Introdução

Sokoban é um jogo popular em terceira pessoa, implementado em quase todos os sistemas operacionais. Foi inventado nos ano 80 no Japão por Hiroyuki Imabayashi. O jogo consiste em um tabuleiro de tamanho 20X20 ou menos, com salas, passagens e corredores. Nesse tabuleiro encontram-se caixas, áreas de armazenamento e o sokoban. O objetivo desse jogo é fazer com que o seu personagem, o sokoban (carregador de caixas em japonês), empurre todas as caixas para os lugares de armazenamento. O sokoban (que chamaremos de agente) 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 se o agente conseguir levar todas as caixas para o lugar de armazenamento. 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 que movimentos errados podem levar o agente a colocar caixas em lugares que nunca mais conseguirão ser retiradas (becos sem saída). Por exemplo: uma caixa localizada num 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. 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.

O interessante do sokoban é que 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

Algoritmos de Busca

Uma das técnicas fundamentais de IA é a busca. Programas baseados em busca podem resolver problemas que dizemos que requer inteligência. Podemos pensar que para resolver uma configuração do sokoban deveríamos fazer uma busca entre todos os estados existentes daquela configuração para achar o estado solução do nosso problema, isso parece ser simples.
Seguindo esse raciocínio, poderíamos basear nosso programa em um agente que baseia suas ações em um mapeamento direto de todos estados e ações. Contudo, um agente assim não pode operar bem em ambientes para os quais esse mapeamento seria grande demais para armazenar.
No caso do Sokoban não temos todos os estados existentes dados previamente, ou seja não temos conhecimento de todos os estados daquela configuração, pois são tantos estados existentes que não teríamos como armanezá-los na memória de um computador, além disso, mesmo se tívessemos todos os estados dados e fosse possível armazená-los de alguma maneira, seria inviável fazer uma busca exaustiva, ou de força bruta em uma estrutura tão grande como seria a de todos os estados do sokoban.
O sucesso de uma busca depende de sua habilidade em visitar a maior parte **relevante** do espaço de busca do problema. Se o espaço de busca for muito grande ou o conhecimento de uma heurística para direcionar a busca não é conhecida é improvável que o sucesso seja alcançado. É necessário *focar* a busca. Quando isso acontece geralmente os computadores ainda falham e os humanos tem uma vantagem considerável, os humanos são melhores em aprender e usar esse conhecimento para diminuir o espaço de busca e diminuir consideravelmente a complexidade do problema

que está resolvendo.
Algoritmos que não se adaptam a um problema específico mas ao contrário disso são baseados nas propriedades gerais do domínio ajudam a melhorar a eficiência da busca, por outro lado são limitados com relação a necessidade de manter o conhecimento aprendido. Os humanos, tem a habilidade de aprender enquanto estão resolvendo um problema, isso sugere desenvolver métodos dinâmicos que aprendam aos poucos coisas específicas do problema que está resolvendo e apliquem esse conhecimento aprendido, isso ajudaria a eliminar as partes menos relevante do espaço de busca e com isso diminuir a complexidade do problema. Contudo o conhecimento atual sobre esses métodos dinâmicos é limitado.
As pesquisas em jogos tem produzido muitas técnicas e métodos úteis para resolver problemas. Porém para resolver problemas reais as coisas ficam mais complexas, pois o espaço de busca dos jogos são consideravelmente menores e melhores estruturados que os espaços de busca de um problema real.

Vários algoritmos foram propostos para percorrer espaços de busca. Serão apresentados os algoritmos mais relevantes e importantes para resolver o Sokoban.

Busca desinformada

Busca desinformada, sem informação ou busca cega, significa que ela não tem nenhuma informação adicional sobre os estados, além daquelas fornecidas na definição do problema. Tudo o que elas podem fazer é gerar sucessores e distinguir um estado objetivo de um estado não objetivo.

Geração aleatória e caminhada aleatória

Caminhada aleatória faz o que o nome sugere, o algoritmo anda aleatoriamente pelos estados do espaço de busca, escolhendo qualquer um dos vizinhos. Pode parecer bobo, mas pode ser uma boa idéia se existirem bastante estados objetivos, se não for relevante a qualidade da solução encontrada e se tivermos pouco ou nenhum conhecimento sobre
o domínio do problema que estamos resolvendo. Algoritmos que seguem uma certa ordem para expandir os estados podem enfrentar problemas como espaço de memória, ciclos, transposições e caminhos infinitos, problemas que geralmente não são enfrentados por algoritmos aleatórios. Por outro lado, algoritmos aleatorios são facilmente superados por algoritmos sistemáticos quando queremos uma solução de boa qualidade, ou quando o número de estados soluções é baixo ou ainda quando temos um bom conhecimento do problema disponível.
Algumas pessoas praticam a caminhada aleatória quando tentam achar um certo produto no super-mercado que nunca compraram antes. Como não sabem onde achar esse produto geralmente cada passo da pessoa a leva a um lugar aleatório, que eventualmente será o lugar do produto que ela está procurando.
O algoritmo da caminhada aleatória ao invés de escolher um vizinho, ele escolhe randomicamente qualquer um dos nós abertos.

