MAC0499 - Trabalho de Formatura Supervisionado

Técnicas de Inteligência Artificial para resolução do Sokoban

Aluna: Jacqueline Rodrigues
Orientadora: Leliane de Nunes Barros

Resumo: O jogo Sokoban é consiste de um único agente que deve empurrar caixas para áreas de armazenamento num ambiente formado por um labirinto de salas e corredores. Dentre as dificuldades desse jogo estão: (i) evitar uma grande quantidade de deadlocks (becos-sem-saida) e (ii) a necessidade de evitar conflitos entre os movimentos de caixas (uma vez que no Sokoban, a estratégia de divide-and-conquer não pode ser aplicada diretamente). Nesse trabalho, apresentamos um estudo sobre as soluções desse jogo encontradas na literatura e a implementação de um agente Sokoban capaz de resolver um conjunto de 61 problemas, chamados de Kids Benchmark, que incorporam as principais dificuldades desse jogo.

1. Introdução

Sokoban é um jogo popular em terceira pessoa, implementado em quase todos os sistemas operacionais. Foi inventado nos anos 80 no Japão por Hiroyuki Imabayashi. O jogo consiste em um tabuleiro de tamanho n X m, $n \leq 20$ e $m \leq 20$, contendo salas, passagens e corredores. Nesse tabuleiro encontram-se caixas, áreas de armazenamento e um agente chamado sokoban (em japonês, carregador de caixas). O objetivo desse jogo é fazer com que o seu personagem, o sokoban, empurre todas as caixas para os lugares de armazenamento. O sokoban pode fazer apenas movimentos básicos como mover-se para cima, para baixo, para direita e para esquerda. Combinado com esses movimentos, o agente pode também empurrar uma caixa por vez, para cima, para baixo, para direita e para esquerda, mas nunca puxar uma caixa. O problema é resolvido quando todas as caixas estiverem armazenadas. Existe ainda a noção de solução ótima: a solução encontrada com o menor número de movimentos do agente e das caixas.

A dificuldade principal em resolver um jogo de Sokoban é de que movimentos errados podem levar o agente a colocar caixas em lugares que nunca mais conseguirão ser retiradas, isto é, deadlocks ou becos-sem-saída. Por exemplo: uma caixa localizada em um canto do armazém (Figura 1) ou encostada numa parede externa do armazém (Figura 2). Além disso, uma caixa pode eventualmente bloquear a outra, dado que o agente não pode empurrar mais de uma caixa por vez (Figura 3). Por esse motivo, o agente deve levar em consideração a ordem em que ele empurra as caixas: idéia fundamental para se encontrar uma solução ótima ou sub-ótima.

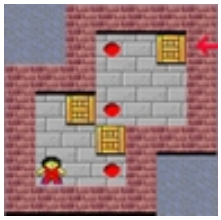


Figura 1: deadlock de canto

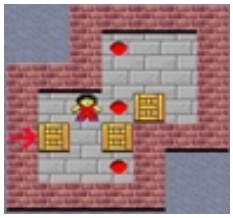


Figura 2: deadlock de parede

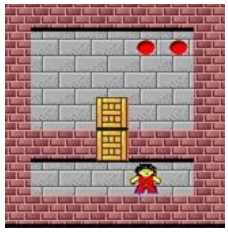


Figura 3: deadlock entre caixas

O Sokoban é um jogo interessante pois apesar da simplicidade de suas regras existem tabuleiros de grande complexidade. Por esse motivo, este jogo tem despertado a atenção de pesquisadores na área de jogos em Inteligência Artificial, como um domínio de grande interesse para a aplicação de técnicas de busca heurística. Existe uma coleção de 90 tabuleiros de Sokoban usados como referência para avaliar a eficiência dos algoritmos propostos para automatizar a solução desse jogo e que chamaremos de ***Problemas Sokoban Benchmark***. A grande maioria desses tabuleiros foram desenvolvidos pela empresa Spectrum Holobyte em 1984 [10].

*Exemplo 1: Considere o tabuleiro de Sokoban da Figura 4 (chamamos esse tabuleiro de **estado inicial**). A solução desse problema está ilustrada na Figura 5 (que chamaremos de **estado objetivo**). O problema de Sokoban visto como um problema de busca e encontrar uma sequencia de movimento (do sokoban e das caixas) que levam o agente do estado inicial para o estado objetivo.*

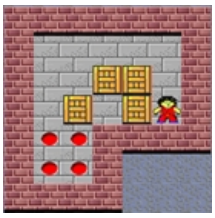


Figura 4: Estado inicial

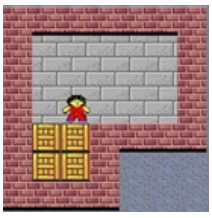


Figura 5: Estado final

O tabuleiro da Figura 4 pode ser representado por uma matriz 7X7 :

```
  ABCDEFG
a #######
b #     #
c #  $$ #
d # $ $@#
e #..####
f #..#
g ####
```

sendo que o símbolo "#", *representa uma parede; "*$" *representa uma caixa; "*.", *representa uma meta de armazenamento e "*@", *representa o agente. Cada elemento dentro do tabuleiro pode ser localizado por suas coordenadas, o agente na Figura 4 está na posição Fd.*

2. Proposta

A proposta desse trabalho é desenvolver um algoritmo que resolva problemas de Sokoban. Para isso, esse trabalho apresenta um estudo das principais propostas encontradas na literatura, aponta as principais dificuldades do problema e as técnicas mais promissoras para resolvê-las. É proposto um algoritmo que usa duas técnicas estudadas: busca heurística e iterativa em profundidade (IDA*) e detecção de ciclos; e uma adaptação de algumas outras técnicas estudadas para detecção de deadlocks.

O sistema desenvolvido, chamado de **SokoKids**, foi testado em um conjunto de 61 problemas, chamados de ***Problemas Kids Benchmark*** [1] (Apêndice A), representando situações consideradas difíceis para o Sokoban e que quando combinadas em tabuleiros maiores, fazem com que o jogo seja considerado um desafio na área de jogos em Inteligência Artificial.

3. Algoritmos de Busca

Para resolver o Sokoban, antes de estudar as técnicas específicas para esse domínio, é necessário relembrar o que são algoritmos de busca para jogos [5]. De um modo geral, algoritmos de busca são algoritmos que recebem como entrada um problema e devolvem uma solução. O conjunto de todas as soluções possíveis é chamado de espaço de busca. Dado um espaço de busca é possível que, dependendo do jogo, possam existir várias árvores de busca, cada uma contendo na raiz um estado inicial diferente.

Algoritmos de *busca exaustiva,* também conhecidos por busca não-informada, não consideram as

particularidades e características de cada problema, isto é, a mesma implementação pode ser usada para resolver vários problemas diferentes. Por esse motivo, o espaço de busca percorrido por um algoritmo de busca não-informada é geralmente muito grande. Por outro lado, as buscas informadas usam funções heurísticas para aplicar o conhecimento adquirido sobre um problema específico e percorrem de forma mais eficiente o espaço de busca, reduzindo assim o tempo de busca nessa estrutura.

O sucesso de uma busca depende de sua habilidade em visitar a maior parte **relevante** do espaço de busca do problema. Para isso, é necessário direcionar muito bem a busca e podar partes irrelevantes do espaço de busca. Mesmo quando isso é feito, os computadores ainda falham quando comparados com os seres humanos. Em geral, humanos são melhores em aprender e usar o conhecimento adquirido para diminuir o espaço de busca. [1]. Isso sugere desenvolver métodos dinâmicos que aprendam aspectos específicos do problema que está sendo resolvido e apliquem esse conhecimento aprendido, nos próximos passos de resolução ou na solução de problemas futuros. Essa estratégia pode ser chamada de aprendizado. Porém, nesse trabalho, não estudaremos técnicas de aprendizado de máquina mas somente as técnicas de busca que utilizam conhecimento a priori, tanto conhecimento de busca geral (busca exaustiva ou não-informada), como conhecimento específico do domínio do problema (busca informada ou busca heurística) para direcionar a busca mais rapidamente para uma solução (ótima ou sub-ótima).

3.1 Busca exaustiva

Busca exaustiva, também chamada de busca não-informada ou busca cega, não possui informação adicional sobre quais são os estados mais relevantes para se encontrar uma solução. São algoritmos de busca generalistas e podem ser aplicados a diversos domínios com a penalidade de ter que percorrer o espaço completo de busca (no pior caso). A operação básica neste tipo de busca é gerar estados sucessores, que chamaremos de ***expansão do nó pai***, e distinguir um estado objetivo de um estado não objetivo.

> *Exemplo 2: Quando o nó pai é expandido, todos os movimentos possíveis são considerados para gerar seus estados sucessores. Para isso, é feita uma tentativa de mover cada caixa existente no tabuleiro para as quatro direções possíveis, para cima, para baixo, para a esquerda e para a direita. No exemplo da Figura 6, por motivo de simplificação, mostramos os quatro estados sucessores considerando apenas as movimentações das caixas, sem considerarmos as trajetórias do agente para alcançar a posição correta para empurrá-las. Assim, a expansão do estado inicial gera apenas 4 estados sucessores pois não é possível mover as caixas para os lados. As arestas da árvore representam as ações ou movimentos no jogo. Em geral, associa-se às arestas uma função custo. Como no Sokoban todas as ações tem custo 1, as ações, a1, a2, a3 e a4 possuem o mesmo custo.*

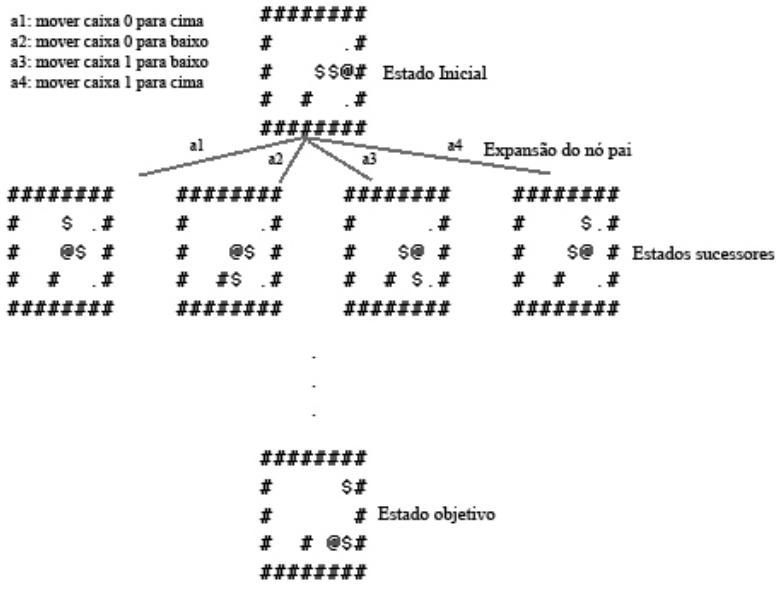


Figura 6: Exemplo de expansão do estado inicial em seus estados sucessores para a construção da árvore de busca.

3.1.2 Busca em Largura

A busca em largura [5] é uma estratégia simples em que o nó da raiz é expandido primeiro, em seguida todos os sucessores do nó raiz são expandidos, depois os sucessores desses nós e assim por diante. Em geral, todos os nós em uma dada profundidade na árvore de busca são expandidos, antes que algum nó no nível seguinte seja expandido. Dessa maneira todos os nós que estão mais próximos da raiz são expandidos antes de visitar aqueles que estão mais profundos. Essa estratégia pode ser usada quando é esperado que o estado objetivo esteja próximo, isto é, em uma altura não muito profunda da árvore.

A Figura 8 ilustra os passos da busca em largura na árvore de busca representada pela Figura 7. É importante notar que a árvore completa de busca da Figura 7 não é dada explicitamente como entrada, isto é, ela é construída passo a passo, através da busca em largura como mostrado na Figura 8. Assim, caso a solução fosse encontrada no passo 5, a árvore completa deixaria de ser expandida. Essa é a principal diferença das técnicas de busca para jogos e busca em grafos.

```
buscaEmLargura(estadoInicial) {
       armazenar(fila, estadoInicial);
       sucesso = false;
       faça {
            estadoAtual = pegarPrimeiro(fila);
            se (solucao(estadoAtual))
            sucesso = true;
            senão
            para cada ( filho(estadoATual)) faça
                  inserir(fila, filho(estadoAtual));
       } até ( sucesso OU vazio(fila));
       se (sucesso) devolve (estadoAtual);
       se não retorne (NULO); }
```

Algoritmo 1: Pseudo-código da busca em largura. Adaptada de [1]

O Algoritmo 1 começa armazenando a raiz da árvore, chamada de *estadoInicial,* em *fila*. Depois, para cada estado que está na *fila*, é feito um teste para verificar se aquele estado é a solução do problema. Essa verificação é feita por meio da função *solucao.* Caso o estado testado seja a solução, o algoritmo devolve esse estado, caso contrário, é feita a expansão desse estado adicionando todos os estados sucessores gerados na *fila.* A busca em largura continua fazendo isso até encontrar o estado objetivo ou até ter explorado todos os estados possíveis, ou seja, quando a variável *fila* estiver vazia. Nesse caso, o problema não tem solução.

