 no melhor caso também possui complexidade $O(V + E)$.

Pior Caso

A aplicação do Algoritmo 11 inicia em um vértice de modo que o maior número de vértices e arestas sejam visitados. Para facilitar a análise exata deste procedimento, inicialmente consideramos o exemplo da Figura 4.4. Neste caso são apresentados todos os passos (a, b, c, d, e, f, g, h, i) em que um vértice ou aresta são visitados. O grafo é composto pelos vértices $V = \{r, s, t, u\}$. As arestas estão conectadas de modo que, o primeiro vértice r está ligado em todos os outros; o segundo vértice s está ligado em todos os vértices exceto o primeiro e assim sucessivamente até o último vértice. Se V e E são iguais, respectivamente, ao número de vértices e arestas em G . Então, temos que o número máximo de arestas de G é dado pela Equação 4.3.

$$E = \frac{V(V - 1)}{2} \quad (4.3)$$

O Algoritmo 11 começa sua busca pelo vértice r , tendo sua prioridade atualizada para 1 (passo a). A seguir a aresta $\{r, u\}$ é visitada de modo que o vértice u tem sua

profundidade atualizada para 2 (passo *b*). Entretanto, não existe nenhuma aresta saindo de *u*. Sendo assim, a próxima aresta $\{r, t\}$ é visitada, tendo a profundidade de *t* atualizada (passo *c*). Este procedimento é repetido percorrendo as arestas em que os vértices estão mais a direita, de modo que o maior número de vértices e arestas sejam visitados. É interessante observar que no passo *f* uma aresta é percorrida mas não ocorre a atualização da prioridade, pois o vértice *u* possui prioridade superior à *s*.

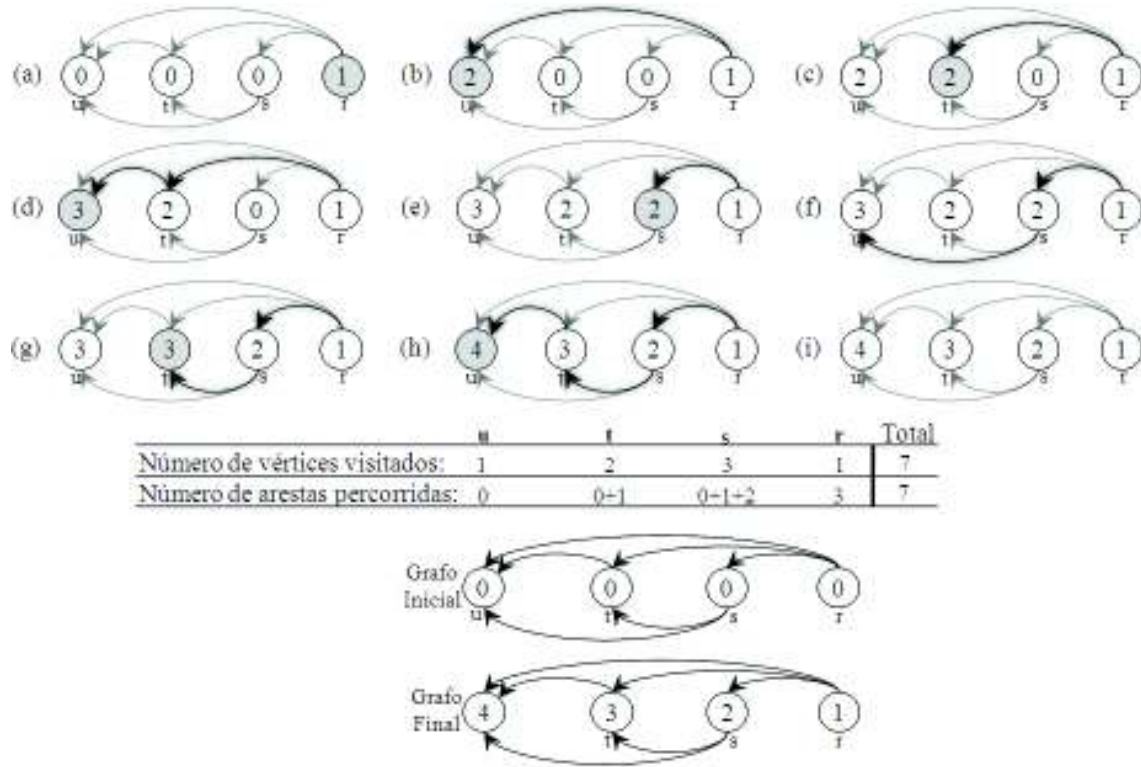


Figura 4.4: Grafo usado na análise do pior caso do Algoritmo 11.

Na parte inferior da Figura 4.4 é exibida uma tabela com as quantidades de vértices e arestas visitados a partir de um vértice. A partir do vértice *u* temos que apenas ele foi visitado e nenhuma aresta. Para o vértice *t* apenas a aresta $\{t, u\}$ e os vértices *t* e *u* são visitados. Para *s* são visitados 3 vértices e 3 arestas (2 arestas começando em *s* e 1 aresta começando em *t*). No caso de *r*, não podemos considerar aqueles vértices e arestas que já foram contabilizados, sendo assim, apenas o vértice *r* e as arestas $\{r, s\}$, $\{r, t\}$ e $\{r, u\}$, são consideradas.

A Figura 4.5 exibe outro exemplo com as quantidades de vértices e arestas visitados a partir de um vértice para um grafo com 6 vértices. Por exemplo, seguindo o mesmo procedimento anterior, para o vértice *s* temos que 5 vértices $\{s, t, u, v, w\}$ e todas as arestas que têm origem nestes vértices são visitados. Ou seja, 4 arestas do vértice *s*, 3 arestas do vértice *t*, 2 arestas do vértice *u*, 1 aresta do vértice *v* e nenhuma aresta do vértice *w*. Para generalizar a contagem destas operações é possível formular algumas somas baseadas no

número de vértices.

	w	v	u	t	s	r	Total
Número de vértices visitados:	1	2	3	4	5	1	16
Número de arestas percorridas:	0	0+1	0+1+3	0+1+2+3	0+1+2+3+4	5	25

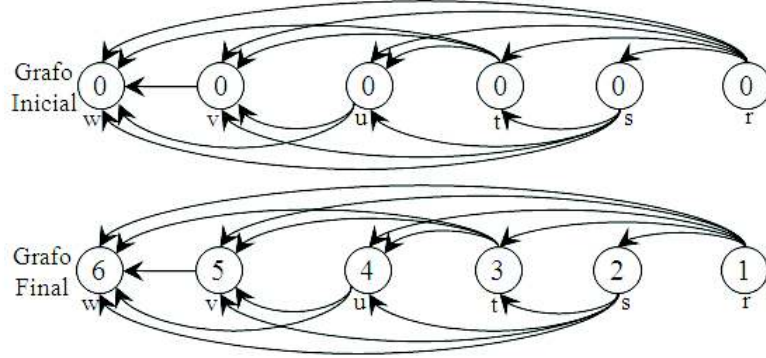


Figura 4.5: Número de vértices e arestas visitados em um grafo com 6 vértices.

A Equação 4.4 mostra este cálculo para o número de vértices visitados. Os somatórios são baseados nos conceitos do Apêndice A.

$$1 + \sum_{i=1}^{V-1} i = \frac{(V-1)(V-1+1)}{2} + 1 = \frac{V(V-1)}{2} + 1 = \frac{V^2}{2} - \frac{V}{2} + 1 \quad (4.4)$$

O cálculo de arestas visitadas é realizado pela Equação 4.5. Como cada vértice possui uma quantidade de arestas decrescente, da “direita” para a “esquerda”, este cálculo é apresentado por duas somas que representam os números de arestas que são visitadas em cada vértice. O limite i é associado ao número de arestas que estão saindo do vértice. O limite da primeira soma é $V-2$ em decorrência das seguintes considerações:

- No primeiro vértice temos apenas $V-1$ arestas que são somadas a parte.
- Analisando da “direita” para a “esquerda”, temos que o segundo vértice possui $V-2$ arestas, o terceiro vértice $V-3$, assim sucessivamente até o último vértice.

$$(V-1) + \sum_{i=0}^{V-2} \sum_{j=0}^i j = (V-1) + \sum_{i=0}^{V-2} \frac{i(i+1)}{2} = (V-1) + \frac{1}{2} \left(\sum_{i=0}^{V-2} i^2 + \sum_{i=0}^{V-2} i \right) \quad (4.5)$$

A Equação 4.5 pode ser simplificada, obtendo a Equação 4.6.

$$(V-1) + \frac{1}{2} \left(\frac{(V-2)(V-1)(2V-3)}{6} + \frac{(V-2)(V-1)}{2} \right) = \frac{V^3}{6} - \frac{V^2}{2} + \frac{4V}{3} - 1 \quad (4.6)$$

O total de operações é dado pela soma de todos os vértices e arestas visitados, sendo assim, o resultado final é computado pela soma das Equações 4.4 e 4.6. Este resultado é apresentado na Equação 4.7.

$$\frac{V^3}{6} + \frac{5V}{6} \tag{4.7}$$

Segundo a Equação 4.7, concluímos que a complexidade do Algoritmo 11 no pior caso é $O(V^3)$.

Capítulo 5

Escalonamento

Diversos algoritmos e estratégias podem ser utilizadas na tarefa de escalonamento de recursos. Normalmente, os algoritmos são *offline*, dispondo de “todas” as informações e tempo. Porém, o planejamento para jogos do tipo RTS ocorre em tempo real. Sendo assim, o algoritmo deve encontrar uma solução dentro de um intervalo de tempo. Para compensar essa restrição, os algoritmos de planejamento *online* devem melhorar a solução e imediatamente após o intervalo de tempo retornar uma solução ou parte de uma solução. Normalmente, estes algoritmos devem dispor de uma ferramenta de aprendizagem e busca.

O objetivo deste capítulo é de propor um método de escalonamento em tempo real (*online*) adequado a um jogo RTS. Uma solução é caracterizada por dois parâmetros: o *makespan* e o tempo de processamento para sua construção. Sendo assim, para realizar essa tarefa este trabalho propõe o uso do algoritmo SLA*.

O plano de ordem parcial gerado no Capítulo 3 é tratado como entrada para estes procedimentos, já que precisa ser escalonado, determinando o tempo exato de cada ação. Este plano é tratado como um projeto de atividades, ou seja, um conjunto de atividades relacionadas por precedências.

A Figura 5.1 mostra um plano de ordem parcial usado para ilustrar este processo. Considerando este plano como sendo um grafo, cada vértice corresponde a uma atividade. Cada atividade é associada a uma quantidade de ações, um número que representa a atividade, o maior valor de duração para completar a atividade (sem considerar as restrições de recursos) e uma prioridade.

O valor de maior duração das atividades é tratado como a heurística inicial, usado no processo de aprendizado para o cálculo das heurísticas dos estados da busca. A heurística é um limite inferior de duração da estimativa para a conclusão do restante de atividades do projeto. É importante lembrar que cada ação possui um tempo de duração, sendo este valor usado na estimativa de heurística. A atividade 0 corresponde ao estado atual de recursos e 6 a meta. A meta representa os recursos mínimos. Sendo assim, é possível que existam recursos extras após a execução de todas as ações.

A prioridade, calculada pelo procedimento do Capítulo 4, não obriga que uma ação

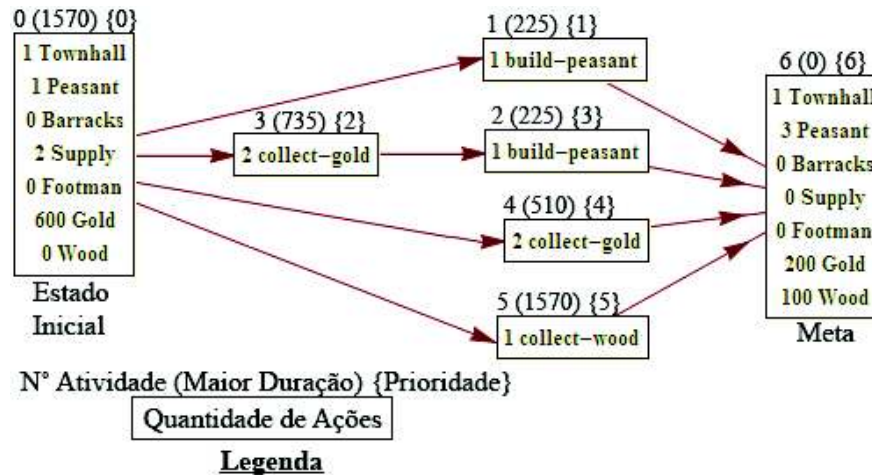


Figura 5.1: Plano de ordem parcial usado como exemplo no escalonamento utilizando o SLA*.

seja executada antes ou depois, mas define uma preferência de execução dentre as demais com menor prioridade. O conceito de prioridade é usado durante o escalonamento.