Pseudo código da Caminhada aleatória:

```
caminhadaAleatoria (estadoInicial) {

    armanezar( filaAberta, estadoInicial);
    sucesso = falso;
```

```
    faça {
        estadoAtual = selecioneRandomicamente(filaAberta);
        se (solucao(estadoAtual))
            sucesso = true
        se não
        para cada (filho(estadoAtual)) faça
            armazenar (filaAberta, filho(estadoAtual));
    } até (sucesso OU vazio(filaAberta));
    se (sucesso) returne (estadoAtual);
    se não returne (NULO);
}
```

Busca em Largura

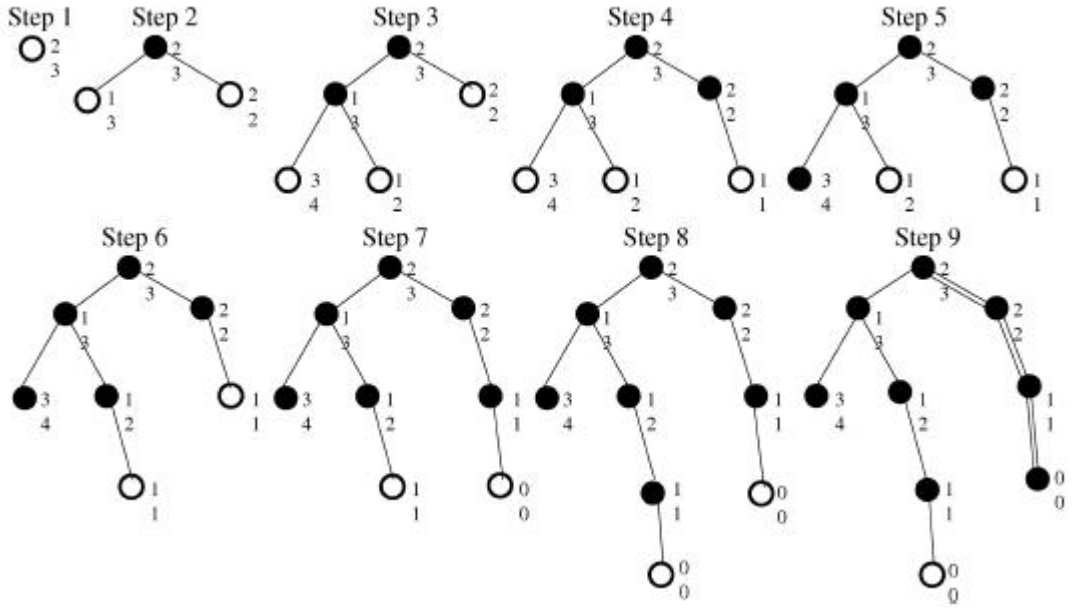
A busca em largura é uma estratégia simples emq ue o nó da raiz é expandido primeiro, em seguia todos o sucessores do nó raiz são expandidos, depois os sucessores desses nós e assim pro diante. Em geral, todos os nós em uma dada profundidade na árvore de busca são expandidos, antes que todos os nós no nível seguinte sejam expandidos.
Dessa maneira todos os nós que estão mais perto da raiz são expandidos antes de visitar aqueles que estão mais longe.
Essa estratégia faz sentido e pode ser usada quando esperados que o nosso esteja objetivo esteja próximo, em uma altura não muito profunda da árvore.

Pseudo código da busca em largura:

```
buscaEmLargura(estadoInicial) {
    armazenar(filaAberta, estadoInicial);
    sucesso = false;
    faça {
        estadoAtual = pegarPrimeiro(filaAberta);
        se ( solucao(estadoAtual))
            sucesso = true;
        se não
            para cada ( filho(estadoATual)) faça
            armazenar(filaAberta, filho(estadoAtual));

    } até ( sucesso OU vazio(filaAberta));
    se (sucesso) retorne (estadoAtual);
    se não retorne (NULO);
}
```

Ilustração da busca em largura:



Com esse simples algoritmo, é garantido que a solução achada é ótima, desde que todas as arestas (ações) tenham o mesmo custo. Caso as ações tenham valores diferentes um do outro, após ter achado uma solução, é necessário continuar expandindo todos os estados da fila com custo menores que o da solução atual.