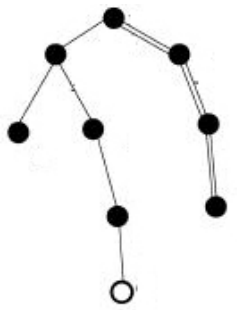


Figura 7: Árvore completa da busca em largura

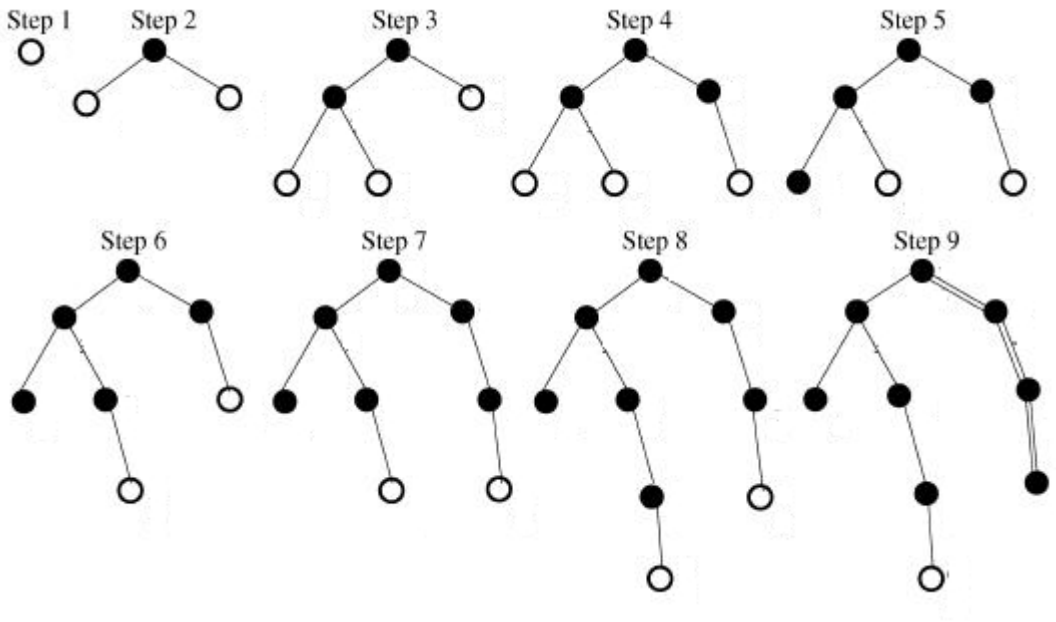


Figura 8: Ilustração de uma busca em largura. Adaptada de [1]

.

O Algoritmo 1 implementa a busca em largura e garante que a solução achada é ótima, desde que todas as arestas (ações) tenham o mesmo custo. O número total de nós gerados é $O(b^{d+1})$, onde b representa o fator de ramificação médio da árvore e d a altura da árvore em que a solução ótima foi encontrada.

3.1.3 Busca em Profundidade

A busca em profundidade [5] atravessa a árvore de busca de cima abaixo, sempre expandindo o nó mais mais profundo na borda atual da árvore de busca até o nível mais profundo da árvore de busca, onde os nós não tem sucessores. A medida em que esses nós são expandidos, eles são retirados da borda e então a busca retrocede ao nó seguinte mais raso que ainda possui sucessores não expandidos. A Figura 9 ilustra passo a passo a construção da árvore de busca quando uma busca em profundidade é realizada.

A vantagem da busca em profundidade com relação a busca em largura é que ela tem requisitos de memória muito modestos: só é preciso armazenar um único caminho da raiz até a folha, juntamente com os nós irmãos não expandidos restantes de cada nó do caminho. Uma vez que um nó é expandido ele pode ser removido da memória, tão logo todos os seus descendentes tenham sido completamente explorados.

```
buscaEmProfundidade(estadoInicial){
      inserir(pilha,estadoInicial);
      sucesso=false;
      faça{
            estadoAtual = pegarTopo(pilha);
            se ( solucao(estadoAtual))
                  sucesso = true;
            senão para cada ( filho(estadoATual)) faça
                  inserir(pilha, filho(estadoAtual));
      } até ( sucesso OU vazio(pilha));
      se (sucesso) retorne (estadoAtual);
      senão retorne (NULO);
      }
```

Algoritmo 2: pseudo-código da busca em profundidade (figura daptada de [1])

O Algoritmo 2 insere o estado inicial, a raiz da árvore, em *pilha*. Em seguida, verifica se esse estado, isto é, a variável *estadoAtual,* é a solução do problema. Caso isso seja verdade, o algoritmo devolve esse estado, caso contrário, faz a expansão desse estado adicionando todos os filhos gerados em *pilha*. Essa verificação é feita até encontrar a solução ou a pilha ficar vazia.

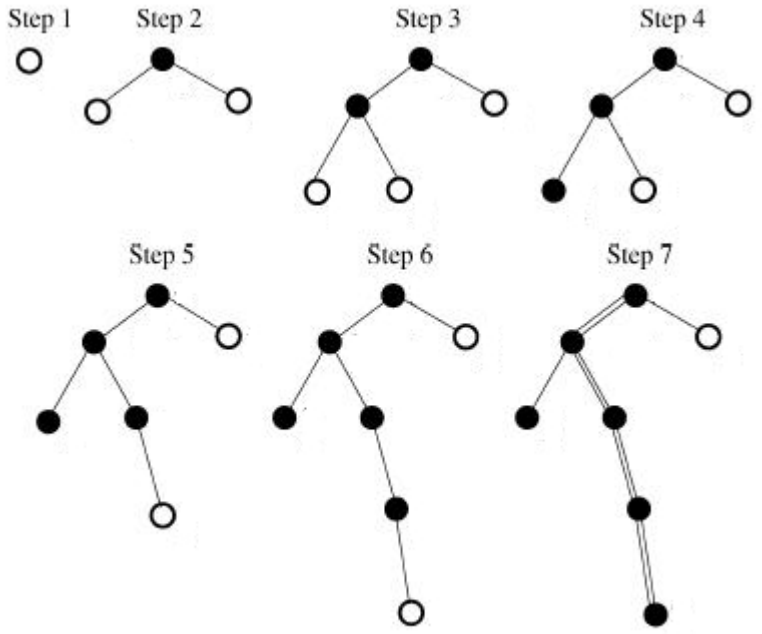


Figura 9: Ilustração da busca em profundidade. Adaptada de [1]

A desvantagem da busca em profundidade é a possibilidade de se perder em ciclos e caminhos infinitos. O número total de nós gerados é O(*bm*), onde *b* representa o fator médio de ramificação da árvore e *m* a profundidade máxima da árvore [5].

3.1.4 Busca em profundidade Iterativa

A busca em profundidade iterativa [5] é uma estratégia de busca exaustiva, usada com freqüência em combinação com a busca em profundidade, em que encontra o melhor limite de profundidade. Ela faz isso aumentando gradualmente o limite, primeiro 0, depois 1, depois 2 e assim por diante, até encontrar um estado objetivo. Isso ocorrerá quando o limite de profundidade alcançar d, a profundidade do nó objetivo mais raso. A busca em profundidade iterativa combina os benefícios da busca em profundidade e da busca em largura. Como na busca em profundidade, seus requisitos de memória são muitos modestos. Como na busca em largura, ele é completo quando o fator de ramificação é finito, e ótimo quando o custo do caminho é uma função não decrescente da profundidade do nó.

A busca em profundidade iterativa pode parecer um desperdício, porque os mesmos estados são gerados várias vezes. Na verdade, esse custo não é muito alto porque, em uma árvore de busca com o mesmo (ou quase mesmo) fator de ramificação, pois a maior parte dos nós estará no nível inferior. Dessa maneira, não importa se os níveis superiores são gerados várias vezes. O número total de nós gerados é: $N = (d)b + (d-1)b^2+...+(1)b^d$. Assim, como podemos notar, a complexidade da busca em profundidade iterativa é a mesma que a da busca em largura, isto é, $O(b^d)$, onde *b* representa o fator de ramificação médio da árvore e *d* a altura da árvore em que a solução ótima foi encontrada. No entanto, a complexidade de memória é a mesma que a da busca em profundidade, ou seja, O(*bh*), onde *b* representa o fator médio de ramificação da árvore e *h* a altura da árvore em que a solução ótima foi encontrada [5].

3.2 Buscas Informadas

Todas as buscas apresentadas anteriormente não possuem informações sobre a utilidade, isto é, o quão importante é o nó para se encontrar a solução, a não ser quantos passos foram necessários para se alcançar aquele estado. Buscas informadas, ao contrário, conseguem estimar quantos estados faltam para chegar do estado atual até um estado objetivo. Para isso, a busca informada usa uma função de avaliação ou função heurística. Essa função de avaliação é chamada de admissível se ela nunca superestimar a distância real até o estado solução.

Uma maneira para definir uma função heurística admissível é relaxar as restrições do problema. Dessa maneira, a heurística calculará uma estimativa otimista da distância até o estado solução. Porém, quanto menos restrições do domínio forem consideradas pela função heurística, mais ignorante ela será e o erro entre a distancia real (h*) e a distância estimada (h) também será maior. A eficiência do algoritmo de busca informada depende da qualidade da função heurística.

3.2.1 O algoritmo de busca heurística A*

O A* é o algoritmo de busca informada mais conhecido [5]. Essa busca avalia nós combinando g(n), o custo para alcançar cada nó, e h(n), o custo estimado para chegar até um estado meta. Assim, o custo estimado total para ir do nó n até o estado objetivo é: f(n) = g(n) + h(n).

Dada uma função heurística admissível o algoritmo A* garante encontrar uma solução ótima, uma vez que ele sempre visita o nó com o menor valor de f(n) [1].

```
A_STAR( estadoInicial ) {
  estadoInicial.g = 0;
  inserir(listaOrdenadaAberta, estadoInicial );
  sucesso = false;
  faça {
       estadoAtual = pegarMelhor(listaOrdenadaAberta);
       se( solucao( estadoAtual))
         sucesso = true;
       senão {
         para cada (filho(estadoAtual)) faça {
              se ( estaEm(listaOrdenadaAberta(filho(estadoAtual)))) {
                 estadoAntigo = pegar(listaOrdenadaAberta(filho(estadoAtual)))
                 estadoAntigo.g = min(estadoAntigo.g, filho(estadoAtual).g)
              }
              senão se ( estaEm( listaFechada(filho(estadoAtual))))
                     propagarG(pegar(listaFechada(filho(estadoATual))));
              senão
                     inserirOrdenado(listaOrdenadaAberta, filho(estadoAtual));
         }
         inserir(listaFechada, estadoAtual);
       }
  } até( sucesso ou vazio (listaAbertaOrdenada));
  se (sucesso) devolva (estadoAtual);
  senão devolva (nulo);
}
```

Algoritmo 3: pseudo-código da busca informada A. Adaptada de [1*

O Algoritmo 3 implementa o A*. Ele inicia inserindo a raiz da árvore em uma fila com prioridade usando a variável *listaOrdenadaAberta.* A ordenação da *listaOrdenadaAberta* leva em consideração a heurística que estima quantos passos ainda são necessários para chegar no estado objetivo. A ordenação também leva em consideração a distância que já percorreu para chegar nesse estado n, isto é g(n). A cada passo, o estado com maior prioridade, *estadoAtual*, é retirado na fila de prioridade. Em seguida, é verificado se o *estadoAtual* é o estado objetivo. Caso isso seja verdade, esse estado é devolvido (bem como, o caminho desse nó ao nó raiz).

Caso contrário, o *estadoAtual* é armazenado na *listaFechada* e é expandido. Para tratar ciclos no espaço de estados, para cada nó sucessor gerado é feita a verificação se eles já estão na *listaOrdenadaAberta. S*e estiverem, *g(n)* é atualizado. Esse passo é repetido até uma solução ser encontrada ou até a *listaOrdenadaAberta* ficar vazia.

3.2.3 IDA*

O algoritmo IDA* combina o algoritmo A* com a idéia de busca em profundidade. Cada iteração do IDA* tenta encontrar uma solução com um caminho de tamanho igual a *tamanhoDeCaminhoLimite*. Na primeira iteração, o *tamanhoDeCaminhoLimite* tem o valor da avaliação heurística calculada para o estado inicial. Após a árvore ter sido percorrida exaustivamente em profundidade limitada e nenhuma solução ter sido encontrada, isso prova que nenhuma solução de tamanho igual a *tamanhoDeCaminhoLimite* existe e então *tamanhoDeCaminhoLimite* é aumentado e uma nova iteração de busca em profundidade limitada começa. [1]

```
IDA_STAR (estadoInicial) {
       tamanhoDeCaminhoLimite = H(estadoInicial) - 1;
       sucesso = false;
       faça {
              tamanhoDeCaminhoLimite++;
              estadoInicial.g = 0;
              inserirOrdenado(fila, estadoInicial);
              faça {
                estadoAtual = pegarMelhor(fila);
                se (solucao(estadoAtual));
                           sucesso = true
                senão se (tamanhoDeCaminhoLimite >= (estadoAtual.g + H(estadoAtual)))
                           para cada (filho( estadoATual)) faça
                                  inserirOrdenado( fila, filho(estadoAtual));
              } até (sucesso ou vazio (pilha))
       } até (sucesso ou fim de memória)
       se (sucesso) devolva (estadoAtual);
       senão devolva (nulo);
}
```

Algoritmo 4: pseudo-código do algoritmo IDA. Adaptado de [1].*

4. Trabalhos desenvolvidos para resolver o Sokoban

Vários trabalhos foram desenvolvidos com o intuito de resolver o Sokoban. Mark James [6], em sua dissertação de mestrado usou o Sokoban para mostrar que as técnicas desenvolvidas por ele que funcionavam bem em outros domínios, não funcionam bem com o Sokoban. Seu programa conseguiu resolver apenas o tabuleiro #1 dos 90 tabuleiros Sokoban Benchmark [9] após 2 horas de processamento.