Para otimizar o escalonamento usando o algoritmos SLA* são aplicadas quatro estratégias para restringir o espaço de busca e encontrar uma solução mais rapidamente. Tais estratégias são denominadas: “Máximo de Ações”, “Menor Tempo”, “Uso de Prioridade” e “Limite de Estados”. Além disso, são apresentados detalhes de implementação que tornam o escalonamento mais rápido.

As próximas seções exemplificam o processo de escalonamento e as estratégias de eficiência. Na Seção 5.1, é apresentada a geração de planos usando o SLA*. Posteriormente, são descritas as estratégias “Máximo de Ações”, “Menor Tempo”, “Uso de Prioridade” e “Limite de Estados”. Ao final são apresentados detalhes de implementação na Seção 5.1.3.

5.1 SLA* Aplicado no Escalonamento de Recursos em Jogos RTS

Nesta seção será descrito o algoritmo SLA* aplicado em planos de ordem parcial criados pelo planejamento do Capítulo 3. Este procedimento é similar ao da Seção 2.2.3. Entretanto, os planos de ordem parcial possuem 7 recursos com quantidade variável, tornando o problema mais complexo e demandando mais tempo de escalonamento. Um detalhe importante é que uma ação pode influenciar na execução das próximas ações, devido a produção dos recursos. Devido a essas características, são aplicadas algumas regras para otimizar este procedimento. Para descrever este escalonamento é demonstrada a aplicação do SLA* no plano da Figura 5.1.

Inicialmente são feitas algumas considerações a respeito do algoritmo. Cada estado corresponde a um escalonamento de ações. Como as atividades do plano são compostas

por uma ou mais ações, um escalonamento pode conter uma ou todas as ações de uma atividade. O escalonamento é composto por todas as possibilidades de combinações distintas (ações que pertencem a mesma atividade são consideradas iguais) das ações que podem ser usadas em um dado momento. O escalonamento considera as atividades individualmente. Deste modo, o escalonamento é realizado segundo as ações destas atividades. Mesmo que as ações de duas atividades sejam as mesmas, o escalonamento contempla ambas como sendo distintas.

De acordo com os conceitos da Seção 2.2.3 e do Algoritmo 3, cada estado consiste em um conjunto de ações que estão em progresso, ou seja, ações sendo escalonadas. Para o escalonamento em jogos RTS, quando os estados sucessores são criados no passo de seleção (linha 7 do Algoritmo 3), são usados os Algoritmos 12 e 13.

Algorithm 12 EstadosSucessores(*Recursos*, *Atividades*)

```

1: Variável global Estados  $\leftarrow \emptyset$ , inicialmente sem elementos. Sucessores que serão criados considerando os recursos Recursos e atividades Atividades.
2:  $n \leftarrow$  quantidade de atividades em Atividades
3: for  $i \leftarrow 1, \dots, n$  do
4:    $S_i \leftarrow 0$ 
5: end for
6: AdicionarEstados( $A_1$ ,  $S$ , Recursos, Atividades)
7: return Estados

```

O Algoritmo 12, **EstadosSucessores**(*Recursos*, *Atividades*), é responsável por instanciar as variáveis usadas na criação dos estados sucessores. O resultado deste procedimento corresponde ao escalonamento das ações presentes em *Atividades* considerando o conjunto de recursos *Recursos*. O primeiro parâmetro representa os recursos disponíveis. O segundo consiste nas atividades que podem ser escalonadas, sendo que cada atividade é associada a uma ou mais ações do mesmo tipo. Por exemplo, a atividade 3 da Figura 5.1 possui 2 ações do tipo *collect-gold*. Ao serem criados, os estados criados são adicionados a variável global *Estados*, inicialmente sem elementos. A variável S corresponde às ações que estão sendo escalonadas em um dado momento. Inicialmente não possui nenhuma ação e contem a mesma quantidade de atividades que *Atividades*.

O Algoritmo **AdicionarEstados**(A_j , S , *Recursos*, *Atividades*) é responsável por criar os estados e os adicionar a variável *Estados*. O primeiro parâmetro especifica a atividade inicial do escalonamento. As atividades presentes em *Atividades* estão organizadas sequencialmente e jamais alteram sua ordem. Essas atividades são identificadas por índices, sendo a primeira igual a 1, a segunda igual a 2 e assim progressivamente até a última atividade. O parâmetro S apresenta a número de ações que foram escalonadas em um estado prévio, sendo relacionadas a cada atividade durante a execução do método. Os recursos disponíveis em *Recursos* são aqueles considerados para a execução das ações, ou seja, uma ação pode fazer parte de um escalonamento, gerando um novo estado, apenas se estes recursos satisfizerem as pré-condições da ação.

Algorithm 13 AdicionarEstados($A_j, S, Recursos, Atividades$)

```

1: for each Atividade  $A_i \in Atividades$ , tal que  $i \geq j$  do
2:    $\beta \leftarrow$  ação associada a atividade  $A_i$ 
3:    $s_i \leftarrow$  quantidade de ações da atividade  $A_i$  já escalonadas em  $S$ 
4:    $a_i \leftarrow$  quantidade de ações da atividade  $A_i$  em  $Atividades$ 
5:   if  $((s_i < a_i)$  and  $(\beta$  pode ser executada com os recursos de  $Recursos$ )) then
6:      $S' \leftarrow$  novo estado com as atividades e ações de  $S$  mais o incremento de 1 ação
       da atividade  $A_i$ 
7:      $Estados.adiciona(S')$ 
8:     AdicionarEstados( $A_i, S'$ , recursos ao executar  $\beta$  em  $Recursos$ ,
        $Atividades$ )
9:   end if
10: end for

```

Um novo estado é criado apenas se a quantidade total de ações já escalonadas for inferior à quantidade disponível e se os recursos satisfazem as pré-condições da ação. É importante salientar que as ações são avaliadas entre as atividades com mesmo índice. Se for possível realizar o escalonamento, um novo estado S' é criado com o acréscimo de uma ação para a respectiva atividade. A seguir, S' é adicionado a lista *Estados* e o método é chamado novamente para tentar criar novos escalonamentos baseado em S' e nos recursos disponíveis. O método **AdicionarEstados** é executado sequencialmente a partir das atividades com índice j até o último elemento.

Para tornar claro estes algoritmos, será descrito um exemplo considerando os parâmetros $Recursos = \{1\ townhall, 2\ peasant, 2\ supply, 700\ gold\}$ e $Atividades = \{1\ build-peasant, 1\ collect-gold, 3\ collect-gold, 2\ collect-wood\}$. Na primeira chamada do algoritmo **AdicionarEstados**, além dos valores iniciais em *Recursos* e *Atividades*, temos que $A_j = 1$ e S possui 4 atividades sem ações. O procedimento inicia pela atividade de índice 1 com a ação *build-peasant*. Tal ação pode ser escalonada, já que existem recursos disponíveis para satisfazer suas pré-condições (*1 townhall, 400 gold, 1 supply*). Sendo assim, um estado contendo uma ação da primeira atividade é adicionado a lista *Estados*.

Posteriormente, uma nova chamada é realizada para o método **AdicionarEstados**, considerando a atividade 1, o estado criado e os recursos após executar *build-peasant* (*0 townhall, 2 Peasant, 1 supply, 300 gold*). Entretanto, para essa atividade não é possível escalonar nenhuma ação, já que a única ação desta atividade já foi escalonada anteriormente. O processo continua com a atividade 2 para a ação *collect-gold*. Os recursos permitem que tal ação seja escalonada, criando mais um estado e chamando novamente **AdicionarEstados**. Este processo continua até que todas as possibilidades sejam encontradas. O resultado final é apresentado na Figura 5.2.

Na Figura 5.2, temos a legenda na área superior esquerda. Observamos que na parte superior direita são descritos os parâmetros iniciais ($A_j, S, Recursos, Atividades$). Para cada estado, em cada linha, temos a quantidade de ações que estão sendo escalonadas e entre colchetes o índice da atividade correspondente. Na parte superior dos estados é

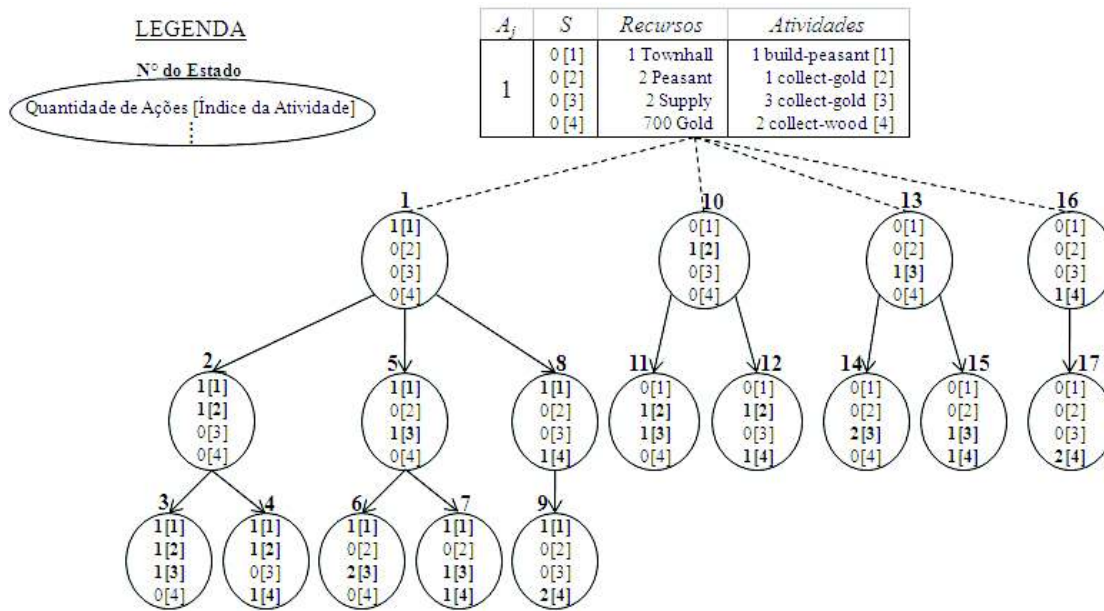


Figura 5.2: Exemplo de estados que são criados em um passo do escalonamento.

apresentado um número que representa o escalonamento. Este valor é crescente a partir de 1 e corresponde à ordem que os estados são criados e adicionados a lista *Estados*. Por exemplo, o estado 6 representa o escalonamento de uma ação *build-peasant* da atividade 1 e duas ações *collect-gold* da atividade 3.

Existem detalhes importantes que são considerados ao utilizar o SLA* na tarefa de escalonamento. As ações possuem diferentes tempos de duração, sendo assim, um estado é criado para cada diferente tempo de término das ações. Como apresentado na Seção 2.2.3, os estados são compostos por três grupos de atividades: *completo*, *em progresso* e *não escalonadas*. O plano de ordem parcial, de modo similar ao projeto de atividade da Figura 2.10, é composto por uma atividade inicial e final, com durações igual a zero. A seguir, na Seção 5.1.1 é descrito um exemplo de aplicação do algoritmo SLA* sobre o plano da Figura 5.1, considerando todos estes conceitos já apresentados.

5.1.1 Exemplo

A partir dos conceitos apresentados é aplicado o algoritmo SLA* sobre o plano de ordem parcial da Figura 5.1. A representação de um estado é composta por três conjuntos, *completo*, *em progresso* e *não escalonadas*. Nestes estados os conjuntos com as atividades são apresentados entre de parênteses, ou seja: (*completo*, *em progresso*, *não escalonadas*). Entretanto, como as atividades do plano podem ser compostas por mais de uma ação, cada estado descreve além das atividades, o número de ações relacionados a tal atividade para cada estado. Essa quantidade de ações é apresentada a frente das atividades, sendo separados por um hífen (-). O “Estado Inicial” e a “Meta” não são associados com ações, representando apenas recursos.