Busca em Profundidade

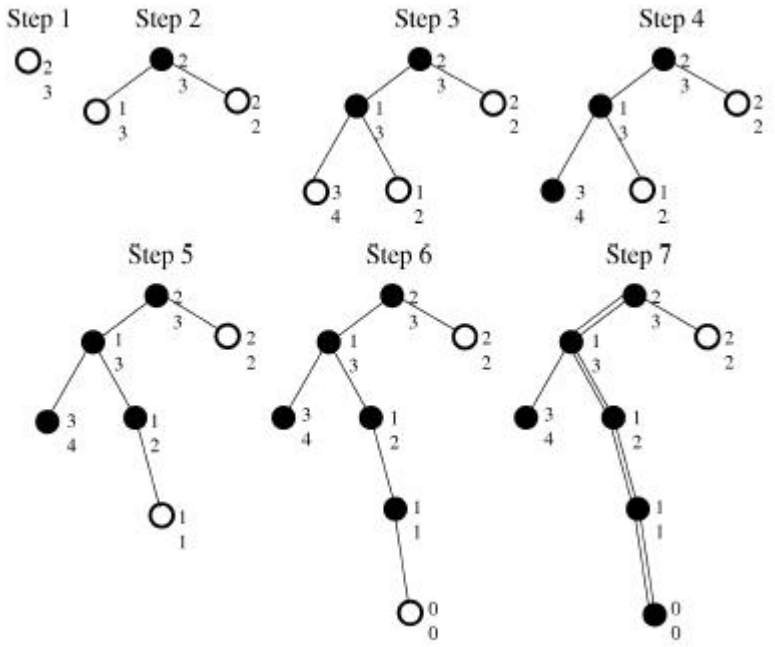
Essa busca atravessa a árvore de busca de cima abaixo, sempre expande o nó mais mais profundo na borda atual da árvore de busca. A busca prossegue imediatamente até o nível mais profundo da árvore de busca, onde os nós não tem sucessores. À medida em que esses nós são expandidos, eles são retirados da borda, e então a busca retorna ao nó seguinte mais raso que ainda tem sucessores inexplorados.
A vantagem da busca em profundidade com relação a busca em largura é que ela tem requisitos de memória muito modestos. Ela só precisa 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 [Russel & Norvig, 2006].

Pseudo código da busca em profundidade:

```
buscaEmProfundidade(estadoInicial) {
    inserir(PilhaAberta, estadoInicial);
    sucesso = false;
    faça {
        estadoAtual = pegarTopo(pilhaAberta);
        se ( solucao(estadoAtual))
            sucesso = true;
        se não
            para cada ( filho(estadoATual)) faça
            inserir(pilhaAberta, filho(estadoAtual));
```

```
    } até ( sucesso OU vazio(pilhaAberta));
    se (sucesso) retorne (estadoAtual);
    se não retorne (NULO);
}
```

Ilustração da busca em profundidade:



Colombo de certa forma usou esse algoritmo para tentar achar a Índia à oeste. Ele não desperdiçou tempo tentando achar a Índia nas proximidades da Espanha, partiu direto em direção a oeste.
A desvantagem desse algoritmo é a possibilidade de ciclos e caminhos infinitos.

Busca em profundidade Iterativa

A busca em profundidade iterativa é uma estratégia geral, 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 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 em cada nível.

a maior parte dos nós estará no nível inferior, e assim não importa muito se os níveis superiores são gerados várias vezes.

Busca em largura iterativa

A busca em largura iterativa está para a busca em largura assim como a busca em profundidade iterativa está para a busca em profundidade. O número de sucessores explorados em cada nó na árvore é restringido a uma quantidade fixa (ou número fixo) de todos sucessores. Se nenhuma solução for achada, um número maior de sucessores serão considerados a cada nó. É importante notar que o tamanho da árvore de busca é apenas diminuído, porém a árvore continua crescendo exponencialmente. Essa busca não limita a profundidade que irá buscar, dessa maneira ciclos e caminhos infinitos podem causar problemas. Para evitar problemas com requisitos de memória, poderíamos combinar a idéia do algoritmo em largura iterativa com o de profundidade iterativa. Porém, se uma iteração não achar uma solução, será difícil decidir se o algoritmo deve aumentar a largura da busca ou a sua profundidade. Poucas pesquisas foram feitas para investigar mecanismos que podem ser úteis em buscas híbridas.