Andrew Mayers desenvolveu um programa que consegue resolver 9 problemas dos 90 Benchmark. O algoritmo de busca usado por ele é o A*. Ele usa uma tabela de hash para armazenar os estados que já foram visitados e sua heurística leva em conta tanto o número de vezes que a caixa deve ser empurrada quanto o número de movimentos do agente. Deadlocks de uma área de 3 x 3 são detectados automaticamente porém, não são armazenados em uma tabela.

Stefan Edelkamp [7], durante seu doutorado, desenvolveu um programa que consegue resolver 13 problemas. O seu programa foca em resolver o Sokoban com um número ótimo de movimentos de caixas, usando um algoritmo sofisticado para avaliar deadlocks e armazenar padrões de deadlock em uma estrutura de dados bem elaborada. Dentre as soluções que usam heurísticas não admissíveis está o trabalho da Universidade de Maiji, dos estudantes A. Ueno, K. Takayama e T. Hikita [8] que desenvolveram um programa que resolve 25 dos 90 problemas. O programa é também baseado no algoritmo A* mas usa uma heurística não admissível. Desta forma, as soluções encontradas não são ótimas para movimentos de caixas nem para os movimentos do agente. O Laboratório de Sokoban (URL desativada), um projeto feito no Japão para desenvolver novos tabuleiros de Sokoban, também tem um programa que resolve o problema automaticamente capaz de resolver 55 dos 90 problemas Benchmark. Porque usa uma heurística não admissível, as soluções encontradas não são ótimas. Esse programa é baseado em parte no programa que foi desenvolvido na Universidade de Meiji.

O melhor programa desenvolvido até hoje resolve 62 dos 90 problemas e foi desenvolvido por uma pessoa que se intitula *Deepgreen* (seu verdadeiro nome é desconhecido). Tudo indica que esse programa é baseado em outros desenvolvidos pela comunidade japonesa do Sokoban. Porém, tanto o programa como artigos relacionados, não podem ser localizados.

Finalmente, as principais técnicas estudadas para a realização desse trabalho foram desenvolvidas por Andreas Junghanns [1] em seu doutorado na Universidade de Alberta, Canadá [2]. Seu programa usa técnicas clássicas de jogos de IA, melhoradas para funcionarem adequadamente no domínio do Sokoban. O trabalho do Andreas foi selecionado para ser estudado pois ele implementou um dos programas mais eficientes para resolver tabuleiros de Sokoban (resolve 54 dos 90 Problemas Benchmark). Além disso, sua tese e diversos artigos estão disponíveis na Internet. O nome do programa desenvolvido é **Rolling Stone.** Uma das principais características desse programa é considerar apenas o número de movimentos das caixas, desconsiderando o número de movimentos do agente (como no Exemplo 2).

No início do projeto, o programa Rolling Stone tinha sido desenvolvido para sempre devolver soluções ótimas. Mas com essa escolha, o Rolling Stone conseguia resolver uma quantidade pequena de problemas. Assim, Andreas preferiu resolver mais problemas com soluções sub-ótimas ao invés de encontrar soluções ótimas para uma menor quantidade de tabuleiros.

5. O Algoritmo Rolling Stone

O programa Rolling Stone usa como algoritmo de busca o **IDA*.** Ele é o mais apropriado para a resolver o Sokoban por suas características: existem poucos estados objetivo num espaço de busca muito grande e esses estados estão localizados em uma profundidade grande [1].

Uma das principais características do algoritmo Rolling Stone proposto por Andreas é que ele implementa um conjunto de técnicas distintas que juntas conseguem resolver os problemas difíceis do Sokoban. Entre elas estão:

1. Heurística com base numa tabela de associação entre caixas e metas que também registra as estimativas de distâncias mínimas para essas associações.
2. Tabelas de hash para armazenar estados repetidos
3. Ordenação de movimentos para resolver conflitos entre movimentos
4. Tabelas de deadlock
5. Macro movimentos
6. Busca de padrões de deadlocks
7. Podas relevantes
8. Heurística de limite superior

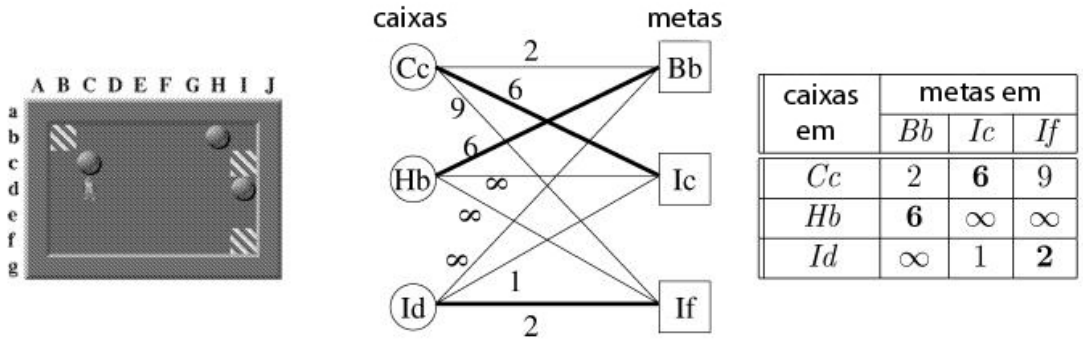
5.1. Heurística

A heurística usada no Rolling Stone é baseada em uma tabela que associa caixas às posições de armazenamento, uma vez que essa associação é fundamental para a resolução do problema.

5.1.1. Tabela de Associação

A Tabela de Associação, associa caixas às posições de armazenamento com base no caminho de menor custo. Inicialmente essa tabela é construída usando-se uma heurística de limite inferior, ou seja, uma heurística que calcula o número mínimo de movimentos que uma solução deve ter. Essa heurística inicial é chamada de *Closest*, que junto com as regras de movimentação das caixas, calcula o número mínimo de movimentos das caixas para todos os pontos de armazenamento com base na distância de Manhatan, ignorando interações entre as caixas.

Podemos dizer que esse problema é idêntico a resolver um problema para um grafo bipartido, já que para cada caixa existe uma meta associada. A heurística tenta encontrar a associação entre as caixas e as metas de custo mínimo (*minmatching*). Nesse grafo bipartido, os pesos das arestas são as distâncias entre as metas e as caixas. O peso de uma aresta pode ser infinito caso não exista um caminho entre uma caixa e uma meta.



caixas em	metas em		
	Bb	*Ic*	*If*
Cc	2	**6**	9
Hb	**6**	∞	∞
Id	∞	1	**2**

Figura 10: Exemplo de como a tabela de menor custo é construída. Adaptada de [1].

O custo para computar a associação de menor custo é alto: para um grafo com n nós e m arestas é de $O(n * m * \log_{(2+m/n)} n)$. Como o problema trata-se de um grafo bipartido completo, $m = n^2/4$, e a complexidade é igual a $O(n^3 * \log(2+n/4)n)$, isso representa uma computação de alto custo [1].

Um recurso importante da associação de menor custo é que ela já é capaz de detectar alguns deadlocks. É importante também notar que a Tabela de Associação é atualizada durante a busca. Além disso, no algoritmo Rolling Stone, o cálculo das distâncias entre as caixas e as metas é atualizado e melhorado durante a busca, como será discutido a seguir.

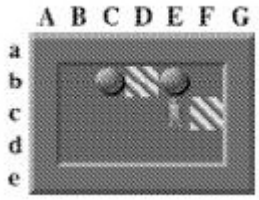


Figura 11: Exemplo de deadlock detectado pela associação de menor custo [1]

5.1.2 Melhorias na Tabela de Associação

Melhora na entrada

Para melhorar a eficiência do cálculo da Tabela de Associação, Andreas propôs a técnica de melhora de entrada.

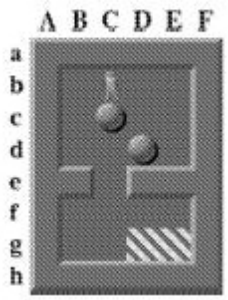


Figura 12: As duas caixas devem passar obrigatoriamente por Ce [1]

No tabuleiro da Figura 12, para as caixas alcançarem suas metas elas devem passar obrigatoriamente pelo quadrado Ce, que será chamado de entrada, X. Então, ao invés da Tabela de Associação de menor custo calcular a distância das caixas até suas respectivas metas, ela calculará a distância das caixas até X, para depois somar as distâncias de X até suas metas (valores constantes).

Posição do Agente

Quando a distância das caixas até suas metas são calculadas, um fato muito importante estava sendo ignorado, a posição do agente no mapa. A heurística descrita anteriormente considera que o agente pode mover-se de um lugar para qualquer outro no tabuleiro. Porém, a presença de paredes e a existência de caixas restringem o espaço no qual o agente pode movimentar-se. Quando uma caixa é empurrada para cima, o agente precisa encontrar espaço no tabuleiro para locomover-se para baixo da caixa e assim empurrá-la para cima. Além disso, existem situações que o agente precisa empurrar a caixa para fora de sua trajetória ótima para posicionar-se atrás da caixa e então empurrá-la em direção a uma meta.

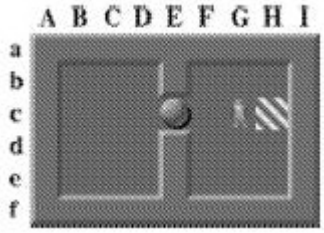


Figura 13: A distância depende da posição do agente [1]

Na Figura 13 o agente tem que empurrar a caixa 2 quadrados para trás e então empurrá-la na direção da meta. Note que a Tabela de Associação não levou em consideração esses dois empurrões para trás que a caixa precisou, a capacidade de detectar essa necessidade e conseguir melhorar a heurística de limite inferior é chamada de *conflito-deve-voltar* (*backout conflict*).

Conflitos Lineares

Com as técnicas anteriormente apresentadas, para calcular a distância de uma caixa até sua meta, sempre foi considerado ser possível empurrar essa caixa, não se considerou as caixas que estão ao seu redor. Mas será que as caixas realmente podem sempre ser empurradas? Na Figura 13, para calcular a distância da caixa localizada em Ec, até a meta a heurística *associação de menor custo* empurraria a caixa da posição Ec para Fc, porém esse movimento é inválido. Para conseguir empurrar essa caixa em direção à sua meta é necessário mover a caixa que está no quadrado Dc de sua trajetória ótima e então continuar com o plano. Esse é um exemplo de conflito linear. Quando uma situação dessas é detectada, essa técnica penaliza o cálculo da heurística em 2, pois a caixa precisa sair de sua trajetória, e depois voltar, ou seja, faz dois

movimentos a mais que o calculado pela associação de menor custo

Problema com Conflitos Lineares

Quando dois conflitos lineares são encontrados, simplesmente penalizar o cálculo da heurística em 4 pode estar errado. Considere a Figura 14:

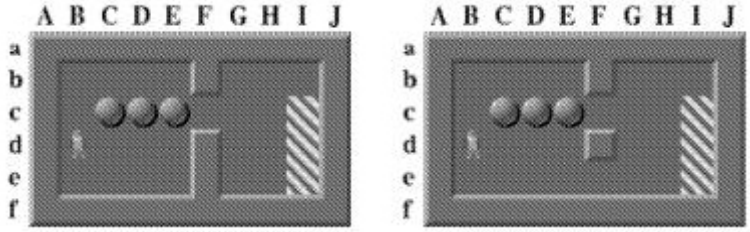


Figura 14: Penalizar o estado a direita em 4 está errado.

Na situação ilustrada a esquerda, na Figura 14, existem dois conflitos lineares, porém só a caixa do meio deve sair da sua trajetória ótima, então essa situação seria penalizada em 2. Mas caso a caixa do meio estivesse bloqueada por uma parede em Dd, por exemplo, esse movimento seria impossível e duas caixas deveriam sair de sua trajetória ótima, então a penalização de 4 seria justa. Mas agora olhe para a situação ilustrada à direita, não existe mais conflitos lineares, pois existe uma entrada extra que foi colocada embaixo no tabuleiro. Se a caixa do meio for empurrada para baixo as outras estarão desbloqueadas. Porém, somente uma caixa pode ser empurrada para baixo, pois só existe uma meta que pode ser alcançada nessa posição. Além disso, se outra caixa for empurrada para baixo, ao invés da caixa do meio, movimentos não ótimos serão necessários, então o conflito linear tem que ser quebrado empurrando a caixa do meio para baixo.

5.2 Tabelas de Hash

No Rolling Stone as tabelas de hash servem para evitar visitas em estados que já foram visitados, ou seja, ciclos na busca. A implementação dessa tabela tem chaves com tamanho de 64 bits. Essa tabela de hash é organizada em dois níveis, por motivos de eficiência. No primeiro nível estão os estados que foram buscados profundamente e no segundo nível os estados gerados mais recentemente.

As chaves das tabelas contém as posições exatas das caixas. Para casar com uma entrada a chave precisa ser idêntica. Como a posição do agente é importante, um segundo teste é feito. Deve existir um caminho válido entre a posição do agente do estado que está sendo testado para ser colocado na tabela e o estado que já está na tabela. Nesse caso, as posições do agente não precisam ser idênticas.