Inicialmente é adicionada a atividade 0 a busca. Essa atividade faz parte do conjunto de estados *completo*, já que não necessita de tempo de duração e consiste nos recursos atuais, sendo assim, temos (0, , 1-1 1-2 2-3 2-4 1-5 6). O valor de heurística deste estado é expresso pelo valor máximo das próximas ações que podem ser escalonadas, deste modo a heurística é $\max(225, 735, 510, 1570) = 1570$. Além disso, é importante lembrar que este estado é caracterizado como sendo a raiz, ou seja, o primeiro escalonamento que representa o estado inicial e jamais é retirado da *PILHA* do SLA*. Para este estado o escalonamento é composto por todas as combinações das ações que podem ser escalonadas, ou seja, as ações das atividades 1, 3, 4 e 5. Sendo assim, são criados 10 novos estados, apresentados a seguir:

- Estado (0, 1-1, 1-2 2-3 2-4 1-5 6)
Duração de 225 e heurística igual a $225 + \max(735, 510, 1570) = 1795$.
- Estado (0, 1-1 1-3, 1-2 1-3 2-4 1-5 6)
Duração de 225 e heurística igual a $225 + \max(735, 510, 1570) = 1795$.
- Estado (0, 1-1 1-3, 1-2 1-3 2-4 1-5 6)
Duração de 510 e heurística igual a $510 + \max(735, 510, 1570) = 2080$.
- Estado (0, 1-1 1-4, 1-2 2-3 1-4 1-5 6)
Duração de 225 e heurística igual a $225 + \max(735, 510, 1570) = 1795$.
- Estado (0, 1-1 1-4, 1-2 2-3 1-4 1-5 6)
Duração de 510 e heurística igual a $510 + \max(735, 510, 1570) = 2080$.
- Estado (0, 1-1 1-5, 1-2 2-3 2-4 6)
Duração de 225 e heurística igual a $225 + \max(735, 510, 1345) = 1570$.
- Estado (0, 1-1 1-5, 1-2 2-3 2-4 6)
Duração de 1570 e heurística igual a $1570 + \max(735, 510) = 2305$.
- Estado (0, 1-3, 1-1 1-2 1-3 2-4 1-5 6)
Duração de 510 e heurística igual a $510 + \max(225, 735, 510, 1570) = 2080$.
- Estado (0, 1-4, 1-1 1-2 2-3 1-4 1-5 6)
Duração de 510 e heurística igual a $510 + \max(225, 735, 510, 1570) = 2080$.
- Estado (0, 1-5, 1-1 1-2 2-3 2-4 6)
Duração de 1570 e heurística igual a $510 + \max(225, 735, 510) = 2305$.

A Figura 5.3 exibe este escalonamento. A seguir, o estado com o menor valor de heurística, (0, 1-1 1-5, 1-2 2-3 2-4 6), é adicionado a busca. É interessante observar que a heurística deste estado é calculada pela duração restante da atividade 5. Os recursos deste estado são: 1 *Townhall*, 1 *Peasant*, 0 *Barracks*, 1 *Supply*, 0 *Footman*, 200 *Gold*, 0 *Wood* e 1 *Peasant Borrow*. O *peasant borrow* está sendo usado pela atividade 5, deste modo esse

recurso é liberado apenas ao término desta ação. O *Peasant* que está disponível neste estado corresponde ao efeito da atividade 1.

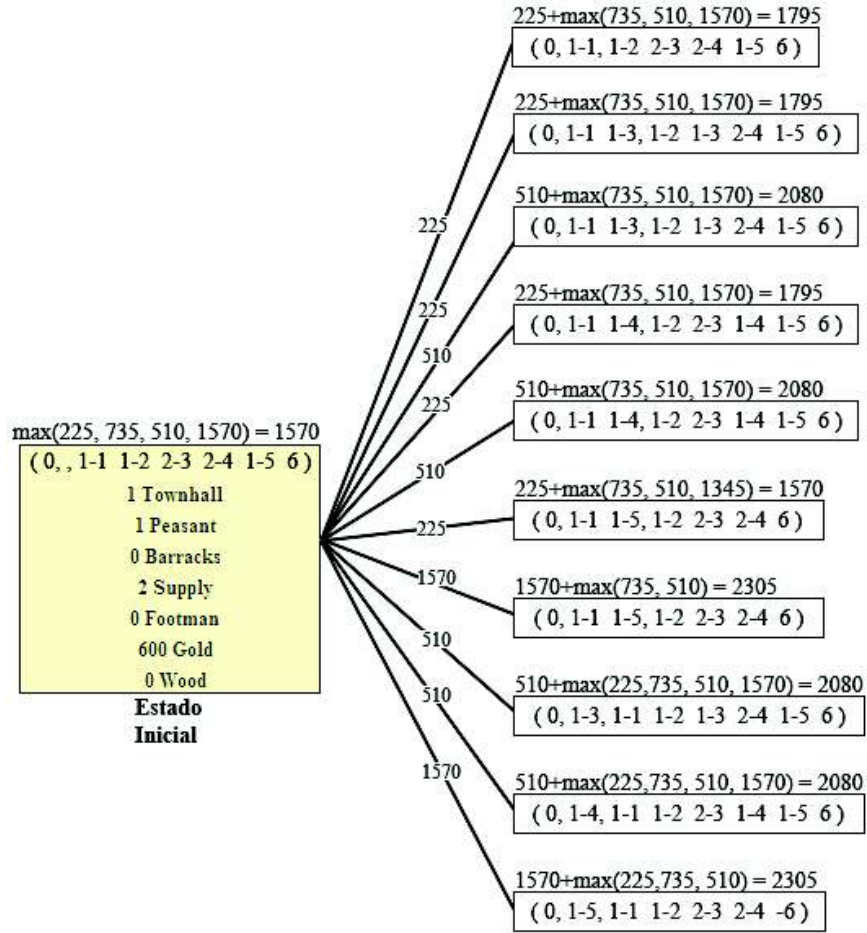


Figura 5.3: Primeiro passo da busca do SLA* para o plano da Figura 5.1.

É executado um novo escalonamento para o estado atual (0, 1-1 1-5, 1-2 2-3 2-4 6). Para essa busca os estados criados são (0 1-1, 1-3 1-5, 1-2 1-3 2-4 6) com duração de 510, (0 1-1, 1-3 1-5, 1-2 1-3 2-4 6) com duração de 1345, (0 1-1, 1-4 1-5, 1-2 2-3 1-4 6) com duração de 510 e (0 1-1, 1-4 1-5, 1-2 2-3 1-4 6) com duração de 1345. Os valores totais de heurística são respectivamente: $225 + 510 + \max(735, 510, 835) = 1570$, $225 + 1345 + \max(735, 510) = 2305$, $225 + 510 + \max(735, 510, 835) = 1570$ e $225 + 1345 + \max(735, 510) = 2305$. O resultado é exibido na Figura 5.4.

O estado (0 1-1, 1-3 1-5, 1-2 1-3 2-4 6) com heurística igual a 1570 é adicionado a busca. A seguir é realizado o escalonamento das próximas ações, obtendo os estados (0 1-1 1-3, 1-3 1-5, 1-2 2-4 6) com duração igual a 510, (0 1-1 1-3, 1-3 1-5, 1-2 2-4 6) com duração igual a 835, (0 1-1 1-3, 1-4 1-5, 1-2 1-3 1-4 6) com duração igual a 510 e (0 1-1 1-3, 1-4 1-5, 1-2 1-3 1-4 6) com duração igual a 835. Para cada estado é calculado o valor total de heurística, sendo igual a, respectivamente: $225 + 510 + 510 + \max(225, 510, 325) = 1755$, $225 + 510 + 835 + \max(225, 510) = 2080$, $225 + 510 + 510 + \max(735, 510, 325) = 1980$

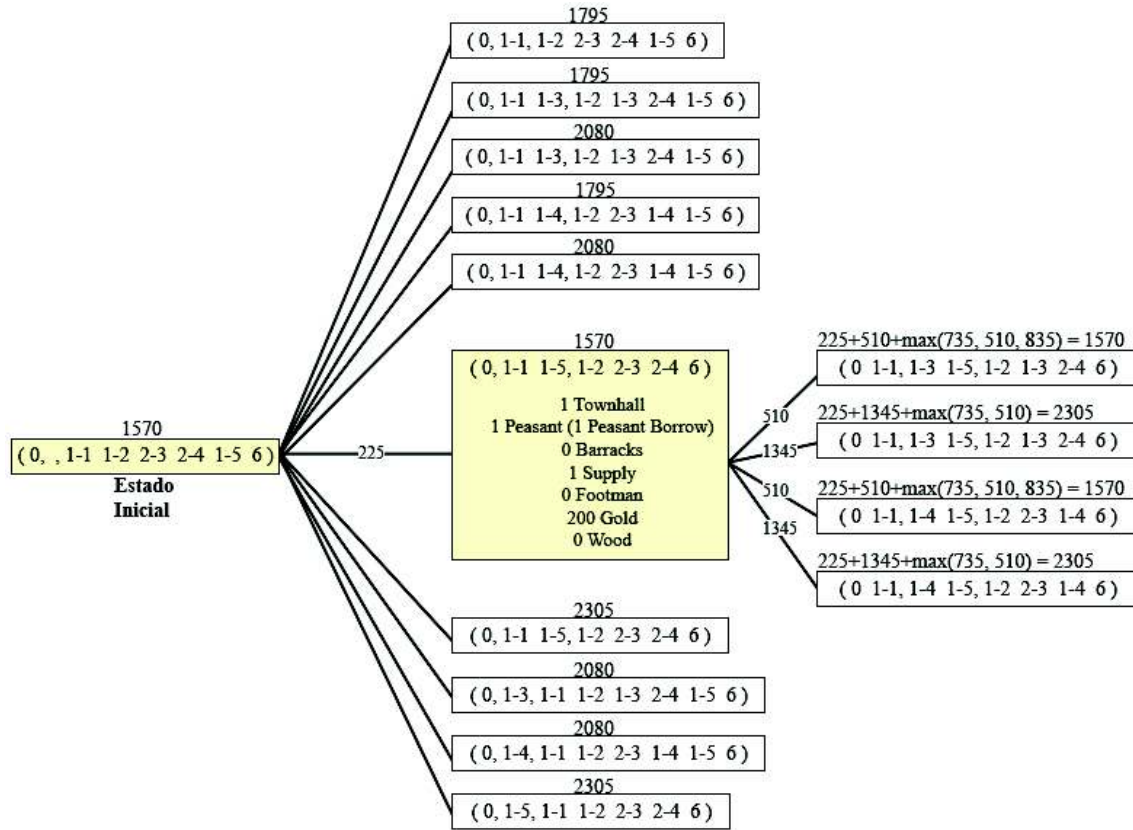


Figura 5.4: Segundo passo da busca do SLA* para o plano da Figura 5.1.

Tabela 5.1: Resultado resumido do processo de busca para o exemplo utilizado no SLA*.

n	$h(\text{raiz})$	$t = 0$	$t = 225$	$t = 735$	$t = 1020$	$t = 1245$	$t = 1795$
1	1570	1.1,1.5	1.3	(1.3)<185>			
2	1570	1.1,1.5	1.4	(1.3)<410>			
3	1570	1.1,1.5	(1.3)<185>				
4	1755	(1.1,1.5)<185>					
5	1755	1.1,1.5	1.3	1.3		(1.2,1.4)<225>	
6	1755	1.1,1.5	1.3	(1.3)<225>			
7	1755	1.1,1.5	(1.3)<225>				
8	1795	(1.1,1.3)<40>					
		1.1,1.3	1.5	1.3	1.2,1.4	1.4	1.6

e $225 + 510 + 835 + \max(735, 510) = 2305$. O estado com menor valor de heurística é escolhido, como seu valor de heurística é maior que o atual, a heurística do estado atual é atualizada. Sendo assim, o estado (0 1-1, 1-3 1-5, 1-2 1-3 2-4 6) com heurística de 1570 é atualizado para 1755. A Tabela 5.1 apresenta os escalonamentos e atualização de heurística para o exemplo tratado.

A coluna n da Tabela 5.1 representa os passos do aprendizado de heurística, $h(\text{raiz})$ é o valor estimado de heurística para o estado inicial e t é o tempo em que cada atividade com uma determinada quantidade de ações começa a ser executada. Para cada tempo t é associado um conjunto de ações referente as atividades, um hífen (-) separa o número de ações executadas e a atividade. Um estado selecionado que causa o aprendizado de

heurística é colocado entre () e o valor do aprendizado entre <>. O retrocesso na busca ocorre em cada linha subsequente. O valor de heurística $h(raiz)$ é atualizado apenas se ocorrer o retrocesso até o estado inicial, de modo que nenhum estado filho possui heurística menor ou igual a $h(raiz)$. O último passo da busca, descrito na Tabela 5.1, consiste na solução ótima, sendo mostrado no plano *C* da Figura 5.5.