Buscas Informadas

Buscas informadas funcionam de uma forma diferente das buscas desinformadas. Todas buscas apresentadas anteriormente não tem muitas informações sobre o estado atual, a não ser quantos passos ela levou para chegar até aquele estado. Buscas informadas ao contrário conseguem estimar quantos estados faltam para chegar do estado atual até um estado solução. Elas fazem isso a partir do seu conhecimento sobre o domínio. Para fazer a estimativa de quantos estados faltam para chegar no estado solução, a busca 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.

Para desenvolver uma função heurística admissível basta não usar todas as informações que temos sobre o domínio do problema, dessa maneira a heurística retornará uma estimativa
otimista da distância até o estado solução. Porém quanto menos informações do domínio for fornecida para a 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 informado depende da qualidade da função heurística, porém para calcular uma boa função heurística temos que fornecer bastante informações sobre o domínio, e quanto melhor for essa heurística menor será o erro entre h e h* e mais caro será para computar essa heurística, mas como ela melhora o desempenho da busca, os ganhos em eficiência recompensam o custo do cálculo de uma boa heurística.

Mas que tipo de busca usar para resolver o sokoban? será possível resolver um tabuleiro, não trivial, com uma busca desinformada?
Vamos considerar os tabuleiros de tamanho 20 x 20, em média tabuleiros desse tamanho tem 113 quadrados nos quais as caixas poderiam se movimentar, mas

apenas 77 desses quadrados não são deadlocks. Então em média o tamanho do espaço de busca seria de 10^18 em média.

Sokoban é um jogo difícil de resolver, o fator de ramificação é grande, uma solução pode estar em um ramo profundo da árvore de busca, a sua heurística de limite inferior é complexa e o espaço de busca é um grafo dirigido e existem estados que não tem solução.

A essa altura já é bem provável imaginar que não é possível resolver um tabuleiro de sokoban com uma busca não informada.

Outros trabalhos desenvolvidos para resolver o Sokoban

Vários trabalhos foram desenvolvidos com o intuito de resolver o Sokoban, Mark James, 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 propostos após 2 horas de processamento.
Andrew Mayers desenvolveu um programa que consegue resolver 9 problemas dos 90. O algoritmo de busca usado por ele é o A*, sua heurística de limite inferior era simples. Ele usa uma tabela de hash para armazenar os estados que já foram visitados, sua lógica leva em conta tanto o número de vezes que a caixa deve ser empurrada e o número de movimentos do agente. Esse programa não suporta macro movimentos e não considera conflitos lineares. Deadlocks de uma área de 3 x 3 são detectados automaticamente porém não são armazenados automaticamente em uma tabela de deadlocks.
Stefan Edelkamp durante seu doutorado, desenvolveu um programa que consegue resolver 13 problemas [16]. O seu programa foca em resolver o problema com um número ótimo de movimentos de caixas, usa um algoritmo sofisticado para avaliar deadlocks e também armazena 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, os estudantes A. Ueno, K. Takayama e T. Hikita desenvolveram um programa que resolve 25 dos 90 problemas. O programa é baseado no algoritmo A* e 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, um projeto feito no Japão para desenvolver novos tabuleiros de Sokoban, também tem um programa que resolve tabuleiros, esse programa é capaz de resolver 55 dos 90 problemas propostos. Ele usa uma heurística baseada no algoritmo best first search, as soluções retornadas por ele não são ótimas para número de movimentos dos homens nem para o número de movimentos das caixas. Esse programa é baseado em parte no programa que foi desenvolvido na Universidade de Meiji.
Finalmente, o melhor programa conhecido até hoje resolve 62 dos 90 problemas. Foi desenvolvido por uma pessoa que se intitula Deepgreen, seu verdadeiro nome não é conhecido. Detalhes desse programa também não é conhecido no momento. Tudo indica que esse programa é baseado em outros desenvolvidos pela comunidade japonesa do Sokoban.

Técnicas estudadas para resolução do Sokoban

As técnicas estudadas para a resolução do sokoban foram desenvolvidas por Andreas Junghanns em seu doutorado na Universidade de Alberta, Canadá. Algumas de suas técnicas foram baseadas em outras, que são usadas para resolver outros problemas, mas foram melhoradas para funcionar mais adequadamente no domínio do Sokoban.
O trabalho do Andreas foi escolhido para ser estudado pois ele implementou um dos programas mais competentes até hoje para resolver tabuleiros de Sokoban.