5.3 Ordenação de Movimentos

Ao invés de visitar estados sucessores em uma ordem arbitrária, essa técnica permite visitar os melhores estados sucessores antes dos demais. Dessa maneira é possível empurrar uma mesma pedra várias vezes seguidas até a área das metas, deixando mais espaço disponível no tabuleiro para movimentar as caixas restantes.
O esquema de ordenação dos movimentos preserva a inércia do movimento, ele faz isso da seguinte maneira:

1. Caixas que já estavam em movimento são consideradas antes
2. Então todos os movimentos que diminuem a estimativa de limite inferior, movimentos ótimos, ordenados pela distância da caixa que foi empurrada até a meta que ele está designada e com as caixas que estão mais próximas das metas antes, são consideradas
3. Então todos movimentos não ótimos são tentados e são ordenados como os movimentos ótimos.

É importante dizer que essa técnica só é usada na última iteração do IDA*, pois não faz sentido gastar processamento ordenando os movimentos dos nós internos já que eles serão percorridos de qualquer maneira,

e como o tamanho da árvore cresce exponencialmente a cada iteração, um grande ganho de desempenho é obtido se a última iteração terminar o mais cedo possível, por isso os movimentos da última iteração é que são ordenados.

5.4 Tabelas de Deadlock

No início do desenvolvimento do Rolling Stone, foram codificadas várias lógicas para detectarem deadlocks online. Porém essa técnica era capaz de detectar somente deadlocks triviais. Além disso, codificar essas lógicas de detecção de deadlock online, a cada novo tabuleiro, tornou-se inviável. Dessa maneira, outra técnica foi elaborada. Um programa que roda antes da busca IDA*, armazena todos os padrões de deadlock de tamanho 5x4 em um banco de dados. Esse banco pode ser consultado durante a busca IDA*. Toda vez que um movimento for gerado, o algoritmo consulta esse banco de dados para checar se gerou um deadlock.

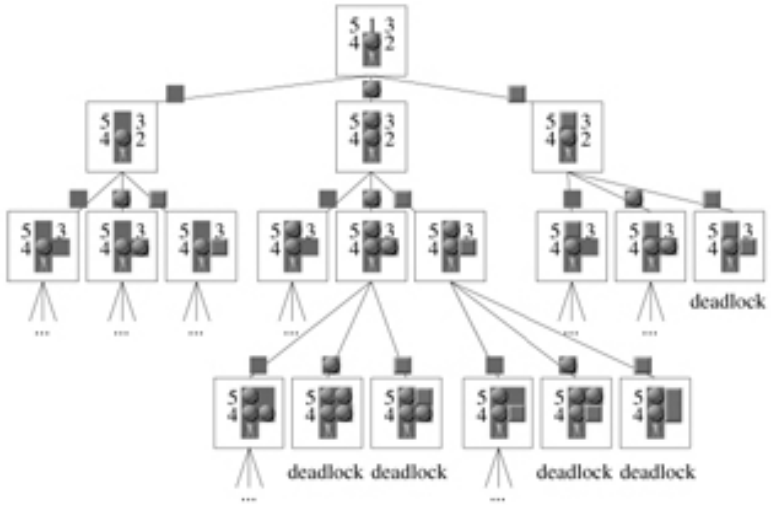


Figura 15: Ilustração parcial da tabela de deadlock [1]

Esse programa é usado para armazenar todas as combinações de paredes, caixas e espaços vazios em um tamanho fixo, no caso do Rolling Stone o tamanho fixo é 5x4. Uma busca é feita para verificar se essa combinação feita é um padrão de deadlock ou não. Essas informações são armazenadas nas tabelas de deadlock, que são implementadas como árvores de decisão. Nós internos representam sub-padrões com ponteiros para árvores sucessoras. Essa árvore sucessora representa o padrão do pai mais uma pedra, um quadrado vazio ou uma parede. Cada nível da árvore de decisão contém sub-padrões diferentes do mesmo formato. As folhas da árvore representam o status do padrão se é deadlock ou não.

Limitações

Pode parecer razoável o tamanho dos padrões de deadlock detectados pela técnica das tabelas de deadlock: 5x4, já que o tamanho do tabuleiro é no máximo 20x20. Porém, muitos deadlocks são encontrados durante a busca IDA* e não são detectados durante a construção dessa tabela. Por outro lado, é impraticável construir tabelas de deadlock com padrões maiores, pois para construir a tabela de deadlock que armazena esses padrões de tamanho 5X4, o programa levou semanas [1].

5.5 Macro Túnel

Túnel ou corredor é uma parte do tabuleiro que restringe a mobilidade do agente a apenas uma direção. A técnica de Macro Túneis trata esses corredores como se fossem apenas uma única localização, ou seja, como o jogador só pode ir para uma direção, uma vez que entra dentro de um túnel, os movimentos intermediários

dentro do túnel não são gerados, eles são substituídos pelo movimento que leva o jogador até o fim do túnel.

Os túneis podem ser classificados em dois tipo: túneis unidirecionais e bidirecionais. Túneis unidirecionais são partes do tabuleiro que são compostas por articulações das paredes, ou seja, uma vez que uma caixa é empurrada dentro dessa área, elas devem ser empurradas até o outro lado, pois não existe caminho no tabuleiro que o agente possa fazer para empurrar a caixa de volta para o lado por onde ela entrou.

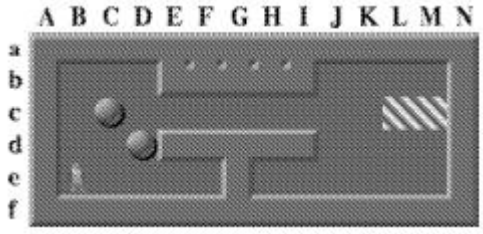


Figura 16: Túnel unidirecional [1]

Túneis bidirecionais podem ser usados como uma área de armazenamento, ao contrário dos túneis unidirecionais, pois há pelo menos dois caminhos diferentes que ligam suas extremidades, dessa maneira o agente pode deixar a caixa dentro de um túnel bidirecional, usar o outro caminho alternativo para empurrar a caixa para o lado por onde ela entrou.

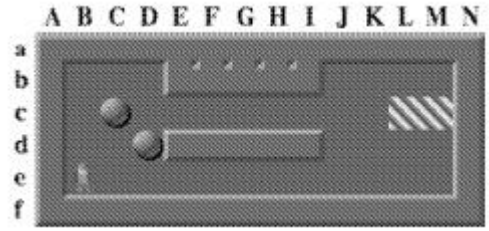


Figura 17: Túnel Bidirecional [1]

5.6 Macro Metas

A técnica de macro metas pode ser dividida em dois estágios, o primeiro é a pré-computação, antes da busca IDA* e a segunda parte que é acionada várias vezes durante a busca.

Em vários tabuleiros do Sokoban as metas são encontradas todas juntas em algumas regiões e para chegar nessas regiões geralmente há pequenas passagens, as entradas. Para resolver esse problema bastaria primeiro fazer com que as caixas cheguem em alguma entrada, depois então, empurrar as caixas para uma dessas metas dentro dessa região.

Na pré computação dessa técnica é feito o seguinte:

1. Detecção das regiões onde as metas estão agrupadas e marcação das entradas dessas regiões
2. Determinação da ordem em que as metas devem ser preenchidas para que não gere um deadlock
3. Criação de uma estrutura de dados para que possa ser acessado durante a busca IDA* com essas informações.

Durante a busca quando uma caixa chega na entrada de uma região com metas, a estrutura que guarda as informações de como empurrar uma caixa até uma determinada meta é acionada. Então, essa caixa é empurrada para a meta correta, a ordem em que as metas são preenchidas foram determinadas na pré-computação. Todos os outros movimentos alternativos possíveis com aquela caixa, que foi posicionada em

sua meta específica, são apagados da lista de movimentos (**poda de metas**).

Mas para resolver alguns problemas, é necessário posicionar as caixas temporariamente dentro de regiões de metas, ou ainda, estacionar caixas em quadrados que são metas. Além disso, regiões de metas que contém várias entradas pode ser um lugar de passagem do tabuleiro para o agente, e se as caixas forem posicionadas para sempre nessa passagem, podem impedir o agente de trafegar por ali

.

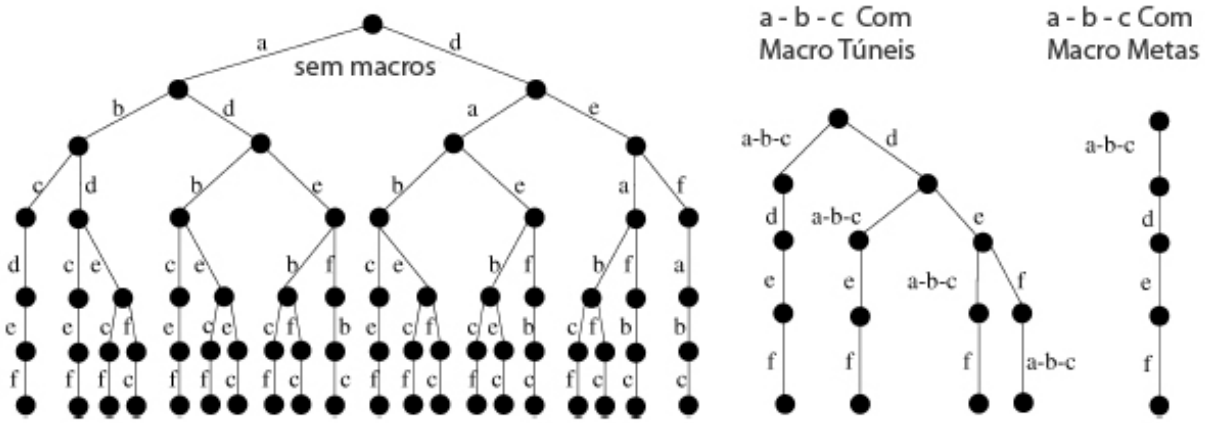


Figura 18: Ilustração do poder de poda das macro metas combinadas com os macro túneis. Adaptado de [1]

É importante dizer que com o acréscimo das macro metas, mais as podas de metas, não é possível garantir que a solução encontrada seja ótima, pois outras soluções podem ter sido ignoradas quando a poda no espaço de busca foi feita após a caixa ter sido posicionada em uma meta.

5.7 Busca de padrões

A busca de padrões é um novo acessório da busca que é capaz de detectar dealocks dinamicamente e com isso melhora a heurística de limite inferior. É um algoritmo que adquire conhecimento em tempo real, identifica as condições mínimas necessárias para formar um deadlock e usa esse conhecimento para eliminar partes da árvore de busca que provavelmente são irrelevantes.

Quando um movimento é feito no tabuleiro, eventualmente um deadlock pode ter sido criado. Um deadlock pode ser formado por todas as caixas do tabuleiro ou apenas uma. Para provar que um subconjunto de caixas, um padrão, é um deadlock, é necessário fazer algumas buscas e verificar que não existe nenhum caminho na árvore que leva a uma solução.

A busca de padrões na verdade são várias buscas IDA*. Cada busca IDA* roda com mais caixas que a busca anterior, até o número total de caixas existentes no tabuleiro ser atingido. Pode ser que a busca de padrões encontre um padrão de deadlock e use esse padrão durante a busca e atribua a heurística correta, que seria infinito, para os estados que coincidirem com os padrões de deadlock.

O algoritmo segue esses quatro passos:

1. Coloca apenas a última caixa movida no tabuleiro de testes
2. Tenta encontrar uma solução
3. Se uma solução não for encontrada, um deadlock foi detectado, saia
4. Se uma solução foi encontrada, adicione uma caixa que está posicionada em um quadrado que é necessário para a solução encontrada.
5. Vai para 2

A busca de padrões é encerrada em três casos:

1. Alcançou o limite do esforço da busca, geralmente esse esforço é medido pelo número de nós expandidos pela busca).
2. Um deadlock foi encontrado.
3. Nenhuma das caixas restantes do tabuleiro original conflitam com a solução encontrada.

Devido ao seu custo de processamento, é inviável chamar a busca a cada movimento feito no tabuleiro. Para descobrir o melhor momento de se chamar a busca algumas coisas são levadas em consideração. Por exemplo, a busca é chamada toda vez que qualquer 1 dos 3 lados da frente da caixa movida ficam inacessíveis para o agente. Caso contrário é improvável que tal movimento tenha gerado um deadlock.

Minimizando os padrões

Antes de armazenar os padrões de deadlock encontrados pela busca de padrões, ele é minimizado. Um padrão mínimo é aquele que nenhuma caixa pode ser retirada dele, caso alguma caixa seja excluída o padrão não gera mais um deadlock.

Uma subrotina é chamada para fazer tal verificação. Caixa a caixa é retirada do padrão encontrado e é checado se o deadlock foi preservado.

5.9. Podas Relevantes

Ao analisar os movimentos feitos pelo algoritmo Rolling Stone, é fácil notar que nenhuma pessoa movimentaria as peças da forma que o algoritmo faz. Muitas de suas ações não estão relacionadas com ações anteriores. A técnica de podas relevantes tenta inserir o conceito de relevância dos movimentos no Rolling Stone.