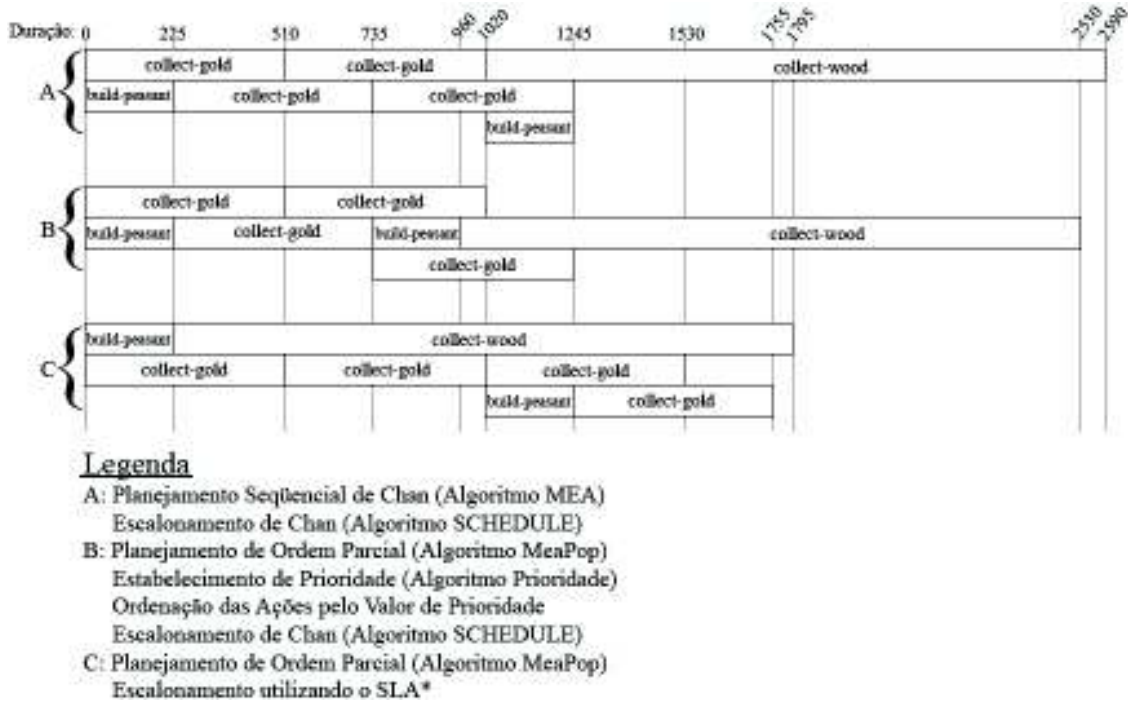


Figura 5.5: Planos obtidos ao usar o planejamento de [Chan et al. 2007b] (A), *MeaPop* (B) e SLA* (C).

O problema consiste em gerar um plano para o estado inicial $\{1 \text{ Townhall}, 1 \text{ Peasant}, 2 \text{ Supply e } 600 \text{ Gold}\}$ e meta $\{1 \text{ Townhall}, 3 \text{ Peasant}, 200 \text{ Gold e } 100 \text{ Wood}\}$. Na Figura 5.5 são apresentadas as soluções *A*, *B* e *C*. Na parte superior temos o tempo em que cada ação é executada. Cada plano mostra o resultado obtido por três diferentes métodos de planejamento. A solução *A* é obtida ao usar o sistema de planejamento de [Chan et al. 2007b]. Em *B* é aplicado o Algoritmo **MeaPop**. Em seguida é estabelecida a prioridade e o plano de ordem parcial é ordenado sequencialmente segundo o valor de prioridade. Ao final este plano sequencial é escalonado utilizando o método de escalonamento de [Chan et al. 2007b]. No plano *C* são aplicados os mesmo procedimento que em *B*, exceto o procedimento de escalonamento que é realizado pelo SLA*. Observamos que o melhor plano é *C*, com *makespan* igual a 1795.

Em problemas mais simples, para um dado plano de ordem parcial, o algoritmo SLA* encontra rapidamente a solução ótima. Mas para problemas com muitas ações a convergência demanda muito tempo, sendo assim, algumas estratégias são usadas para otimizar este processo. São propostas quatro estratégias para otimizar o SLA*, como é

descrito na Seção 5.1.2.

5.1.2 Estratégias de Eficiência

Para melhorar a eficiência da aplicação do SLA* são usadas algumas estratégias que tornam o algoritmo mais rápido. Entretanto, a solução ótima nem sempre é encontrada para um determinado plano de ordem parcial. A seguir são apresentadas, respectivamente, as estratégias “Máximo de Ações”, “Menor Tempo”, “Uso de Prioridade” e “Limite de Estados”.

Máximo de Ações

A estratégia “Máximo de Ações” consiste em aplicar as ações enquanto for possível e considerar os estados com a maior quantidade de ações. O estado é criado apenas se as ações das atividades que podem ser escalonadas não sejam possíveis de serem aplicadas. O Algoritmo 13 é facilmente alterado para realizar essa tarefa. Para isso é preciso apenas alterar o momento que o estado é adicionado. Este procedimento é apresentado no Algoritmo 14.

Algorithm 14 MaximoAcoes($A_j, S, Recursos, Atividades$)

```

1: maximo  $\leftarrow$  true
2: for each Atividade  $A_i \in Atividades$ , tal que  $i \geq j$  do
3:    $\beta \leftarrow$  ação associada a atividade  $A_i$ 
4:    $s_i \leftarrow$  quantidade de ações da atividade  $A_i$  já escalonadas em  $S$ 
5:    $a_i \leftarrow$  quantidade de ações da atividade  $A_i$  em  $Atividades$ 
6:   if  $((s_i < a_i)$  and  $(\beta$  pode ser executada com os recursos de  $Recursos$ )) then
7:     maximo  $\leftarrow$  false
8:      $S' \leftarrow$  novo estado com as atividades e ações de  $S$  mais o incremento de 1 ação
       da atividade  $A_i$ 
9:     MaximoAcoes( $A_i, S',$  recursos ao executar  $\beta$  em  $Recursos, Atividades$ )
10:  end if
11: end for
12: if maximo = true then
13:   Estados.adiciona( $S$ )
14: end if

```

A alteração do algoritmo é dada com o acréscimo da variável *maximo*. Se em um dado momento nenhuma ação pode ser escalonada, *maximo* fica com valor igual a *true*, caso contrário *false*. Se *maximo* é *true*, o estado antecessor S é adicionado na variável *Estados*. Considerando o exemplo da Figura 5.2, os estados obtidos com a aplicação desta estratégia são as “folhas” 3, 4, 6, 7, 9, 11, 12, 14, 15 e 17. A Figura 5.6 apresenta os estados em negrito que são criados.

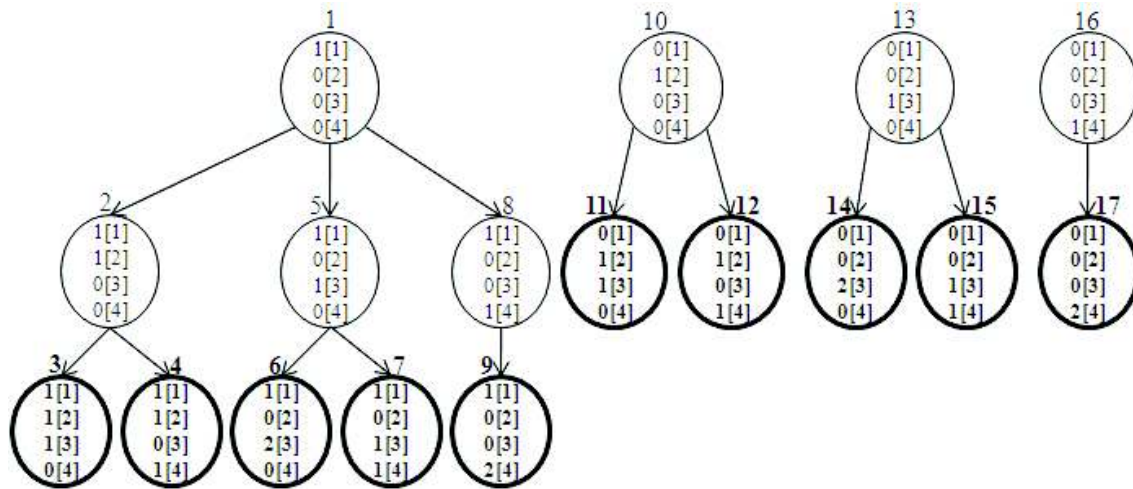


Figura 5.6: Estados que são criados ao considerar a técnica “Máximo de Ações”.

Menor Tempo

Como descrito na Seção 5.1, para cada estado é definido um tempo de duração de acordo com o término das ações. Todos os tempos de duração distintos constituem um novo estado para o SLA*. Entretanto, mesmo usando o Algoritmo 14, o número de estados é muito grande. Além disso, os tempos distintos tornam o problema ainda pior. Para compensar em parte este problema, uma estratégia consiste em criar apenas dois estados considerando os tempos de duração. O primeiro avalia o menor tempo da ação que estava em progresso de um estado pai. O segundo corresponde ao menor tempo entre as ações em progresso do atual momento do escalonamento. Um exemplo é apresentado na Figura 5.7.

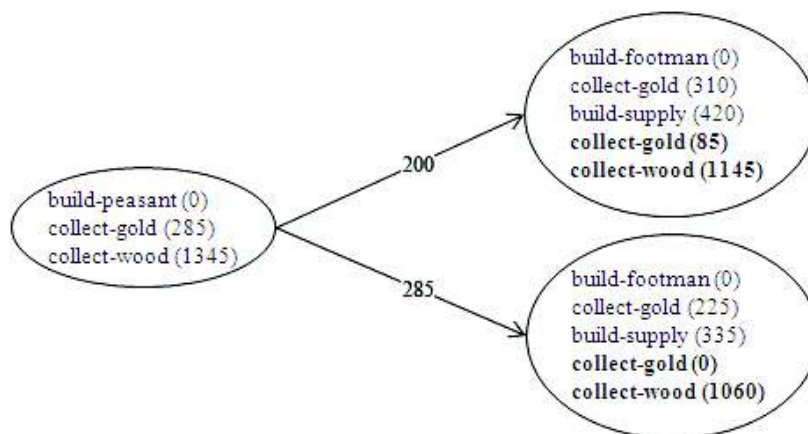


Figura 5.7: Estados criados ao usar a técnica “Menor Tempo”.

O primeiro estado, a esquerda, consiste no escalonamento de três ações: *build-peasant*, *collect-gold* e *collect-wood*. O tempo deste estado é igual a 225. À direita das ações é mostrado o tempo que falta para terminar cada ação. Na parte direita da figura são

exibidos os dois estados que foram criados. Estes estados estão escalonando três novas ações: *build-footman*, *collect-gold* e *build-supply*. O primeiro estado, com duração igual a 200, corresponde ao escalonamento com menor tempo dentre as ações que começaram a ser executadas neste momento. O segundo estado, com duração de 285, é referente ao menor tempo dentre as ações do estado pai.

Uso de Prioridade

O problema do planejamento usando o SLA* é que nem sempre uma ação que foi escalonada é aproveitada. Isso ocorre pois existem algumas atividades que possuem o mesmo tipo de ação. De certo modo não faz diferença qual ação é usada no escalonamento se forem iguais. A tentativa é de permitir que a ação escalonada torne possível o quanto antes a satisfação das atividades mais importantes. Por exemplo, uma atividade que possui a ação *build-peasant* é melhor para o planejamento que uma atividade que possui a ação *build-footman*. Além disso, quanto antes as atividades precedentes a ação *build-peasant* forem escalonadas, melhor o planejamento, já que não faz sentido de se executar ações de *build-footman* enquanto existir ações que satisfazem *build-peasant*.

Segundo o plano de ordem parcial da Figura 5.8 e considerando as atividades 1 e 8, é melhor escalonar a atividade 1 que a atividade 8. Isso ocorre pois o *peasant* pode ser usado para melhorar o plano, enquanto o *footman* não altera o plano, já que é usado apenas para satisfazer a meta.