Influência

Uma boa maneira de introduzir o conceito de relevância é primeiramente introduzir o conceito de influência. Se um movimento não influencia o outro, é improvável que um seja relevante para o outro. Por exemplo, considere que uma caixa A influencia outra caixa B se ela estiver no caminho que B usa para alcançar uma meta. Se A estiver mais perto da meta do que a caixa B, então A influencia mais B do que B influencia A. No caso de um túnel, dois quadrados nas pontas desse túnel terão sempre a mesma influência sobre o outro, não importando o tamanho do túnel.

Existem duas maneiras de um movimento ser podado da busca:

1. Se nos últimos m movimentos mais de k movimentos distantes foram feitos, a poda irá desencorajar trocas arbitrárias entre áreas não relacionadas no tabuleiro.
2. Se um movimento é distante do movimento anterior, mas não é distante dos últimos m movimentos. Isto é, não será permitido trocas para áreas do tabuleiro que foram abandonadas recentemente.

As podas relevantes forçam o Rolling Stone a favorecer movimentos locais ao invés de movimentos globais, o que imita a maneira dos seres humanos resolverem problemas.

> *Exemplo 3: Resolver o problema da Figura 19, significa resolver 2 sub-problemas menores. As podas relevantes encorajam o algoritmo Rolling Stone resolver cada um desses sub-problemas de uma vez, ao invés de tentar resolver todas as caixas ao mesmo tempo.*

Figura 19: Exemplo onde resolver problemas locais antes é mais interessante [1]

5.10. Heurística de limite superior

A heurística de limite superior, ao invés de se preocupar em usar uma heurística admissível na busca, tenta aproximar-se ao máximo da heurística real, h*.

WIDA*

Estudos estatísticos são feitos para medir a diferença entre h e h* e encontrar uma constante w que pode ser usada para melhorar o calculo de f(*n*). Assim, ao invés de IDA*, a busca é chamada de WIDA* (Weighted IDA*). WIDA* usa a função f(n) = g(n) + w*h(n). Dessa maneira, a busca ganha um bônus na profundidade que pode buscar. Quanto mais ela aprofunda na árvore de busca mais ela é encorajada a aprofundar. Nós próximos da raiz têm uma heurística pior do que os nós que estão perto das folhas. Entretanto, isso pode fazer a busca considerar movimentos não ótimos [1].

Penalidade parcial total

A busca de padrões é limitada, pois toda vez que um padrão de deadlock é encontrado apenas algumas das penalidades podem ser usadas para atualizar aqueles estados para preservar a admissibilidade. Mas não penalizar esses padrões é o mesmo que ignorar conhecimento.

Com essa técnica, toda vez que um padrão de deadlock é encontrado todos as penalidades são contabilizadas naquele estado, e o valor da penalidade, ao invés de ser atribuído ao estado, e dividido entre as caixas que fazem parte do padrão que gerou o deadlock. Dessa maneira menos conhecimento é desperdiçado e mais eficiente torna-se o algoritmo de busca.

5.11. Resultados do Rolling Stone

Após a adição de todas as técnicas descritas nas seções anteriores sobre o Rolling Stone, o algoritmo de Andreas foi capaz de resolver 54 tabuleiros dos 90 problemas Benchmark. Com a implementação da busca IDA* mais a heurística de limite inferior avançada, o Rolling stone não foi capaz de resolver nenhum tabuleiro. Como a árvore de busca é muito grande, muitas podas eram necessárias para ser possível encontrar uma solução para os tabuleiros. Com a adição das tabelas de hash, o Rolling stone conseguiu resolver 5 tabuleiros, houve uma grande redução no tamanho da árvore de busca, pois eram removidas sub-árvores idênticas. Com a inclusão da ordenação de movimentos o algoritmo voltou a resolver apenas 4 tabuleiros, mais algumas técnicas deveriam ser implementadas para ser possível extrair o máximo do que essa técnica poderia oferecer. As tabelas de deadlock fizeram o Rolling Stone voltar a resolver 5 tabuleiros, algumas partes da árvore que não continham solução foram podadas com o uso dessas tabelas. A técnica de macro túnel podou a profundidade da árvore, pois tratou túneis como um único quadrado e fez o Rolling Stone resolver mais um tabuleiro. Dessa maneira, o Rolling Stone resolve 6 tabuleiros. A implementação de macro metas foi uma complementação a técnica dos macro túneis. Ela podou muito mais a profundidade da árvore de busca, pois quando uma caixa entrava na área de metas, já era direcionada para a sua meta específica. Com as macro metas o Rolling Stone passou a resolver 17 tabuleiros. Já as podas de metas foi uma complementação a macro metas. Quando uma caixa era posicionada na sua meta específica, eram podadas

todas as outras possibilidades para aquela caixa, pois ela já era considerada resolvida. Com essa técnica o fator de ramificação da árvore de busca diminuiu e o Rolling Stone passou a resolver 24 tabuleiros.

A técnica que mais trouxe bons resultados com relação aos números de tabuleiros resolvidos foi a busca de padrões. Após a adição dessa técnica o Rolling Stone passou a resolver 49 tabuleiros. A cada movimento suspeito de ter gerado um deadlock a busca é chamada. Com o banco de dados de padrões gerados por essa técnica, grandes partes irrelevantes da árvore de busca foram podadas e levou a uma redução drástica no tamanho da árvore de busca.

Com as podas relevantes, ao invés de considerar todos os movimentos possíveis na árvore, só eram considerados movimentos relevantes. Com a adição dessa técnica o Rolling Stone conseguiu resolver 51 tabuleiros. Alguns cortes extras foram feitos na árvore de busca após a inclusão da técnica de heurística de limite superior. Ao invés de usar apenas uma heurística admissível para calcular a distância dos estados até um estado meta, uma nova heurística que se aproxima mais da heurística real é usada. Com a heurística de limite superior o Rolling Stone passou a resolver 54 tabuleiros.

6. O Algoritmo SokoKids

A proposta desse trabalho é desenvolver um algoritmo que resolva todos os problemas Kids Benchmark [1]. Chamaremos esse algoritmo de **SokoKids,** uma vez que o objetivo era resolver todos os probelmas Kids Benchmark [1]. Para isso, nem todas as técnicas apresentadas na Seção 5, foram implementadas.

As técnicas implementadas no programa Sokokids foram: detecção de deadlocks, IDA*, heurística de limite inferior, posição do agente e tabela de hash. A seguir, serão dados os detalhes de cada uma dessas técnicas.

6.1 IDA*

O IDA* foi implementado como descrito na Seção 3.2.3. A lista de estados foi armazenada em uma estrutura ArrayList e sua ordenação foi feita utilizando o algoritmo HeapSort, implementado especialmente para a ordenação da lista, na propriedade f(n), g(n) + h(n), dos estados, em ordem crescente.

6.2 Detecção de deadlocks

Detecção offline de deadlocks

A técnica de detecção de deadlocks implementada no Sokokids se baseou no algoritmo Rolling Stone. Para fins de simplificação, a matriz de deadlocks é gerada a partir de análises de espaços 3x3. O resultado é uma matriz que detecta todos os deadlocks de parede do tabuleiro (que chamaremos de *pontos mortos*) e deadlocks de canto (que chamaremos simplemesnte de deadlocks) dentro dessa área de análise. Isso é feito offline como se o tabuleiro estivesse vazio, sem nenhuma caixa dentro dele, mas considerando os estados meta. Os quadrados mortos e deadlocks detectados por essa técnica são armazenados em uma estrutura que é consultada pela busca IDA*, antes de gerar os movimentos para um quadrado qualquer do tabuleiro. Se o quadrado for morto ou for um deadlock, aquele movimento não será gerado. Por esse motivo, essa técnica diminui muito a quantidade de estados gerados, como será mostrado na Seção 6.6 de resultados.

Figura 20: Os quadrados destacados em vermelhos são mortos e são detectados offline (figura adaptada de [1])

Com a implementação da detecção offline de deadlocks o algoritmo de busca foi capaz de podar todos os movimentos para os quadrados mortos. Dessa maneira o algoritmo não gasta mais tempo procurando a solução em partes irrelevantes da árvore de busca.

Detecção Online de DeadLocks

A detecção online de deadlocks é feita de forma recursiva. Ela funciona da seguinte maneira: toda vez que um movimento é gerado, essa função é chamada para verificar se um deadlock de tamanho 3x3 foi gerado. Para fazer essa verificação, após o movimento ter sido gerado é feita uma tentativa de mover a caixa para os lados. Se a caixa for movida com sucesso para qualquer lado, o movimento gerado não é um deadlock, e pode ser adicionado na fila do algoritmo IDA* como um estado válido, caso contrário, ainda existe a possibilidade de que a caixa que foi movimentada esteja bloqueada por outras caixas que podem ser movidas e liberar o caminho para ela também possa ser movida. Se o que está bloqueando a passagem da caixa que está sendo testada são outras caixas, esse método também é chamado para as que estão bloqueando a passagem da caixa testada e assim por diante, se a caixa que está bloqueando a passagem consegue ser movida e sair do caminho, o método devolve Verdadeiro, se a caixa que está bloqueando o caminho não consegue ser movida, o método devolve Falso. Caso o que esteja bloqueando a passagem da caixa testada seja uma parede o método também devolve Falso. Se o movimento gerado for considerado um deadlock ele é descartado e não é adicionado à fila do algoritmo IDA*.

6.3 Posição do Agente

Como descrito na Seção 5.1.2, a posição do agente é muito importante para geração de estados válidos. Sempre que estados sucessores forem gerados a posição do agente deve ser verificada, pois pode acontecer de ao se tentar movimentar uma caixa, esta não seja acessível para o agente e se esse movimento fosse considerado ele seria ilegal.

A implementação dessa técnica no programa desenvolvido foi feita da seguinte maneira: antes de um movimento ser gerado, é feita uma verificação se a caixa que deve ser movida está acessível para o agente. Se ela estiver acessível, o movimento é considerado válido e o estado resultante é adicionado na fila do algoritmo IDA*, caso contrário, o movimento é descartado.

Essa verificação é feita de maneira recursiva, através de uma busca em largura. Por exemplo, se a caixa deve ser empurrada para baixo, é verificado se a parte de cima da caixa é acessível. Uma busca em largura é feita para verificar se o agente está em algum dos quadrados adjacentes ao lado de cima do quadrado, a cada iteração. A base da recursão é quando se encontra um caminho que alcança o agente, uma parede ou uma caixa que não pode ser movida. Se o agente for encontrado, o método devolve Verdadeiro. Se uma parede ou caixa for encontrada, o método devolve Falso.

6.4 Heurística de Limite inferior

A heurística de limite inferior usada no Sokokids é baseada na associação de menor custo proposta pelo algoritmo Rolling Stone. Porém, no Sokokids, a Tabela de Associação (entre caixas e metas) não é construída explicitamente. Isso foi feito para evitar as diversas atualizações da tabela necessárias durante a busca (isto é, as melhorias descritas na Seção 5.1.2). Assim, no Sokokids, a associação de menor custo é calculada toda vez que um estado é gerado.

A função *Count*, que conta quantas caixas ainda não estão resolvidas, é usada para dar preferência a estados que tem mais caixas já resolvidas que outros estados.

6.5 Tabela de Hash

A implementação da tabela de Hash foi baseada na técnica apresentada por Andreas. Como foi dito anteriormente, durante a execução do IDA*, alguns estados podem ser gerados mais de uma vez, pois existem diversos caminhos circulares no espaço de busca. Para evitar esse tipo de problema, antes de inserir um estado válido na lista do algoritmo do IDA*, é feita uma consulta na tabela hash de estados para verificar se este já havia sido gerado ou visitado.

A implementação da tabela de hash utilizou a biblioteca Java Hashtable e usou como índices diversas características do estado:

- Posição das caixas
- Posição das metas
- Posição do agente.

Esse último ítem é muito importante, pois queremos considerar também como estados equivalentes aqueles que, geram a mesma solução, independente da posição do agente. Dois estados são equivalentes quando o agente pode mover-se de uma posição até outra sem mover nenhuma caixa. Em termos computacionais, ambas posições permitem chegar ao mesmo resultado. Para determinar se uma posição é equivalente a outra, basta realizar uma busca em largura a partir de uma das posições e tentar encontrar a outra, sem que haja barreiras (caixas ou paredes) impedindo o encontro. A implementação é muito parecida com a que vimos no item 6.3 e segue a mesma lógica.

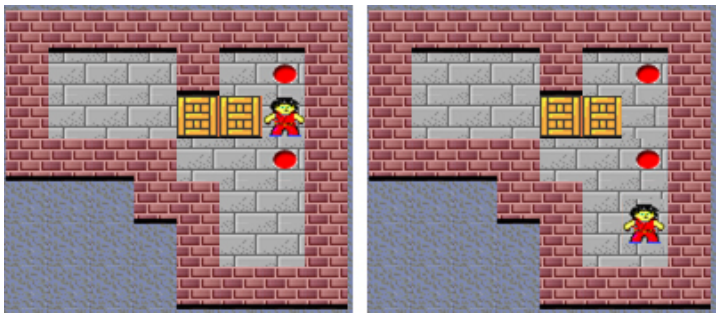


Figura 21: Dois estados considerados equivalente na tabela hash, independente das diferentes posições do agente.

6.6 Problemas de Teste (Problemas Kids Benchmark) e Resultados

Com a implementação das técnicas selecionadas e descritas acima o programa Sokokids foi capaz de resolver 100% dos Problemas Kids Benchmark, conforme o objetivo desse trabalho.

No Apêndice A é mostrada uma tabela dos 61 Problemas Kids Benchmark, incluindo a descrição de todas as soluções encontradas pelo Sokokids, bem como o número de estados gerados para encontrar a solução.

Apesar da heurística implementada não ser admissível, como no programa Rolling Stone, a maioria das soluções encontradas pelo Sokokids (Apêndice A) foram as de menor caminho (solução ótima). Isso foi verificado por inspeção, uma vez que os problemas são pequenos e não temos acesso a nenhum outro programa que garanta encontrar a solução ótima que pudesse ser comparado diretamente com o Sokokids.

Análise das diferentes versões do Sokokids

Como era esperado, com a adição de cada uma das técnicas implementadas foi possível observar uma grande melhoria de desempenho do algoritmo de busca IDA*, permitindo assim que todos os Problemas Kids Benchmark fossem resolvidos de forma mais rápida e com menos consumo de recursos.
Como podemos ver na tabela a seguir, vários tabuleiros não foram resolvidos sem a implementação das tabelas de hash. O uso das tabelas de hash foi a técnica que mais diminuiu o consumo de recursos do algoritmo e fizeram com que o Sokokids resolvesse 19 novos problemas que não eram resolvidos sem a sua implementação.

A implementação de detecção offline de deadlocks também teve um importante papel na melhora de desempenho do algoritmo, mas não contribuiu com a resolução de mais tabuleiros. O algoritmo com as tabelas de hash gerou em média 36,6% menos nós, contra 45,78% da implementação de detecção offline de deadlock.

Nome	Qtd. Estados Gerados sem Tabela de Hash, com detecção offline de deadlocks	Qtd. Estados Gerados sem detecção offline de deadlocks, com Tabela de Hash	Qtd. Estados Gerados com todas técnicas
kid 1	442	271	97
kid 2	10	22	10
kid 3	71	47	41
kid 4	19	35	17
kid 5	77	69	37
kid 6	12	21	11
kid 7	34	32	26
kid 8	159	304	80
kid 9	43	37	21
kid 10	32	33	31
kid 11	44	60	26
kid 12	35	35	23
kid 13	10	65	10
kid 14	17	16	13
kid 15	33	30	25
kid 16	não resolveu	54	42
kid 17	22	17	14
kid 18	288	130	99
kid 19	89	100	38
kid 20	não resolveu	220	56
kid 21	não resolveu	74	41
kid 22	não resolveu	169	67
kid 23	32	34	25
kid 24	43	39	31
kid 25	10	23	10
kid 26	não resolveu	436	184
kid27	11	14	11
kid 28	não resolveu	436	184
kid 29	355	372	108
kid 30	51	72	35
kid 31	não resolveu	86	50
kid 32	14	18	14
kid 33	56	49	37
Kid 34	não resolveu	459	304
kid 35	683	190	111
kid 36	45	56	31
kid 37	não resolveu	273	77

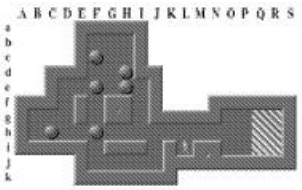
kid 38	não resolveu	486	280
kid 39	não resolveu	224	110
kid 40	não resolveu	417	281
kid 41	não resolveu	293	125
kid 42	48	58	35
kid 43	393	263	106
kid 44	44	82	33
kid 45	206	234	110
kid 46	608	197	123
kid 47	46	115	37
kid 48	46	115	37
kid 49	23	28	16
kid 50	155	539	320
kid 51	não resolveu	1349	317
kid 52	não resolveu	32672	14778
kid 53	não resolveu	35	26
kid 54	não resolveu	655	227
kid 55	Não resolveu	3641	832
kid 56	167	76	54
kid 57	não resolveu	não resolveu	94388
kid58	não resolveu	não resolveu	76644
kid 59	102	74	74
kid 60	não resolveu	400	323
kid 61	não resolveu	não resolveu	253290

Limitações do Algoritmo Sokokids

Tabuleiros que contém áreas estreitas para manobras e consequentemente tem grandes chances de gerar um maior número de deadlocks dificultando assim a detecção offline e online de deadlocks foram mais difíceis de serem resolvidos pelo Sokokids (problemas 57, 58 e 61). Mesmo assim, o Sokokis foi capaz de encontrar uma solução. Uma possível solução para essa deficiência do algoritmo seria a implementação das tabelas de deadlock.

Outro aspecto que pode dar muito trabalho para o algoritmo são tabuleiros que dependem da escolha dos movimentos mais relevantes. Para resolver esse tipo de tabuleiro, o agente deve resolver pequenos problemas, sem pensar em resolver todas as caixas ao mesmo tempo, levando em consideração a necessidade de recuar algumas caixas para longe de suas metas para depois então resolvê-las. A técnica que poderia ser implementada para melhorar o algoritmo com a percepção de relevância dos movimentos e influência entre as caixas é a de podas relevantes (Seção 5.9).

Apesar de não ter sido esse o objetivo do desenvolvimento do Sokokids, ele foi capaz de resolver um dos 90 Problemas Benchmark [1], cuja solução é mostrada a seguir.

tabuleiro	Qtd. Estados Gerados	Solução
	117364	Caixa 0: Fc Caixa 1: Hd Caixa 2: Fe Caixa 3: He Caixa 4: Ch Caixa 5: Fh Caixa 3: Ge , Caixa 1: Hc , Caixa 2: Ff , Caixa 5: Gh , Caixa 5: Hh , Caixa 5: Ih , Caixa 5: Jh , Caixa 5: Kh , Caixa 5: Lh , Caixa 5: Mh , Caixa 5: Nh , Caixa 5: Oh , Caixa 5: Ph , Caixa 5: Qh , Caixa 2: Fg , Caixa 2: Fh , Caixa 2: Gh , Caixa 2: Hh , Caixa 5: Qg , Caixa 2: Ih , Caixa 2: Jh , Caixa 2: Kh , Caixa 2: Lh , Caixa 2: Mh , Caixa 2: Nh , Caixa 2: Oh , Caixa 2: Ph , Caixa 2: Qh , Caixa 4: Dh , Caixa 4: Eh , Caixa 4: Fh , Caixa 4: Gh , Caixa 4: Hh , Caixa 4: Ih , Caixa 4: Jh , Caixa 4: Kh , Caixa 4: Lh , Caixa 4: Mh , Caixa 4: Nh , Caixa 4: Oh , Caixa 4: Ph , Caixa 4: Pi , Caixa 4: Qi , Caixa 3: Fe , Caixa 3: Ff , Caixa 3: Fg , Caixa 3: Fh , Caixa 3: Gh , Caixa 3: Hh , Caixa 2: Rh , Caixa 3: Ih , Caixa 3: Jh , Caixa 3: Kh , Caixa 3: Lh , Caixa 3: Mh , Caixa 3: Nh , Caixa 3: Oh , Caixa 3: Ph , Caixa 3: Qh , Caixa 0: Fd , Caixa 0: Fe , Caixa 0: Ff , Caixa 0: Fg , Caixa 0: Fh , Caixa 0: Gh , Caixa 0: Hh , Caixa 0: Ih , Caixa 4: Ri , Caixa 0: Jh , Caixa 0: Kh , Caixa 0: Lh , Caixa 0: Mh , Caixa 0: Nh , Caixa 0: Oh , Caixa 0: Ph , Caixa 0: Pi , Caixa 0: Qi , Caixa 1: Hd , Caixa 1: He , Caixa 1: Ge , Caixa 1: Fe , Caixa 1: Ff , Caixa 1: Fg , Caixa 1: Fh , Caixa 1: Gh , Caixa 1: Hh , Caixa 1: Ih , Caixa 1: Jh , Caixa 1: Kh , Caixa 1: Lh , Caixa 1: Mh , Caixa 1: Nh , Caixa 1: Oh , Caixa 1: Ph , Caixa 5: Rg , Caixa 1: Pg , Caixa 1: Qg

7. Conclusões e trabalhos futuros

Algoritmos de busca geral não são suficientes para resolver problemas como o Sokoban, que tem o espaço de busca muito grande. Para obter um resultado significativo, além de escolher o melhor algoritmo para esse domínio, é necessário implementar várias técnicas para melhorar o desempenho da busca e outras que diminuem o espaço de busca.

As técnicas estudadas com o objetivo de diminuir a árvore de busca ou melhorar o desempenho da busca são complexas e algumas delas influenciam no resultados de outras. Por exemplo, o algoritmo de heurística de limite superior e macro metas tiram do programa Rolling Stone a característica de devolver uma solução ótima que tinha sido prometida pela busca IDA* com uma heurística admissível. No entanto, somente assim esse algoritmo foi capaz de resolver um grande número de problemas de Sokoban.

Apesar do estudo feito se basear no domínio do Sokoban, as técnicas apresentadas são independentes do domínio. Por exemplo, a busca de padrões, tabelas de deadlocks, macro movimentos e tabelas de hash para estados repetidos, entre outras, podem ser usadas em outros domínios [1].
Não foram feitas todas as implementações que eu gostaria e ainda há muito trabalho a ser feito. As próximas

implementações sugeridas são tabelas de deadlocks, macro túneis e macro metas, uma vez que essas técnicas não são as mais difíceis implementadas no Rolling Stone mas podem apresentar um resultado muito positivo no desempenho do algoritmo.

7. Desafios e frustrações encontradas

O trabalho de estudar e implementar uma solução para o problema do Sokoban foi muito desafiador. Além de aplicar técnicas e conceitos já estudados durante o curso, foi necessário estudar diversas técnicas e conceitos específicos para o domínio do Sokoban. Muitas delas estavam descritas na tese de doutorado de Andreas e em seus artigos publicados. Ainda assim, essa técnicas não eram capazes de resolver todos os problemas. Apesar da minha principal frustração é a de não atingir um nível de implementação capaz de resolver problemas mais complexos, percebi que esse problema pode ser tão desafiador quanto resolver um jogo complexo como o xadrez. O desafio principal, que era estudar as principais técnicas e implementar as principais de modo a resolver os Problemas Kids Benchmark foi atingido.

8. Lista das disciplinas cursadas mais relevantes para o trabalho

- Estruturas de dados
- Programação orientada a objetos
- Algoritmos em grafos

Algumas disciplinas que poderiam ter contribuído, mas não foram cursadas:

- Inteligência Artificial

As disciplinas que serviram como base na implementação foram: Estruturas de Dados, Algoritmos em Grafos e Programação Orientada a Objetos. O domínio do problema do Sokoban exigiu muitas estruturas de dados diferentes para armazenar estados e construir uma árvore de busca. O entendimento dessas estruturas e de algoritmos recursivos foi essencial para a execução do trabalho. Os algoritmos aplicados sobre essas estruturas baseiam-se muito nos algoritmos vistos na matéria de Algoritmos em Grafos, o que, portanto, ajudou muito também. A linguagem utilizada para a implementação foi Java e, por esse motivo, além de propósitos de modelagem, foram muito utilizados os conceitos trabalhados na disciplina de Programação Orientada a Objetos.

9. Próximos passos

Como dito anteriormente, (na Seção 7), os próximos passos consistem em implementar as técnicas que ainda não foram implementadas como: tabelas de deadlocks, macro túneis e macro metas, de modo que o trabalho seja capaz de resolver problemas mais complexos, com o objetivo de resolver uma quantidade maior dos 90 Problemas Benchmark. Para isso, talvez seja necessário implementar novas técnicas e, portanto, também seria objetivo do estudo analisar técnicas alternativas para resolução do Sokoban. Outro caminho a ser seguido é melhorar as técnicas já apresentadas, de modo a melhorar o tempo de resolução de problemas.

8. Links relevantes

Para acessar o trabalho completo de Andreas Junghanns acesse sua página.

http://www.cs.ualberta.ca/~games/Sokoban/

Veja um programa que resolve tabuleiros e mostra a animação da resolução.

http://www.geocities.com/erimsever/sokoban5.htm

Sokoban Automatic solver, feito por Ken'ichiro Takahashi.

http://www.ic-net.or.jp/home/takaken/e/soko/index.html

Sokoban Puzzle Solver , feito por Faris Serdar Taşel

http://codecola.net/sps/index.php

Wiki do Sokoban

http://www.sokobano.de/wiki/index.php?title=Main_Page

Bibliografia

[1] Andreas Junghanns. Pushing the Limits: New Developments in Single-Agent Search. *Ph.D. Thesis*, University of Alberta, Department of Computing Science, 1999.

[2] Sokoban Homepage. http://www.cs.ualberta.ca/~games/Sokoban/

[3] Drew McDermott. Using Regression-Match Graphs to Control Search in Planning. *Artificial Intelligence*, 109 (1-2), pp. 111-159. (skip to page 150 to read the Sokoban section)

[4] Wolfgang Holzinger. Sokoban - Facinf the infeasible. May 16, 1998

[5] Stuart J. Russel e Peter Norvig. Inteligência Artificial. Editora Campus. Tradução da segunda edição, 2004

[6] Mark James. Optimal tunneling: A heuristic for learning macro-operators. Master's thesis, University of Calgary, Calgary, Alberta, Canada, Maio 1999.

[7] Stefan Edelkamp. The branching factor of regular search spaces. Em anais do AAAI, 1998.

[8] A. Ueno, K. Takayama e T. Hikita. Sokoban, “Bit special issue on Game programming”, 1997.