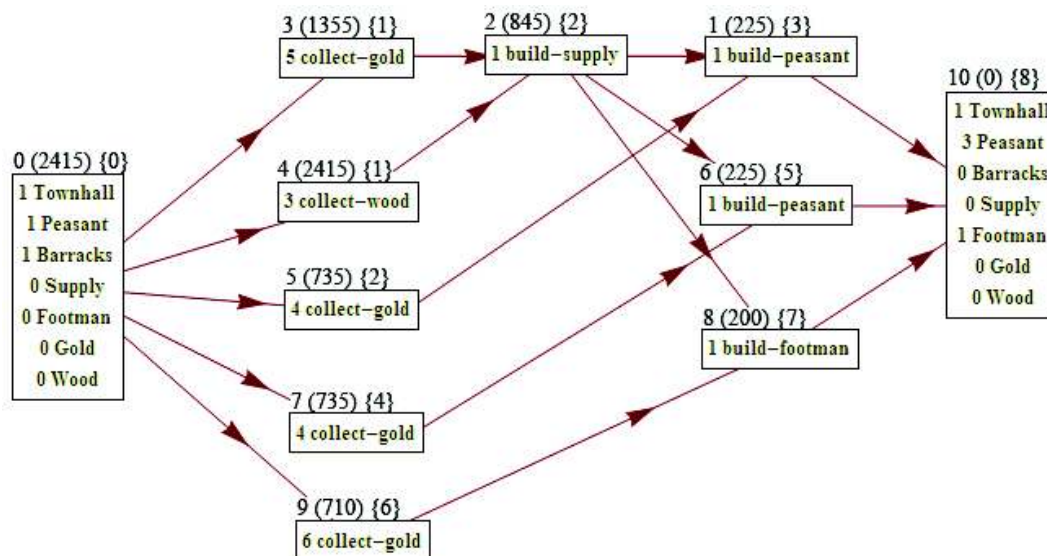


Figura 5.8: Exemplo de um plano de ordem parcial usado na estratégia “Uso de Prioridade”.

Para este plano de ordem parcial, o primeiro escalonamento considera as atividades 3, 4, 5, 7 e 9. Se forem gerados todos os estados, o problema pode demorar muito, sendo que alguns escalonamentos são desnecessários. Por exemplo, não faz sentido executar

as ações *collect-gold* da atividade 9 enquanto existirem ações da atividade 3 disponíveis, pois é mais importante criar um *peasant* que um *footman*. Para eliminar alguns estados aparentemente desnecessários, é usado o valor de prioridade calculado no Capítulo 4.

A cada passo onde serão criados novos estados, as atividades que possuem os mesmos tipos de ações são agrupadas de acordo com o valor de prioridade. No momento que uma atividade não possuir mais ações o procedimento passa a escalonar as ações das próximas atividades. Como apresentado anteriormente, para o escalonamento das atividades 3, 4, 5, 7 e 9, são criados 5 estados, 1 com cada ação de cada atividade. A estratégia, para este caso, consiste em considerar o escalonamento possível para dois grupos:

- *collect-gold*: formado sequencialmente pelas atividades 3, 5, 7 e 9.
- *collect-wood*: formado pela atividade 4.

Após o primeiro escalonamento, o procedimento continua sendo executado normalmente, considerando a ordenação das atividades pelo tipo de ação e prioridade. Considerando essa estratégia, para o exemplo anterior, seriam criados apenas 2 estados, um com cada ação das atividades 3 e 4.

Para o exemplo da Figura 5.4 o uso da estratégia “Uso de Prioridade” tem como resultado os estados da Figura 5.9.

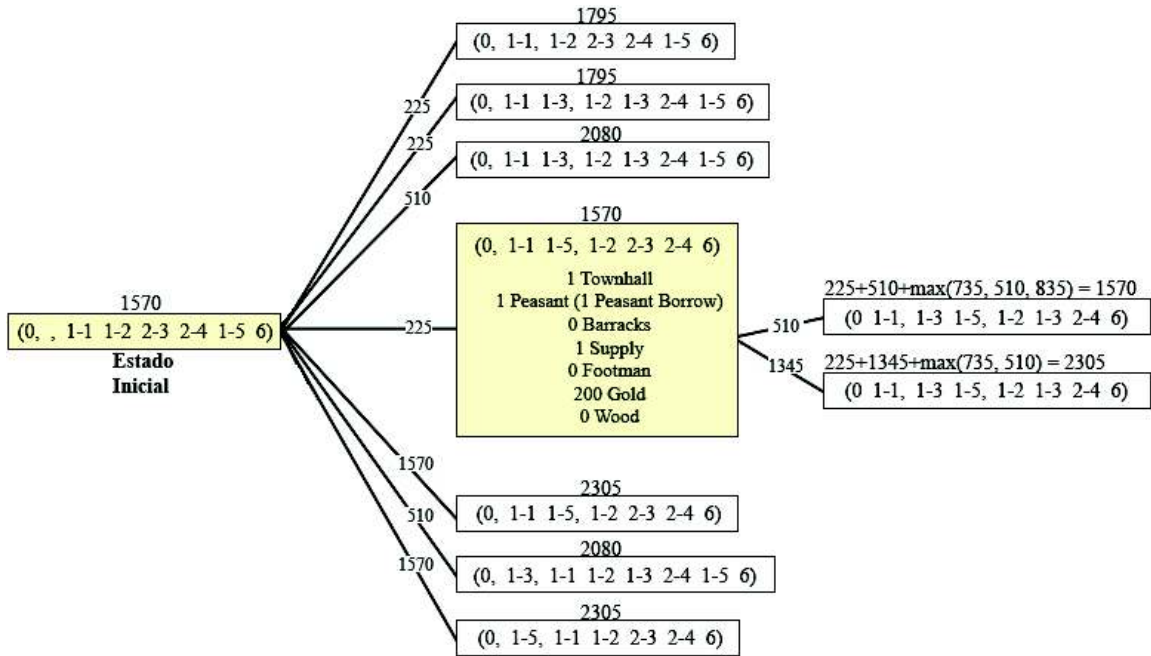


Figura 5.9: Primeiro e segundo passos da busca ao usar a técnica “Uso de Prioridade”.

São criados 14 estados para o exemplo da Figura 5.4, enquanto o uso da estratégia “Uso de Prioridade” cria 9 estados. É importante lembrar que este exemplo é muito simples, e que para problema complexos, com diversas ações, a quantidade de estados pode ser muito grande, tornando a busca extremamente demorada.

Limite de Estados

Essa estratégia é muito simples. Basicamente, é determinado uma quantidade máxima de estados que podem ser criados em um passo do escalonamento. Inicialmente é determinado um valor n , indicando que no máximo n estados podem ser criados. Os melhores estados são mantidos e os piores são eliminados. Para fazer este corte, é considerado o valor inicial de heurística, quanto menor, melhor é o estado. Caso ocorra o empate entre os valores de heurística, a quantidade de ações é considerada, quanto mais ações, melhor. Se a igualdade se mantiver, é feita uma escolha aleatória.

Por exemplo, se $n = 3$ para o escalonamento da Figura 5.3, apenas os estados a seguir são criados:

- Estado (0, 1-1 1-3, 1-2 1-3 2-4 1-5 6), com duração de 225 e heurística igual a 1785.
- Estado (0, 1-1 1-4, 1-2 2-3 1-4 1-5 6), com duração de 225 e heurística igual a 1785.
- Estado (0, 1-1 1-5, 1-2 2-3 2-4 6), com duração de 225 e heurística igual a 1570.

É interessante observar que ocorreu a igualdade de heurística entre alguns estados, sendo realizado o desempate pela quantidade de ações. O objetivo deste procedimento é de limitar a quantidade de estados e considerar apenas os melhores no momento que são criados.

5.1.3 Detalhes de Implementação

Um dos problemas do SLA* é que durante a busca alguns estados podem se repetir. Dois estados são considerados repetidos se os sucessores forem os mesmos. Isso implica em atualizações redundantes. Uma solução para este problema consiste em evitar que os estados repetidos sejam criados. Contudo, dependendo da estrutura de dados utilizada para armazenar os estados, verificar se um estado existe pode demandar uma busca por todo o espaço de estados.

Para solucionar os problemas de estados repetidos e busca é usada a estrutura de dados *Hashtable* [Cormen et al. 2001]. *Hashtable* é composta por uma “chave” e um “valor”. A chave é uma variável que identifica um objeto, sendo que este objeto corresponde ao “valor”. Normalmente, a chave é um número ou uma cadeia de caracteres. Não são permitidas chaves duplicadas. O “valor” é um variável que indica um determinado objeto.

É utilizada uma *Hashtable* principal que tem como chave um conjunto de recursos e o valor é outro *Hashtable*. Este segundo *Hashtable* tem como chave as ações que podem ser escalonadas e em progresso, enquanto o valor identifica um estado. Utilizar um *Hashtable* referenciando outro *Hashtable* torna a busca pelas “chaves” mais eficiente. Se fossem consideradas as duas chaves como sendo únicas o número de possibilidades poderia ser muito grande.

Para cada estado são atribuídos dez parâmetros, sendo:

completed: Ações que já foram escalonadas.

cost: Duração/Custo de realizar o escalonamento.

countTotalAction: Número de ações que estão em progresso (*inProgressCompleted* + *inProgressUncompleted*).

edge: Estados (escalonamentos) que são criados considerando os recursos e ações disponíveis após este estado. Ou seja, são os estados sucessores/adjacentes.

h: Valor de heurística.

inProgressCompleted: Ações que estão em progresso e serão finalizadas após a duração deste estado (*cost*).

inProgressUncompleted: Ações que estão em progresso, mas não terão término após a duração deste estado. Em outras palavras, parte da ação ainda fica em progresso após a duração deste estado (*cost*).

resourceStart: Recursos disponíveis no início deste estado.

resourceEnd: Recursos disponíveis no fim deste estado. Ou seja, considerando os recursos liberados e produzidos ao final das ações *inProgressCompleted*.

toScheduling: Ações que podem ser escalonadas após este estado.

A Figura 5.10 apresenta a arquitetura dos *Hashtables* e dos estados. O *Hashtable 1* é formado por uma chave de recursos. Essa chave é uma cadeia de caracteres que descreve a quantidade de cada recurso. A ordem dos recursos desta chave é *townhall*, *peasant*, *barracks*, *supply*, *footman*, *gold*, *wood*, *townhall borrow* e *peasant borrow*. Apenas os recursos *townhall* e *peasant* são do tipo *borrow*, por isso são os únicos recursos *borrow* da chave.

O *Hashtable 2* é formado por uma chave de ações. Essa chave é uma cadeia de caracteres que descreve a quantidade de ações de cada atividade que pode ser escalonada mais (separado por “_”) as ações em progresso não concluídas. A chave é ordenada de modo crescente segundo o número da atividade. Por exemplo, a chave “1 1 3 2 1 3 _1 1 200 1 4 150” possui o seguinte significado:

1. Ações que podem ser escalonadas.
 - 1 ação da atividade 1.
 - 3 ações da atividade 2.
 - 1 ação da atividade 3.
2. Ações que estão em progresso mas não concluídas.

siderações garantimos que um estado do *Hashtable* é o melhor. Se um estado repetido for fazer parte da busca ele possuirá heurística pior ou igual ao estado presente no *Hashtable*. Este estado repetido não é adicionado ao *Hashtable*.

Além do *Hashtable*, outra estrutura de dados foi usada na implementação do SLA*. Cada iteração do SLA* necessita selecionar o melhor sucessor do estado corrente. Essa seleção pode precisar de uma busca por todos os sucessores. O melhor sucessor corresponde ao estado com o menor valor de heurística h . Além da seleção, ocorrem atualizações da heurística h . A Figura 5.10 mostra o parâmetro *edge*. Nesta variável são armazenados os sucessores de cada estado.

Considerando n igual ao número de sucessores, no pior caso a complexidade para a seleção e atualização é, respectivamente, $O(n)$ e $O(1)$. A seleção do melhor estado ocorre inúmeras vezes, sendo assim, muito tempo é gasto neste passo. Para diminuir o tempo gasto é utilizada uma fila de prioridade baseada em *heap* [Cormen et al. 2001]. A complexidade para cada seleção e atualização utilizando *heap* é, respectivamente, $O(1)$ e $O(\log(n))$. A atualização é pior ao utilizar *heap*, porém, antes de ocorrer qualquer atualização é realizada uma seleção. Além disso, o número de seleções é muito maior que o de atualizações. É importante salientar que a construção do *heap* possui complexidade $O(n \cdot \log(n))$. Contudo, isso é compensado pelo grande número de passos de seleção, tornando a fila de prioridade baseada em *heap* melhor que uma busca sequencial.