[9] Problemas Sokoban Benchmark. http://www.cs.ualberta.ca/~games/Sokoban/status.html

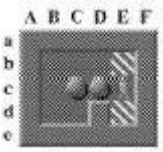
[10] Spectrum Holobyte. http://www.lysator.liu.se/adventure/Spectrum_Holobyte.html

Apêndice A

A Tabela 1 contém os Problemas Kids Benchmark, com a solução encontrada pelo SokoKids e a quantidade de nós gerados pelo programa.

Na coluna 1, chamada Nome, está presente o nome de cada tabuleiro dos 61 Problemas Kids Benchmark. Na coluna 2, chamada Tabuleiro, estão as ilustrações de cada tabuleiro. Na coluna 3, chamada Quantidade de estados gerados, mostra a quantidade de nós que foram gerados pelo Sokokids para encontrar a solução. Finalmente, na coluna Solução, é apresentado o mapeamento das caixas no tabuleiro e a solução encontrada pelo Sokokids.

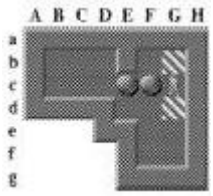
Tabela

Nome	Tabuleiro	Qtd. Estados Gerados	Solução
Kid 1		97	Caixa 0: Dc Caixa 1: Ec Caixa 2: Cd Caixa 3: Ed movimentos: Caixa 2: Ce , Caixa 3: Dd , Caixa 3: Cd , Caixa 3: Bd , Caixa 3: Be , Caixa 2: Cf , Caixa 0: Dd , Caixa 0: Cd , Caixa 0: Ce , Caixa 1: Dc , Caixa 3: Bf , Caixa 1: Cc , Caixa 1: Bc , Caixa 1: Bd , Caixa 1: Be
Kid 2		10	Caixa 0: Cd Caixa 1: Dd Caixa 2: Ed movimentos: Caixa 1: Dc , Caixa 1: Db , Caixa 2: Dd , Caixa 2: De , Caixa 2: Df , Caixa 0: Dd
Kid 3		41	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fb , Caixa 0: Dc , Caixa 1: Gb , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Fd , Caixa 0: Gd
Kid 4		17	Caixa 0: Dc Caixa 1: Ec movimentos: Caixa 1: Eb , Caixa 0: Cc , Caixa 1: Fb , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc
Kid 5		37	Caixa 0: Dc Caixa 1: Ec movimentos: Caixa 1: Eb , Caixa 0: Cc , Caixa 1: Fb , Caixa 1: Gb , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Fd , Caixa 0: Gd
Kid 6		11	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: Cb , Caixa 1: Ec , Caixa 1: Ed , Caixa 0: Db , Caixa 0: Eb

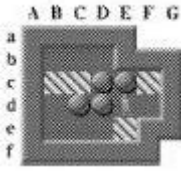
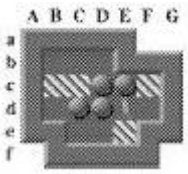
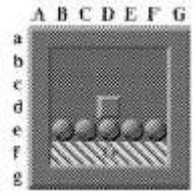
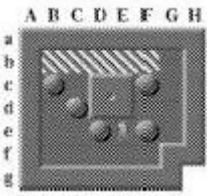
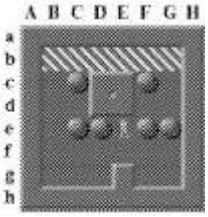
Nome	Tabuleiro	Qtd. Estados Gerados	Solução
kid 7		26	Caixa 0: Cc Caixa 1: Dc Caixa 2: Cd movimentos: Caixa 2: Dd , Caixa 2: Ed , Caixa 1: Db , Caixa 1: Eb , Caixa 0: Dc , Caixa 0: Ec
kid 8		80	Caixa 0: Cd Caixa 1: Dd movimentos: Caixa 1: Dc , Caixa 1: Cc , Caixa 1: Dc , Caixa 1: Ec , Caixa 1: Fc , Caixa 1: Gc , Caixa 0: Dd , Caixa 0: Ed , Caixa 0: Ee , Caixa 0: Fe , Caixa 0: Ge
kid 9		21	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: Cd , Caixa 1: Ec , Caixa 0: Cc , Caixa 0: Cb , Caixa 0: Db , Caixa 0: Eb
kid 10		31	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: De , Caixa 1: Fd , Caixa 1: Fc , Caixa 0: Df , Caixa 0: Cf
kid 11		26	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: Cd , Caixa 1: Ec , Caixa 1: Ed , Caixa 0: Cc , Caixa 0: Cb , Caixa 0: Db , Caixa 0: Eb
kid 12		23	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: Cd , Caixa 0: Ce , Caixa 1: Cc , Caixa 1: Cb , Caixa 1: Db , Caixa 1: Eb
kid 13		10	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 1: Db , Caixa 0: Dc , Caixa 1: Eb , Caixa 0: Ec
kid 14		13	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: Cd , Caixa 0: Ce , Caixa 1: Cc , Caixa 1: Cb , Caixa 1: Db , Caixa 1: Eb
kid 15		25	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: Cd , Caixa 1: Ec , Caixa 1: Eb , Caixa 0: Ce , Caixa 0: De , Caixa 0: Ee

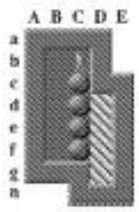
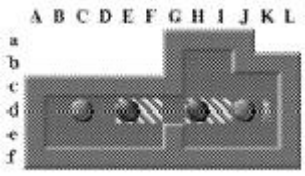
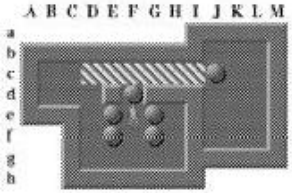
Nome	Tabuleiro	Qtd. Estados Gerados	Solução
kid 16		42	Caixa 0: Db Caixa 1: Ec Caixa 2: Ed movimentos: Caixa 1: Fc , Caixa 1: Fd , Caixa 0: Cb , Caixa 2: Ec , Caixa 2: Dc , Caixa 0: Db , Caixa 0: Eb , Caixa 0: Fb , Caixa 2: Cc , Caixa 2: Cd , Caixa 2: Ce , Caixa 2: De
kid 17		14	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: Cd , Caixa 0: Ce , Caixa 1: Cc , Caixa 1: Cb , Caixa 1: Db , Caixa 1: Eb
kid 18		99	Caixa 0: Cc Caixa 1: Dc Caixa 2: Cd movimentos: Caixa 2: Dd , Caixa 2: Ed , Caixa 2: Fd , Caixa 1: Db , Caixa 1: Eb , Caixa 1: Fb , Caixa 0: Cd , Caixa 0: Dd , Caixa 0: Ed , Caixa 2: Fc , Caixa 0: Fd
kid 19		38	Caixa 0: Dc Caixa 1: Ec Caixa 2: Fc movimentos: Caixa 1: Eb , Caixa 2: Gc , Caixa 0: Cc , Caixa 1: Fb , Caixa 1: Gb , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Fd , Caixa 0: Gd
kid 20		56	Caixa 0: Cd Caixa 1: Dd Caixa 2: Ed movimentos: Caixa 2: Ec , Caixa 2: Fc , Caixa 2: Fd , Caixa 0: Cc , Caixa 1: Ed , Caixa 2: Fe , Caixa 1: Fd , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc
kid 21		41	Caixa 0: Cc Caixa 1: Dc Caixa 2: Ec movimentos: Caixa 1: Db , Caixa 2: Fc , Caixa 2: Fd , Caixa 1: Eb , Caixa 1: Fb , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc
kid 22		67	Caixa 0: Db Caixa 1: Cd movimentos: Caixa 1: Cc , Caixa 1: Dc , Caixa 0: Cb , Caixa 1: Ec , Caixa 1: Ed , Caixa 0: Db , Caixa 0: Eb
kid 23		25	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 1: Fe , Caixa 1: Ge , Caixa 0: Dc , Caixa 0: Cc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa

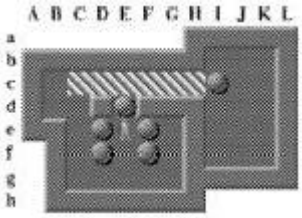
Nome	Tabuleiro	Qtd. Estados Gerados	Solução
			0: Gb
kid 24		31	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 1: Gd , Caixa 1: Gc , Caixa 0: Dc , Caixa 0: Cc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 1: Gb , Caixa 0: Gc
kid 25		10	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 1: Db , Caixa 0: Dc , Caixa 1: Eb , Caixa 0: Dd
kid 26		184	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fb , Caixa 0: Dc , Caixa 0: Cc , Caixa 1: Eb , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 0: Gd , Caixa 1: Fb , Caixa 1: Gb
kid 27		11	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 0: Cd , Caixa 1: Ec , Caixa 1: Eb
kid 28		184	Caixa 0: Ce Caixa 1: De movimentos: Caixa 1: Fb , Caixa 0: Dc , Caixa 0: Cc , Caixa 1: Eb , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 0: Gd , Caixa 1: Fb , Caixa 1: Gb
kid 29		108	Caixa 0: Ed Caixa 1: Fd movimentos: Caixa 1: Fc , Caixa 1: Ec , Caixa 0: Dd , Caixa 0: Cd , Caixa 1: Ed , Caixa 1: Fd , Caixa 1: Gd , Caixa 0: Dd , Caixa 0: Ed , Caixa 0: Fd , Caixa 1: Ge , Caixa 0: Gd
kid 30		35	Caixa 0: Cc Caixa 1: Dc Caixa 2: Bd movimentos: Caixa 2: Bc , Caixa 2: Bb , Caixa 0: Cd , Caixa 1: Ec , Caixa 1: Eb , Caixa 0: Ce , Caixa 0: De , Caixa 0: Ee
kid 31		50	Caixa 0: Cc Caixa 1: Dc Caixa 2: Bd movimentos: Caixa 2: Bc , Caixa 2: Bb , Caixa 0: Cd , Caixa 1: Ec , Caixa 1: Eb , Caixa 0: Ce , Caixa 0: De

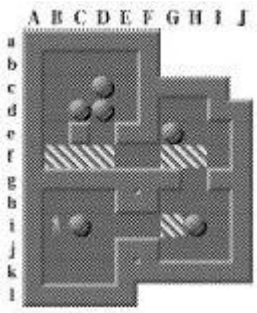
Nome	Tabuleiro	Qtd. Estados Gerados	Solução
kid 32		14	Caixa 0: Cd Caixa 1: Dd movimentos: Caixa 1: Dc , Caixa 0: Ce , Caixa 1: Ec
kid 33		37	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 1: Gd , Caixa 1: Hd , Caixa 0: Cc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 0: Gb , Caixa 0: Hb
kid 34		304	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 1: Fe , Caixa 0: Cc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 0: Gb , Caixa 1: Fd , Caixa 1: Gd
kid 35		111	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 0: Cc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 1: Ed , Caixa 1: Ec , Caixa 1: Dc , Caixa 1: Cc , Caixa 1: Dc , Caixa 1: Ec , Caixa 1: Fc , Caixa 0: Gb , Caixa 1: Gc
kid 36		31	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 1: Fc , Caixa 1: Gc , Caixa 0: Cc , Caixa 1: Gb , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc
kid 37		77	Caixa 0: Db Caixa 1: Cc Caixa 2: Dc movimentos: Caixa 1: Cd , Caixa 0: Cb , Caixa 2: Ec , Caixa 2: Ed , Caixa 0: Db , Caixa 0: Eb , Caixa 1: Cc , Caixa 1: Dc , Caixa 1: Ec
kid 38		280	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 0: Cc , Caixa 1: Fe , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 0: Gb , Caixa 1: Fd , Caixa 1: Gd

Nome	Tabuleiro	Qtd. Estados Gerados	Solução
kid 39		110	Caixa 0: Cc Caixa 1: Dc movimentos: Caixa 1: Dd , Caixa 0: Dc , Caixa 0: Ec , Caixa 1: Dc , Caixa 1: Db , Caixa 0: Fc , Caixa 0: Fd , Caixa 0: Fe , Caixa 1: Eb , Caixa 1: Fb
kid 40		281	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 0: Cc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 0: Gb , Caixa 1: Gd
kid 41		125	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 0: Cc , Caixa 1: Ed , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 0: Gb , Caixa 1: Fd , Caixa 1: Gd
kid 42		35	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 1: Fc , Caixa 1: Gc , Caixa 0: Cc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 1: Gb , Caixa 0: Gc
kid 43		106	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 0: Dc , Caixa 0: Cc , Caixa 1: Ed , Caixa 1: Ec , Caixa 1: Fc , Caixa 1: Gc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 1: Gb , Caixa 0: Gc
kid 44		33	Caixa 0: Dc Caixa 1: Ec movimentos: Caixa 1: Ed , Caixa 1: Dd , Caixa 0: Ec , Caixa 0: Fc , Caixa 1: Dc , Caixa 0: Fb , Caixa 1: Ec , Caixa 1: Fc
kid 45		110	Caixa 0: Dd Caixa 1: Ed movimentos: Caixa 1: Ee , Caixa 0: Cd , Caixa 0: Cc , Caixa 1: De , Caixa 1: Dd , Caixa 1: Ed , Caixa 1: Fd , Caixa 0: Cd , Caixa 0: Dd , Caixa 1: Fc , Caixa 0: Ed , Caixa 0: Fd
kid 46		123	Caixa 0: Ec Caixa 1: Fc movimentos: Caixa 1: Fd , Caixa 1: Ed , Caixa 0: Dc , Caixa 0: Cc , Caixa 0: Dc , Caixa 0: Ec , Caixa 0: Fc , Caixa 0: Gc , Caixa 1: Dd , Caixa 1: Dc , Caixa 1:

Nome	Tabuleiro	Qtd. Estados Gerados	Solução
			Ec , Caixa 1: Fc , Caixa 0: Gb , Caixa 1: Gc
kid 47		37	Caixa 0: Dc Caixa 1: Ec Caixa 2: Cd Caixa 3: Dd movimentos: Caixa 2: Cc , Caixa 3: Ed , Caixa 0: Dd , Caixa 1: Fc , Caixa 3: Ee , Caixa 0: Cd , Caixa 2: Bc , Caixa 0: Cc
kid 48		37	Caixa 0: Dc Caixa 1: Ec Caixa 2: Cd Caixa 3: Dd movimentos: Caixa 2: Cc , Caixa 3: Ed , Caixa 0: Dd , Caixa 1: Fc , Caixa 3: Ee , Caixa 0: Cd , Caixa 2: Bc , Caixa 0: Cc
kid 49		16	Caixa 0: Dc Caixa 1: Cd Caixa 2: Dd Caixa 3: Ce movimentos: Caixa 3: De , Caixa 1: Cc , Caixa 3: Ce , Caixa 2: Cd , Caixa 0: Dd
kid 50		320	Caixa 0: Be Caixa 1: Ce Caixa 2: De Caixa 3: Ee Caixa 4: Fe movimentos: Caixa 1: Cd , Caixa 1: Cc , Caixa 3: Ef , Caixa 4: Ff , Caixa 2: Ce , Caixa 0: Bd , Caixa 0: Bc , Caixa 2: Cf , Caixa 1: Cd , Caixa 1: Ce , Caixa 2: Df , Caixa 1: Cf , Caixa 0: Bd , Caixa 0: Be , Caixa 0: Bf
kid 51		317	Caixa 0: Bc Caixa 1: Fc Caixa 2: Cd Caixa 3: De Caixa 4: Fe movimentos: Caixa 0: Bb , Caixa 3: Ce , Caixa 1: Fb , Caixa 4: Ee , Caixa 4: Fe , Caixa 3: De , Caixa 2: Cc , Caixa 2: Cb , Caixa 4: Fd , Caixa 1: Eb , Caixa 1: Db , Caixa 4: Fc , Caixa 4: Fb , Caixa 4: Eb , Caixa 3: Ee , Caixa 3: Fe , Caixa 3: Fd , Caixa 3: Fc , Caixa 3: Fb
kid 52		14778	Caixa 0: Cc Caixa 1: Fc Caixa 2: Ce Caixa 3: De Caixa 4: Fe Caixa 5: Ge movimentos: Caixa 0: Cb , Caixa 5: Gd , Caixa 5: Gc , Caixa 5: Gb , Caixa 1: Fb , Caixa 2: Cd , Caixa 0: Db , Caixa 2: Cc , Caixa 0: Eb , Caixa 2: Cb , Caixa 3: Ce , Caixa 3: Cd ,

Nome	Tabuleiro	Qtd. Estados Gerados	Solução
			Caixa 2: Db , Caixa 3: Cc , Caixa 3: Cb , Caixa 4: Ee , Caixa 4: De , Caixa 4: Ce , Caixa 4: Be , Caixa 4: Bd , Caixa 4: Bc , Caixa 4: Bb
kid 53		26	Caixa 0: Cc Caixa 1: Cd Caixa 2: Ce Caixa 3: Cf movimentos: Caixa 1: Dd , Caixa 3: Df , Caixa 2: Cf , Caixa 3: Dg , Caixa 2: Df , Caixa 0: Cd , Caixa 0: Ce , Caixa 0: De
kid 54		227	Caixa 0: Cc Caixa 1: Bd Caixa 2: Dd Caixa 3: Ed movimentos: Caixa 2: De , Caixa 3: Dd , Caixa 2: Ce , Caixa 1: Bc , Caixa 1: Bb , Caixa 3: Dc , Caixa 3: Db , Caixa 0: Dc , Caixa 3: Eb , Caixa 3: Fb , Caixa 0: Db , Caixa 0: Eb , Caixa 2: De , Caixa 2: Dd , Caixa 2: Dc , Caixa 2: Db
kid 55		832	Caixa 0: Cd Caixa 1: Ed Caixa 2: Hd Caixa 3: Jd movimentos: Caixa 2: Hc , Caixa 1: Fd , Caixa 1: Gd , Caixa 1: Hd , Caixa 0: Dd , Caixa 0: Ed , Caixa 1: Id , Caixa 2: Hd , Caixa 1: Ic , Caixa 2: Gd , Caixa 1: Id , Caixa 2: Fd , Caixa 3: Jc , Caixa 3: Ic , Caixa 3: Hc , Caixa 3: Hd
kid 56		54	Caixa 0: Cc Caixa 1: Dc Caixa 2: Ic movimentos: Caixa 1: Dd , Caixa 0: Cb , Caixa 0: Bb , Caixa 1: Ed , Caixa 1: Fd , Caixa 1: Gd , Caixa 1: Hd , Caixa 1: Id , Caixa 1: Hd , Caixa 1: Gd , Caixa 1: Fd , Caixa 1: Ed , Caixa 1: Dd , Caixa 2: Id , Caixa 2: Hd
kid 57		94388	Caixa 0: Jc Caixa 1: Fd Caixa 2: Ee Caixa 3: Ge Caixa 4: Ef Caixa 5: Gf movimentos: Caixa 0: Kc , Caixa 0: Kd , Caixa 1: Fc , Caixa 1: Gc , Caixa 2: De , Caixa 2: D d , Caixa 2: Dc , Caixa 1: Hc , Caixa 1: Ic , Caixa 1: Jc , Caixa 1: Kc , Caixa 2: Ec , Caixa 1: Jc , Caixa 0: Ke , Caixa 1: Jd , Caixa 2: Fc , Caixa 2: Gc , Ca ixa 4: Ee , Caixa 4: De , Caixa 4: Dd , Caixa 4: Dc , Caixa 2: Hc , Caixa 2: Ic

Nome	Tabuleiro	Qtd. Estados Gerados	Solução
			, Caixa 2: Jc , Caixa 2: Kc , Caixa 4: Ec , Caixa 2: Kd , Caixa 4: Fc , Caixa 3: Fe , Caixa 4: Gc , Caixa 3: Ee , Caixa 3: De , Caixa 3: Dd , Caixa 3: Dc , Caixa 3: Ec , Caixa 5: Ff , Caixa 5: Ef , Caixa 5: Df , Caixa 5: De , Caixa 5: Dd , Caixa 5: Dc , Caixa 4: Hc , Caixa 4: Ic , Caixa 4: Jc , Caixa 4: Kc , Caixa 4: Jc , Caixa 4: Ic , Caixa 4: Hc , Caixa 4: Gc , Caixa 4: Fc , Caixa 1: Jc , Caixa 1: Ic , Caixa 1: Hc , Caixa 1: Gc , Caixa 2: Jd , Caixa 2: Jc , Caixa 2: Ic , Caixa 2: Hc , Caixa 0: Kd , Caixa 0: Kc , Caixa 0: Jc , Caixa 0: Ic
kid 58		76644	Caixa 0: Ic Caixa 1: Ed Caixa 2: De Caixa 3: Fe Caixa 4: Df Caixa 5: Ff movimentos: Caixa 0: Jc , Caixa 0: Jd , Caixa 1: Ec , Caixa 1: Fc , Caixa 2: Ce , Caixa 2: Cd , Caixa 2: Cc , Caixa 1: Gc , Caixa 1: Hc , Caixa 2: Dc , Caixa 1: Ic , Caixa 1: Jc , Caixa 1: Ic , Caixa 0: Je , Caixa 1: Id , Caixa 2: Ec , Caixa 3: Ee , Caixa 2: Fc , Caixa 3: De , Caixa 3: Ce , Caixa 3: Cd , Caixa 3: Cc , Caixa 2: Gc , Caixa 2: Hc , Caixa 4: Cf , Caixa 3: Dc , Caixa 2: Ic , Caixa 2: Jc , Caixa 2: Jd , Caixa 3: Ec , Caixa 3: Fc , Caixa 4: Ce , Caixa 4: Cd , Caixa 4: Cc , Caixa 5: Ef , Caixa 4: Dc , Caixa 5: Df , Caixa 5: Cf , Caixa 5: Ce , Caixa 5: Cd , Caixa 5: Cc , Caixa 3: Gc , Caixa 3: Hc , Caixa 3: Ic , Caixa 3: Jc , Caixa 3: Ic , Caixa 3: Hc , Caixa 3: Gc , Caixa 3: Fc , Caixa 3: Ec , Caixa 1: Ic , Caixa 1: Hc , Caixa 1: Gc , Caixa 1: Fc , Caixa 2: Id , Caixa 2: Ic , Caixa 2: Hc , Caixa 2: Gc , Caixa 0: Jd , Caixa 0: Id , Caixa 0: Ic , Caixa 0: Hc
kid 59		74	Caixa 0: Ec Caixa 1: Cd Caixa 2: Dd Caixa 3: Ee movimentos: Caixa 3: Fe , Caixa 3: Ff , Caixa 0: Fc , Caixa 0: Fb , Caixa 2: Dc , Caixa 1: Bd , Caixa 1: Be , Caixa 1: Bf , Caixa 2: Db , Caixa 2: Cb , Caixa 2: Bb
kid 60		323	Caixa 0: Ec Caixa 1: Bd Caixa 2: Dd Caixa 3: Ee movimentos: Caixa 2: Dc , Caixa 2: Db , Caixa 1: Be , Caixa 1: Bf , Caixa 0: Fc , Caixa 0: Fd , Caixa 2: Eb , Caixa 2: Fb , Caixa 0: Fe , Caixa 0: Ff , Caixa 3: De , Caixa

Nome	Tabuleiro	Qtd. Estados Gerados	Solução
			3: Dd , Caixa 3: Dc , Caixa 3: Cc , Caixa 3: Bc
kid 61		253290	Caixa 0: Dc Caixa 1: Cd Caixa 2: Dd Caixa 3: Ge Caixa 4: Ci Caixa 5: Hi movimentos: Caixa 4: Cj , Caixa 5: Gi , Caixa 0: Cc , Caixa 3: Gf , Caixa 3: Ff , Caixa 3: Ef , Caixa 3: Df , Caixa 3: Cf , Caixa 2: Dc , Caixa 3: Df , Caixa 3: Ef , Caixa 3: Ff , Caixa 3: Gf , Caixa 3: Hf , Caixa 2: Dd , Caixa 2: De , Caixa 3: Hg , Caixa 2: Df , Caixa 2: Ef , Caixa 2: Ff , Caixa 2: Gf , Caixa 2: Hf , Caixa 2: Gf , Caixa 2: Ff , Caixa 2: Ef , Caixa 2: Df , Caixa 3: Hh , Caixa 3: Hi , Caixa 3: Hh , Caixa 3: Hg , Caixa 3: Hf , Caixa 5: Fi , Caixa 5: Ei , Caixa 5: Di , Caixa 5: Ci , Caixa 3: He , Caixa 3: Hf , Caixa 3: Hg , Caixa 3: Hh , Caixa 3: Hi , Caixa 5: Di , Caixa 5: Dj , Caixa 3: Gi , Caixa 2: Cf , Caixa 2: Df , Caixa 2: Ef , Caixa 2: Ff , Caixa 2: Gf , Caixa 1: Dd , Caixa 2: Hf , Caixa 1: De , Caixa 2: Hg , Caixa 1: Df , Caixa 1: Ef , Caixa 1: Ff , Caixa 1: Gf , Caixa 0: Bc , Caixa 0: Bd , Caixa 0: Be , Caixa 0: Bf , Caixa 1: Hf , Caixa 1: Gf , Caixa 1: Ff , Caixa 1: Ef , Caixa 1: Df , Caixa 2: Hh , Caixa 2: Hi , Caixa 2: Hh , Caixa 2: Hg , Caixa 2: Hf , Caixa 2: He , Caixa 2: Hf , Caixa 1: Cf , Caixa 2: Gf , Caixa 2: Ff , Caixa 2: Ef , Caixa 2: Df , Caixa 3: Fi , Caixa 3: Ei , Caixa 3: Di , Caixa 3: Ci , Caixa 3: Di , Caixa 3: Ei , Caixa 3: Fi , Caixa 3: Gi , Caixa 3: Hi , Caixa 3: Hh , Caixa 3: Hg , Caixa 3: Hf , Caixa 5: Di , Caixa 5: Ei , Caixa 5: Fi , Caixa 5: Gi , Caixa 5: Hi , Caixa 5: Gi , Caixa 3: He , Caixa 3: Hf , Caixa 3: Gf , Caixa 5: Fi , Caixa 5: Ei , Caixa 5: Di , Caixa 5: Ci , Caixa 5: Di , Caixa 5: Ei , Caixa 5: Fi , Caixa 5: Gi , Caixa 5: Hi , Caixa 5: Hh , Caixa 5: Hg , Caixa 5: Hf , Caixa 4: Dj , Caixa 4: Di , Caixa 4: Ei , Caixa 4: Fi , Caixa 4: Gi