Capítulo 6

Experimentos e Discussão dos Resultados

Para análise e discussão dos resultados foi usado um ambiente de testes. Um computador *Pentium Core Duo* 1.66 GHz com 1 GB de *Ram* executando o sistema operacional *Windows XP* foi utilizado nesta tarefa. O tempo de CPU gasto em cada problema é calculado pela média dos tempos de 10 execuções, sendo medido em milissegundos (ms). Em nenhum dos problemas é considerado o cálculo de recursos adicionais, apresentado na Seção 2.4.3.

Mea

Planejamento seqüencial de [Chan et al. 2007b] (Algoritmo 4).

Escalonamento de [Chan et al. 2007b] (Algoritmo 5).

MeaPop

Planejamento de ordem parcial (Algoritmo 7).

Estabelecimento de prioridade para atividades (Algoritmo 10).

Ordenação seqüencial das ações segundo o valor de prioridade.

Escalonamento de [Chan et al. 2007b] (Algoritmo 5).

SLA*(1)

Planejamento de ordem parcial (Algoritmo 7).

Escalonamento utilizando o SLA* (Algoritmo 3).

SLA*(2), SLA*(3), SLA*(4), SLA*(5)

Planejamento de ordem parcial (Algoritmo 7).

Estratégia “Máximo de Ações”.

Estratégia “Menor Tempo”.

Estratégia “Uso de Prioridade”.

Escalonamento utilizando o SLA* (Algoritmo 3).

SLA*(3)

Estratégia “Limite de Estados - 2”.

SLA*(4)

Estratégia “Limite de Estados - 4”.

SLA*(5)

Estratégia “Limite de Estados - 6”.

Figura 6.1: Resumo dos procedimentos usados nos experimentos.

Tabela 6.1: Resultados do Experimento 1.

	Mea	MeaPop	SLA*(1)	SLA*(2)	SLA*(3)	SLA*(4)
1	0,208	0,247	4221,578	11,912	11,207	11,846
2	0,211	0,212	4317,136	11,942	8,373	9,198
3	0,217	0,201	4302,851	9,342	8,575	9,450
4	0,210	0,195	4163,233	9,798	8,680	9,522
5	0,204	0,195	4419,502	13,757	8,740	17,342
6	0,204	0,204	4301,392	9,029	8,802	9,489
7	0,218	0,196	4228,612	17,760	11,580	9,486
8	0,208	0,194	4394,255	9,326	8,428	9,320
9	0,208	0,195	4363,905	9,003	12,093	9,549
10	0,208	0,204	4181,062	9,769	8,425	10,098
tempo (ms)	0,210	0,204	4289,353	11,164	9,490	10,530
makespan	16265	12920	12920	12920	12920	12920

A discussão dos resultados consiste em comparar os algoritmos e estratégias apresentados neste trabalho. Além disto, tem-se o objetivo de realizar mais experimentos, além daqueles apresentados em [Branquinho e Lopes 2009]. O método que não encontrar um plano em 30 segundos é desconsiderado da análise, sendo representado nas tabelas com um hífen. São utilizados sete procedimentos para os experimentos: *Mea*, *MeaPop*, *SLA** (1), *SLA** (2), *SLA** (3), *SLA** (4) e *SLA** (5). Na Figura 6.1 é apresentado o resumo dos algoritmos que são aplicados nos experimentos.

O *Mea* corresponde ao sistema de planejamento de [Chan et al. 2007b], descritos nas Seções 2.4.1 e 2.4.2. Este procedimento corresponde aplicar seqüencialmente os Algoritmos 4 e 5.

O *MeaPop* consiste no sistema de planejamento proposto no trabalho. Primeiro é aplicado o Algoritmo 8. Em seguida é estabelecida a prioridade (Capítulo 4) e as atividades são ordenadas seqüencialmente pelo valor de prioridade. Ao final o plano seqüencial é escalonado utilizando o Algoritmo 2.4.2.

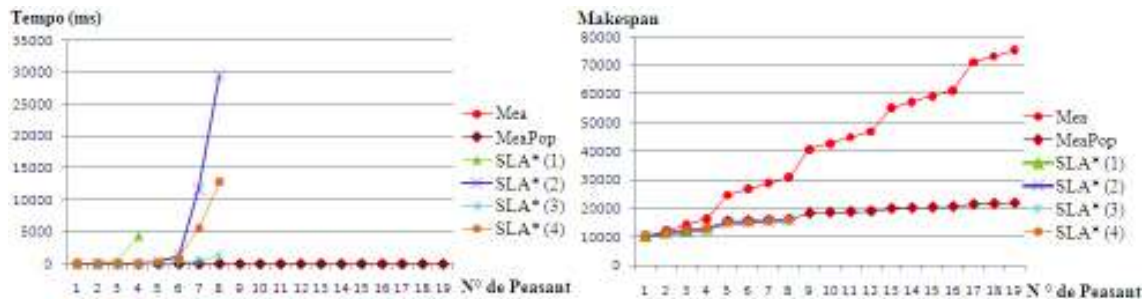
*SLA** (1), *SLA** (2), *SLA** (3), *SLA** (4) e *SLA** (5), dizem respeito ao uso do *SLA** com respeito as estratégias para melhora da eficiência (Seção 5.1.2). Em cada um destes procedimentos é criado o plano de ordem parcial utilizando o Algoritmo 8. Em seguida é estabelecida a prioridade (Capítulo 4) e o plano é escalonado utilizando o *SLA**. Entretanto, no momento do escalonamento são usadas estratégias de eficiência. No *SLA** (1) não é aplicada nenhuma estratégia de eficiência. Para o *SLA** (2), *SLA** (3), *SLA** (4) e *SLA** (5), são usadas as estratégias “Máximo de Ações”, “Menor Tempo” e “Uso de Prioridade”. Sobre o *SLA** (3), *SLA** (4) e *SLA** (5), é usada a estratégia “Limite de Estados” (Seção 5.1.2) com valores respectivos 2, 4 e 6.

O experimento 1 é apresentado na Tabela 6.1. O estado inicial é (*1 townhall, 1 peasant*) e a meta (*1 townhall, 5 peasant*). São realizados 10 execuções para este problema. Cada execução é representada na coluna da esquerda de 1 a 10. Nas colunas seguintes são

Tabela 6.2: Resultados do experimento 2.

	Mea		MeaPop		SLA*(1)	
	tempo	makespan	tempo	makespan	tempo	makespan
1	0,116	10145	0,107	10145	8,401	10145
2	0,141	12185	0,131	11390	62,043	11390
3	0,171	14225	0,164	12185	562,986	12185
4	0,207	16265	0,204	12920	4317,239	12920
5	0,286	24615	0,305	15335	-	-
6	0,338	26655	0,394	15560	-	-
7	0,378	28695	0,437	15785	-	-
8	0,432	30735	0,445	16010	-	-
9	0,539	40655	0,616	18170	-	-
10	0,596	42695	0,666	18395	-	-
11	0,662	44735	0,739	18620	-	-
12	0,762	46775	0,800	18845	-	-
13	0,866	55125	1,003	19790	-	-
14	0,924	57165	1,086	20015	-	-
15	1,106	59205	1,184	20240	-	-
16	1,135	61245	1,280	20465	-	-
17	1,457	71165	1,547	21230	-	-
18	1,326	73205	1,707	21455	-	-
19	1,426	75245	1,750	21680	-	-

apresentados os tempos em milissegundos de cada plano para os respectivos métodos.

Figura 6.2: Gráficos do tempo e *makespan* para o Experimento 2.

Ao final da Tabela 6.1 são colocados respectivamente o tempo médio e o *makespan* obtido por cada procedimento. É importante salientar que para cada procedimento o *makespan* é o mesmo independente da execução.

No experimento 1 são usados os métodos *Mea*, *MeaPop*, SLA*(1), SLA*(2), SLA*(3) e SLA*(4). O pior *makespan* é obtido pelo *Mea*. Entretanto seu tempo é inferior aos outros, com exceção do *MeaPop*. O menor tempo é encontrado pelo *MeaPop*. Os métodos que utilizam o SLA* necessitam de muito tempo. O SLA*(1) não utiliza nenhuma estratégia de eficiência. Em função disto seu tempo é muito superior aos demais. Como não existe limite de estados no SLA*(2) seu tempo é superior ao SLA*(3) e SLA*(4).

Tabela 6.3: Resultados do experimento 2 (continuação).

	SLA*(2)		SLA*(3)		SLA*(4)	
	tempo	makespan	tempo	makespan	tempo	makespan
1	2,397	10145	2,468	10145	2,446	10145
2	3,950	11390	4,002	11390	4,168	11390
3	6,596	12185	6,645	12185	6,616	12185
4	10,245	12920	9,288	12920	10,274	12920
5	285,072	14785	56,169	14785	240,094	14785
6	1312,417	15010	197,677	15185	879,791	15010
7	11664,281	15470	470,724	15510	5629,227	15470
8	29527,353	15785	1244,039	15860	12779,834	15785
9	-	-	-	-	-	-
10	-	-	-	-	-	-
11	-	-	-	-	-	-
12	-	-	-	-	-	-
13	-	-	-	-	-	-
14	-	-	-	-	-	-
15	-	-	-	-	-	-
16	-	-	-	-	-	-
17	-	-	-	-	-	-
18	-	-	-	-	-	-
19	-	-	-	-	-	-

Tabela 6.4: Resultados do experimento 3.

	Mea		MeaPop		SLA*(1)	
	tempo	makespan	tempo	makespan	tempo	makespan
1	0,373	19565	0,319	19565	-	-
2	0,421	18035	0,389	17750	-	-
3	0,497	17635	0,440	17015	-	-
4	0,591	17995	0,483	16730	-	-

Tabela 6.5: Resultados do experimento 3 (continuação).

	SLA*(2)		SLA*(3)		SLA*(4)	
	tempo	makespan	tempo	makespan	tempo	makespan
1	340,059	19525	77,875	19565	829,584	19525
2	-	-	453,100	17710	5299,233	17710
3	-	-	1390,519	16975	-	-
4	-	-	10483,795	16690	-	-

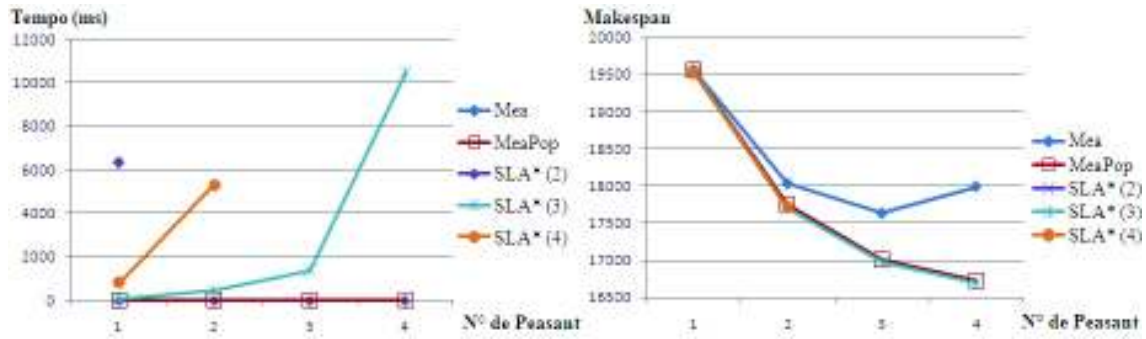


Figura 6.3: Gráficos do tempo e *makespan* para o Experimento 3.

Nas Tabelas 6.2 e 6.3 são apresentados os resultados do experimento 2. Este problema é composto pelo estado inicial (*1 townhall, 1 peasant*). A meta é incremental na quantidade de *peasants*, sendo o limite de 20. A partir do estado inicial temos 19 metas. Para cada um destes problemas são realizados 10 execuções. O resultado de cada problema consiste no tempo médio das 10 execuções e no *makespan*.

Em alguns problemas o SLA* não encontra a solução em tempo satisfatório (inferior a 30 segundos). Porém, em todos os casos em que o SLA* encontrou uma solução o *makespan* é menor ou igual ao *Mea* e *MeaPop*. Os melhores métodos, ao avaliar apenas o *makespan*, são o SLA* (2) e o SLA* (4). Isso demonstra que apesar de necessitar de mais tempo para o planejamento as soluções são melhores. Comparando os tempos do *Mea* e *MeaPop*, observamos que estes valores são muito próximo. Mas de modo geral o *Mea* apresentou melhores resultados ao avaliar este quesito. Entretanto, o *makespan* do *MeaPop* em todos os problemas é melhor que o *Mea*.

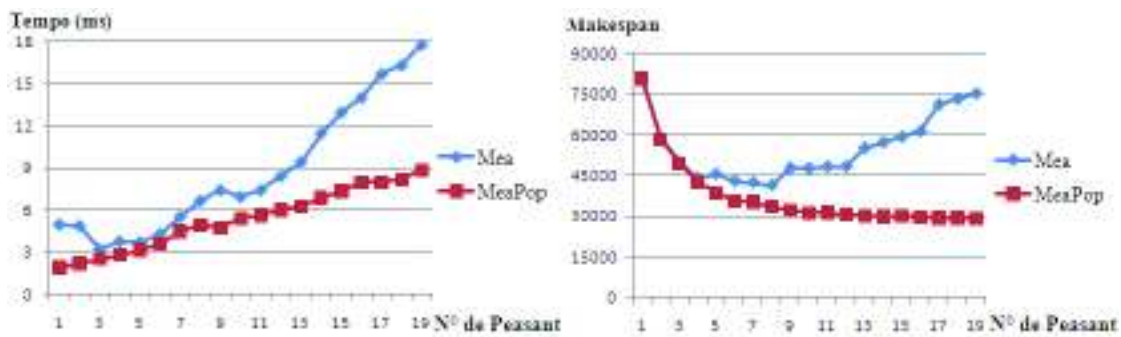


Figura 6.4: Gráficos do tempo e *makespan* para o Experimento 4.

Na Figura 6.2 são apresentados respectivamente os gráficos do tempo e *makespan* para o experimento 2. Nestes gráficos observamos que o tempo dos algoritmos que utilizam o SLA* é muito superior ao *Mea* e *MeaPop*. Considerando o *makespan* observamos que o *Mea* apresenta os piores resultados.

Nas Tabelas 6.4 e 6.5 são apresentados os resultados do experimento 3. Neste problema o estado inicial é (*1 townhall, 1 peasant, 1 barracks*). Na meta o recurso *peasant* é

Tabela 6.6: Resultados do experimento 4.

	Mea		MeaPop	
	tempo	makespan	tempo	makespan
1	4,973	80795	1,903	80640
2	4,878	58700	2,188	58280
3	3,208	50035	2,501	49390
4	3,766	43875	2,806	42665
5	3,690	45445	3,160	38525
6	4,272	42895	3,598	35580
7	5,514	42385	4,476	35205
8	6,661	41325	4,934	33390
9	7,423	47715	4,734	32170
10	6,947	47595	5,414	31225
11	7,404	48195	5,625	31310
12	8,403	48280	6,024	30540
13	9,439	55125	6,273	30120
14	11,477	57165	6,869	29775
15	12,972	59205	7,330	29920
16	14,024	61245	7,975	29595
17	15,699	71165	7,967	29405
18	16,337	73205	8,185	29350
19	17,806	75245	8,827	29230

Tabela 6.7: Resultados do experimento 5.

	Mea		MeaPop		SLA*(3)	
	tempo	makespan	tempo	makespan	tempo	makespan
1	0,641	35645	0,538	35645	80,534	35645
2	0,662	28505	0,669	28280	138,219	28280
3	0,729	25445	0,739	24935	283,033	24935
4	0,841	24200	0,755	22895	560,0176	22895
5	1,243	29715	0,998	23160	-	-
6	1,618	29715	1,318	22385	-	-
7	1,641	29715	1,625	21875	-	-
8	2,202	30735	1,432	21395	-	-
9	2,662	40655	1,861	22140	-	-

incremental a partir do estado inicial, até o limite de 5. A primeira meta é (1 townhall, 1 barracks, 4 footman, 2 peasant), a segunda meta é (1 townhall, 1 barracks, 4 footman, 3 peasant), e assim sucessivamente até o limite de 5 peasants.

Nos casos em que as soluções foram encontradas pelo SLA* o makespan é normalmente melhor que o Mea e MeaPop. É interessante observar que boas soluções são encontradas pelo SLA*(3). Isso mostra que limitar os estados torna o procedimento mais eficiente. Um problema que ocorre nesta estratégia é que para limitar os estados todos os sucessores

Tabela 6.8: Resultados do experimento 5 (continuação).

	SLA*(4)		SLA*(5)	
	tempo	makespan	tempo	makespan
1	78,113	35645	78,158	35645
2	191,839	28280	191,998	28280
3	560,639	24935	566,059	24935
4	4225,079	22895	4294,174	22895
5	-	-	-	-
6	-	-	-	-
7	-	-	-	-
8	-	-	-	-
9	-	-	-	-

devem ser criados na primeira vez que ocorre o passo de seleção do SLA* (linha 7 do Algoritmo 3). Após determinar limitar o número de sucessores, nas próximas verificações os estados não são verificados novamente. Porém, o simples fato de criar todos os estados em algum momento demanda muito tempo. Para determinar se o SLA* é uma boa solução para o planejamento de recursos para jogos RTS, é necessário aplicá-lo em um ambiente real de um jogo RTS. Esta tarefa é parte dos trabalhos futuros.

Na Figura 6.3 são apresentados, respectivamente, os gráficos do tempo e *makespan*. Observamos que criar *peasants* ajuda a atingir a meta principal. Isso ocorre pois criar um *peasant* auxilia a satisfação de outras metas. O quanto antes um *peasant* é criado melhor pode ser o plano.

O experimento 4 é apresentado na Tabela 6.6. Neste problema o estado inicial é (1 *townhall*, 1 *peasant*, 1 *barracks*). A meta consiste em um conjunto de recursos, onde a quantidade de *peasants* é incremental. A primeira meta é (1 *townhall*, 2 *peasant*, 30 *footman*, 1 *barracks*), a segunda (1 *townhall*, 3 *peasant*, 30 *footman*, 1 *barracks*), e assim sucessivamente até o limite de 20 *peasants*. Na Figura 6.4 são apresentados, respectivamente, os gráficos de tempo e *makespan* do experimento 4.

O objetivo do experimento 4 consiste em comparar os métodos *Mea* e *MeaPop*. Neste problema todas as metas apresentaram melhores resultados ao usar o *MeaPop*. Em muitas metas o tempo obtido pelo *MeaPop* é significativamente inferior ao *Mea*. O *makespan* do *MeaPop* é cada vez menor devido a quantidade de *peasants* que são criados. Os *peasants* criados auxiliam a execução de outras ações. Entretanto, a quantidade de *peasants* não pode crescer indefinidamente, pois em algum momento o tempo gasto para criar *peasants* não compensará o tempo gasto.

Nas Tabelas 6.7 e 6.8 são apresentados os resultados do experimento 5. Neste problema o estado inicial é (1 *townhall*, 1 *peasant*) e a meta é incremental na quantidade de *peasants*. A primeira meta é (1 *townhall*, 10000 *gold*, 2 *peasant*), a segunda é (1 *townhall*, 10000 *gold*, 3 *peasant*), e assim sucessivamente até a quantidade de 10 *peasants*.

Tabela 6.9: Resultados do experimento 6.

	Mea		MeaPop	
	tempo	makespan	tempo	makespan
1	2,767	137645	2,289	137645
2	3,008	96620	2,897	96335
3	2,814	76445	2,768	75935
4	2,695	65000	2,744	63695
5	2,892	63885	3,790	57065
6	3,152	58785	4,202	51455
7	3,477	55215	4,681	47375
8	4,170	52950	5,175	44090
9	4,520	59015	6,225	42540
10	5,188	57770	6,360	40520
11	6,236	56975	6,797	38960
12	7,452	56465	7,210	37530
13	8,578	62265	8,444	36920
14	9,890	62040	9,243	35910
15	11,924	61755	9,459	35040
16	13,596	62265	10,085	34370
17	15,598	71165	10,412	34205
18	17,845	73205	10,438	33675
19	20,664	75245	10,483	33185
20	22,266	77285	10,969	32735
21	28,049	85635	11,286	32675
22	26,551	87675	11,088	32315
23	28,995	89715	11,137	31980
24	30,964	91755	11,791	31745

Os métodos que utilizam o algoritmo SLA* não encontraram solução em alguns problemas. Porém, nos casos em que a solução foi encontrada o *makespan* é melhor ou igual aos demais. Analisando o tempo do *MeaPop* observa-se que seu tempo para encontrar as soluções em geral é inferior ao *Mea*.

Os resultados do experimento 6 são apresentados na Tabela 6.9. Assim como o experimento 4, neste problema são usados os métodos *Mea* e *MeaPop*. O estado inicial é (1 *townhall*, 1 *peasant*) e a meta é incremental na quantidade de *peasants*. A primeira meta é (1 *townhall*, 2 *peasant*, 50000 *gold*), a segunda meta é (1 *townhall*, 3 *peasant*, 50000 *gold*), e assim sucessivamente até a quantidade de 25 *peasants*. Os resultados deste problema demonstram que o *MeaPop* novamente obteve melhores valores de *makespan*. Entretanto, neste caso o tempo de processamento de alguns casos do *MeaPop* é ligeiramente inferior ao *Mea*. Entretanto, nas metas que necessitam criar mais *peasants* o *MeaPop* obteve melhores resultados. Na Figura 6.5 são apresentados os gráficos de tempo e *makespan* do experimento 6.

Na Tabela 6.10 são apresentados os resultados do experimento 7. Neste problema são



Figura 6.5: Gráficos do tempo e *makespan* para o Experimento 6.

Tabela 6.10: Resultados do experimento 7.

	Mea		MeaPop	
	tempo	makespan	tempo	makespan
1	0,649	88645	0,531	88645
2	0,737	63995	0,657	64320
3	0,884	52535	0,806	51435
4	1,025	45870	0,900	44320
5	1,446	47940	0,993	41180
6	1,417	45270	1,258	38040
7	1,621	43615	1,344	35520
8	1,876	42515	1,393	34085
9	2,437	49765	1,749	33310

usados os métodos *Mea* e *MeaPop*. O estado inicial é (1 *townhall*, 1 *peasant*) e a meta é incremental na quantidade de *peasants*. A primeira meta é (1 *townhall*, 2 *peasant*, 10000 *wood*), a segunda meta é (1 *townhall*, 3 *peasant*, 10000 *wood*), e assim sucessivamente até a quantidade de 10 *peasants*. Os resultados demonstram que o *MeaPop* novamente encontrou melhores valores de *makespan* e tempo de processamento.

A Tabela 6.11 apresenta os resultados do experimento 8. Neste problema são usados os métodos *Mea* e *MeaPop*. O estado inicial é (1 *townhall*, 1 *peasant*) e a meta é incremental na quantidade de *peasants*. A primeira meta é (1 *townhall*, 2 *peasant*, 10000 *gold*, 5000 *wood*), a segunda meta é (1 *townhall*, 3 *peasant*, 10000 *gold*, 5000 *wood*), e assim sucessivamente até a quantidade de 10 *peasants*. Os resultados demonstram que o *MeaPop* novamente encontrou melhores valores de *makespan* e tempo de processamento.

Analisando todos os experimentos observa-se que os melhores resultados de *makespan* são encontrados pelo método *MeaPop*. Entretanto, no momento que os algoritmos SLA* encontram uma solução o valor de *makespan* é sempre melhor ou igual aos demais procedimentos. Porém, em muitos casos a solução não foi encontrada em tempo satisfatório ao usar o SLA*. Nos primeiros experimentos o *Mea* demandou menos tempo que o *MeaPop*. Contudo, em problemas mais complexos (com mais recursos) o *MeaPop* obteve melhores

Tabela 6.11: Resultados do experimento 8.

	Mea		MeaPop	
	tempo	makespan	tempo	makespan
1	1,010	74895	0,779	74895
2	1,106	55195	0,960	54910
3	1,179	45855	0,994	45060
4	1,095	39900	0,989	38595
5	1,397	43845	1,383	36890
6	1,674	41765	1,652	34435
7	2,556	40705	1,776	32385
8	2,450	39135	2,285	30590
9	2,960	47015	2,822	29990

resultados.

Capítulo 7

Conclusão e Trabalhos Futuros

Nesta dissertação foram propostos algoritmos de planejamento de recursos para jogos RTS. Esta tarefa de planejamento é recente e existem poucos trabalhos que tratam deste problema. O sistema proposto é dividido em três componentes. O primeiro consiste em um algoritmo de planejamento de ordem parcial. O segundo é composto pelo estabelecimento de prioridade para as atividades pertencentes ao plano de ordem parcial. Ao final é apresentado um algoritmo de busca e aprendizado para realizar a tarefa de escalonamento.

Analisando os resultados, observamos que o planejador *MeaPop* apresentou melhores resultados quando comparado com o planejador seqüencial proposto por [Chan et al. 2007b]. Além da possibilidade de priorizar as ações mais importantes, usar o planejamento de ordem parcial permitiu a aplicação dos algoritmos de busca e aprendizado, tal como o SLA*.

Apesar do SLA* ter demandado grande tempo para resolver os experimentos, é importante destacar que este algoritmo possibilitou o uso de busca e aprendizado em tempo real. Essa característica é importante em um ambiente real de um jogo pois permite a melhora contínua da solução e a possibilidade de se obter uma determinada solução dentro de um intervalo de tempo, dotando assim o sistema de reatividade.

Como trabalhos futuros são propostas diversas melhorias e testes. Entre estes trabalhos temos:

1. Desenvolvimento do sistema de planejamento proposto nesta dissertação em um ambiente real de jogo RTS. Isso possibilitará realizar experimentos mais precisos e usufruir da ferramenta de busca e aprendizado do algoritmo SLA*.
2. Pesquisa de outros algoritmos de busca e aprendizado, tais como: LRTA*, SLA*T e LRTA*-LS.
3. Estudo de melhorias no planejamento de ordem parcial, de modo que não seja necessário especificar a ordem que os recursos são satisfeitos.
4. O procedimento de estabelecimento de prioridade normalmente é executado no melhor caso nos planos de ordem parcial obtidos neste trabalho. Entretanto, acredita-se

que seja possível melhorar o pior caso deste método. Sendo assim, um dos trabalhos futuros consiste em propor um algoritmo mais eficiente para tal tarefa.

5. Além dos algoritmos de busca e aprendizado em tempo real aplicados no escalonamento deve se investigar a aplicação de outros métodos que realizem essa tarefa. Métodos utilizando algoritmos genéticos, por exemplo, podem ser aplicados para resolver o problema de escalonamento.
6. Pesquisa de métodos para melhorar o planejamento em mapas dos jogos RTS. Ou seja, mapas que sofrem alterações com o decorrer do jogo.

Referências Bibliográficas

- [Adams 2006] Adams, D. (2006). The State of the RTS. IGN PC.com. Disponível em: <http://pc.ign.com/articles/700/700747p1.html>.
- [Ahsan e Tsao 2003] Ahsan, M. K. e Tsao, D. (2003). A Comment on Solving Project-Scheduling Problems with a Heuristic Learning Algorithm. *Journal of the Operational Research Society*, 54:666–668.
- [Anderson et al. 1998] Anderson, C. R., Smith, D. E., e Weld, D. S. (1998). Conditional Effects in Graphplan. In *AIPS*, pp. 44–53.
- [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.
- [Branquinho e Lopes 2009] Branquinho, A. A. B. e Lopes, C. R. (2009). Uma Abordagem de Planejamento de Recursos para Jogos do Tipo RTS Baseada em Planejamento de Ordem Parcial. *XXXV Conferência Latino Americana de Informática (CLEI)*.
- [Bulitko e Lee 2006] Bulitko, V. e Lee, G. (2006). Learning in Real-Time Search: A Unifying Framework. *J. Artif. Intell. Res. (JAIR)*, 25:119–157.
- [Chan et al. 2007a] Chan, H., Fern, A., Ray, S., Wilson, N., e Ventura, C. (2007a). Extending Online Planning for Resource Production in Real-Time Strategy Games with Search. *Workshop on Planning in Games, ICAPS*.
- [Chan et al. 2007b] Chan, H., Fern, A., Ray, S., Wilson, N., e Ventura, C. (2007b). Online Planning for Resource Production in Real-Time Strategy Games. In Boddy, M. S., Fox, M., e Thiébaux, S. (editores), *ICAPS*, pp. 65–72. AAAI.
- [Cormen et al. 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., e Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, 2 edition.
- [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.
- [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.
- [Furcy e Koenig 2000] Furcy, D. e Koenig, S. (2000). Speeding up the Convergence of Real-Time Search. In *In Proceedings of the National Conference on Artificial Intelligence*, pp. 891–897.

- [Furcy 2004] Furcy, D. A. (2004). *Speeding Up the Convergence of Online Heuristic Search and Scaling Up Offline Heuristic Search*. PhD thesis, Georgia Institute of Technology.
- [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.
- [Hernández e Meseguer 2007] Hernández, C. e Meseguer, P. (2007). Improving LRTA*(k). In *IJCAI'07: Proceedings of the 20th international joint conference on Artificial intelligence*, pp. 2312–2317, Hyderabad, India, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- [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.
- [Knuth et al. 1989] Knuth, D. E., Graham, R. L., e Patashnik, O. (1989). *Concrete Mathematics: a Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [Korf 1985] Korf, R. E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109.
- [Korf 1990] Korf, R. E. (1990). Real-Time Heuristic Search. *Artificial Intelligence*, 42(2-3):189–211.
- [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.
- [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.
- [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.
- [Rintanen et al. 2005] Rintanen, J., Rintanen, J., e Albert-ludwigs (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.
- [Russell e Norvig 1995] Russell, S. J. e Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ.
- [Sacerdoti 1975] Sacerdoti, E. D. (1975). The Nonlinear Nature of Plans. In *IJCAI*, pp. 206–214.
- [Shue e Li 2008] Shue, L.-Y. e Li, S.-T. (2008). A Generalized Heuristic Learning Approach to Project Scheduling Problems with Resource Constraints. *Journal of the Chinese Institute of Industrial Engineers (TSSCI, EI)*, 25(3):204–214.
- [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.
- [Zamani e Shue 1998] Zamani, R. e Shue, L.-Y. (1998). Solving Project Scheduling Problems with a Heuristic Learning Algorithm. *Journal of the Operational Research Society*, 49:709–716.
- [Zamani e Shue 2003] Zamani, R. e Shue, L.-Y. (2003). Reply to MK Ahsan and D-b Tsao. *Journal of the Operational Research Society*, 54:668–670.

Apêndice A

Somatórios

Este apêndice tem como base [Cormen et al. 2001] e [Knuth et al. 1989]. Seu objetivo é descrever os conceitos básicos dos somatórios usados nesta dissertação.

Um somatório é composto por uma sequência $a_m, a_{m+1}, a_{m+2}, \dots, a_{n-1}, a_n$ de números, onde a soma destes elementos $a_m + a_{m+1} + a_{m+2} + \dots + a_{n-1} + a_n$, é expressa pela seguinte notação:

$$\sum_{i=m}^n a_i \quad (\text{A.1})$$

Se $m > n$, o valor do somatório é definido como 0. O valor de uma série finita é sempre bem definido e seus termos podem ser somados em qualquer ordem. Normalmente, os termos m e n , são chamados de limite inferior e limite superior, respectivamente.

Linearidade

Para qualquer número constante real c e sequência finita $a_m, a_{m+1}, \dots, a_{n-1}, a_n$, temos a seguinte igualdade:

$$\sum_{i=m}^n ca_i = c \sum_{i=m}^n a_i \quad (\text{A.2})$$

Se considerarmos as seqüências finitas $a_m, a_{m+1}, \dots, a_{n-1}, a_n$ e $b_m, b_{m+1}, \dots, b_{n-1}, b_n$, temos a igualdade:

$$\sum_{i=m}^n (a_i + b_i) = \sum_{i=m}^n a_i + \sum_{i=m}^n b_i \quad (\text{A.3})$$

Série Aritmética

O somatório

$$\sum_{i=1}^n i = 1 + 2 + \dots + n \quad (\text{A.4})$$

é uma série aritmética e tem o seguinte valor (Fórmula Fechada)

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad (\text{A.5})$$

Somas de Quadrados

O somatório

$$\sum_{i=1}^n i^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 \quad (\text{A.6})$$

tem o seguinte valor (Fórmula Fechada)

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad (\text{A.7})$$

Apêndice B

Prova da Solução Ótima do SLA*

A seguir é apresentada a prova que o SLA* encontra a solução ótima.

Teorema B.1. *Para um problema com espaço de busca finito, arestas de custo positivo, heurística inicial não super estimada (ou seja, $\forall s \in S, h_0(s) \leq h^*(s)$, onde: $h_0(s)$ é a heurística inicial e $h^*(s)$ é a heurística ótima) e meta que pode ser alcançada a partir do estado inicial, a aplicação do SLA* encontra o caminho de menor custo.*

Prova. A prova deste teorema é feita por contradição assumindo que existe um caminho P' até a meta com um custo melhor que o caminho P encontrado pelo SLA*. Para tal prova são feitas as seguintes considerações:

- Os estados do caminho P são denotados por $x(1), x(2), \dots, x(n-1), x(n)$.
- Os estados do caminho P' são denotados por $y(1), y(2), \dots, y(q-1), y(q)$.
- O estado inicial é o mesmo para ambos os caminhos, representado pelos elementos $x(1)$ e $y(1)$.
- A meta é a mesma para ambos os caminhos, representada pelos elementos $x(n)$ e $y(q)$.
- Sendo $y(m)$ o primeiro estado de P' que não se encontra em P , então, para ambos os estados $x(m)$ e $y(m)$ o vizinho que os antecede $x(m-1)$ é o mesmo.
- A variável k representa o custo de um estado até o seu vizinho.
- Sendo um caminho ótimo C , denotado $c(1), c(2), \dots, c(n-1), c(n)$ e custo k entre os estados vizinhos. Temos que o valor de heurística $h(i)$ nunca é maior que o somatório dos custos $k\{c(i), c(i+1)\} + \dots + k\{c(n-1), c(n)\}$.

A relação B.1 deve ser satisfeita para que o custo de P' seja inferior a P .

$$\begin{aligned} & k\{x(m-1), y(m)\} + k\{y(m), y(m+1)\} + \dots + k\{y(q-1), y(q)\} < \\ & k\{x(m-1), x(m)\} + k\{x(m), x(m+1)\} + \dots + k\{x(n-1), x(n)\} \end{aligned} \tag{B.1}$$

Essa relação pode ser modificada e expressa em B.2 ao considerar que a heurística $h(y(m))$ jamais é maior que a soma dos custos do estado $y(m)$ até a meta. Para facilitar a representação da inequação B.1 também é usada uma soma para o custo do caminho de P .

$$k\{x(m-1), y(m)\} + h\{y(m)\} < \sum_{i=m}^n k\{x(i-1), x(i)\} \quad (\text{B.2})$$

Como indicado no passo 8 do algoritmo SLA*, a seguinte equação B.3 é sempre verdadeira para um estado r .

$$\begin{aligned} h\{x(r)\} &\geq k\{x(r), x(r+1)\} + h\{x(r+1)\} \Leftrightarrow \\ h\{x(r)\} - h\{x(r+1)\} &\geq k\{x(r), x(r+1)\} \end{aligned} \quad (\text{B.3})$$

Através da soma da inequação B.3 para o caminho P de $x(m)$ até a meta $x(n)$ é obtida a seguinte inequação:

$$\begin{aligned} h\{x(m)\} - h\{x(n)\} &\geq k\{x(m), x(m+1)\} + \dots + k\{x(n-1), x(n)\} \Leftrightarrow \\ h\{x(m)\} - h\{x(n)\} &\geq \sum_{i=m}^{n-1} k\{x(i), x(i+1)\} \end{aligned} \quad (\text{B.4})$$

Como a heurística do estado inicial é 0, ou seja, $h\{x(n)\} = 0$, temos a seguinte expressão:

$$h\{x(m)\} \geq \sum_{i=m}^{n-1} k\{x(i), x(i+1)\} \quad (\text{B.5})$$

Seja o estado $x(m-1)$ e os sucessores $x(m)$ e $y(m)$, pelo passo 7 do SLA*, consideramos que o estado $x(m)$ é escolhido. A partir destas considerações, podemos formular a seguinte expressão:

$$k\{x(m-1), y(m)\} + h\{y(m)\} \geq k\{x(m-1), m(m)\} + h\{x(m)\} \quad (\text{B.6})$$

Aplicando a substituição do termo $h\{x(m)\}$ da inequação B.5 em B.6, encontramos a expressão:

$$\begin{aligned} k\{x(m-1), y(m)\} + h\{y(m)\} &\geq k\{x(m-1), m(m)\} + \sum_{i=m}^{n-1} k\{x(i), x(i+1)\} \Leftrightarrow \\ k\{x(m-1), y(m)\} + h\{y(m)\} &\geq \sum_{i=m}^n k\{x(i-1), x(i)\} \end{aligned} \quad (\text{B.7})$$

Analisando as expressões B.2 e B.7, fica clara a contradição ao assumir que o caminho

P' possui um melhor caminho que P . Deste modo, o caminho P' não existe e provamos que o algoritmo SLA^* encontra a melhor solução.