O nome do programa desenvolvido é **Rolling Stone** e o tipo de solução que ele otimiza é o que considera o número de empurrões das caixas, o número de movimentos do agente não é considerado.

No início do projeto, o Rolling Stone tinha sido desenvolvido para em sempre retornar soluções ótimas, mas com essa escolha o Rolling Stone conseguia resolver uma quantidade menor de problemas do que quando esse critério foi deixado de lado. Ou seja, Andreas preferiu resolver mais problemas com soluções não ótimas ao invés de resolver quantidades menores de tabuleiros.
Andreas acredita que soluções ótimas não têm grande valor se o número de problemas resolvidos for pequeno, além disso ele percebeu que as pessoas ficam felizes e motivadas quando encontram uma solução para um problema, não necessariamente uma solução ótima.

Sobre o Rolling Stone

Algoritmo de busca

O algoritmo de busca escolhido foi o IDA* que é um algoritmo de busca informada. Ele é o mais apropriado para a resolver o Sokoban pois existem poucos estados solução no espaço de busca, esses estados solução são localizados em uma profundidade grande além disso o espaço de busca é muito grande.

Resumo de como o IDA* funciona:

Heurística de limite inferior

Para desenvolver uma heurística de limite inferior foi ignorado as interações entre as caixas e o agente e a distância das caixas até a meta. Então a heurística conta somente o número de caixas que não estão nas metas. Essa heurística foi chamada de *Count.* Agora ainda ignorando as interações entre o agente e as caixas mas levarmos em consideração a distância das caixas até a sua meta mais próxima, temos uma outra heurística que foi chamada de *Closest* e também é muito barata de ser calculada.
Essas duas heurísticas são muito simples e provavelmente não vão gerar um bom

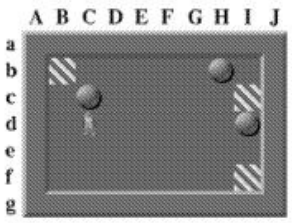
estimador de limite inferior. A cada variável que colocamos dentro de nossa heurística de limite inferior, mais próxima da realizade ela vai ficar, porém o custo para calculá-la será bem maior.

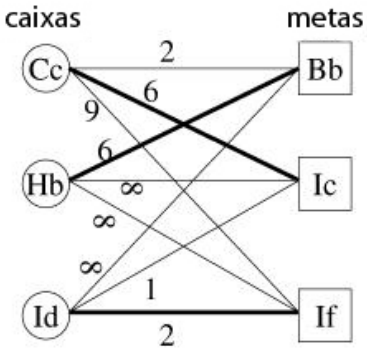
Combinação de menor custo (minmatching)

Em um tabuleiro de sokoban existem n caixas e n metas, ou seja, existe uma caixa para cada meta. Conseqüentemente existe um número, mínimo, de movimentos que levam cada caixa para uma meta específica, esse número não considera interações entre as caixas no tabuleiro ou paredes, como se o tabuleiro estivesse vazio. O objetivo da combinação de menor custo é achar uma combinação das n caixas até as n metas que minimiza a distância que teremos que empurrar as caixas para alcançarem as suas respectivas metas.

Podemos dizer que esse problema é idêntico a resolver um problema para um grafo bipartido, já que para cada caixa existe uma meta associada, e estamos tentando achar a combinação da distância mínima entre as caixas e as metas, os pesos da 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 caixas e uma meta.

Um recurso importante da combinação de menor custo é que ela é capaz de detectar deadlocks, a nossa heurística de limite inferior não é capaz de fazer isso.

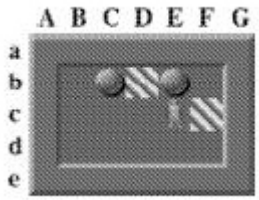




caixas em	metas em		
	Bb	*Ic*	*If*
Cc	2	**6**	9
Hb	**6**	∞	∞
Id	∞	1	**2**

Exemplo de combinação de menor custo. [1] Junghanns, A.

Exemplo de deadlock detectado pela combinação de menor custo:

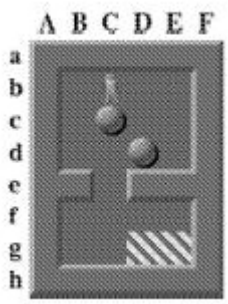


Combinação de menor custo detecta deadlock. [1] Junghanns, A.

O custo para computar a combinação 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). Nota-se que é uma computação cara.

Melhora da Entrada

Para melhorar a eficiência do cálculo da combinação de menor custo foi feita essa técnica de melhora de entrada.



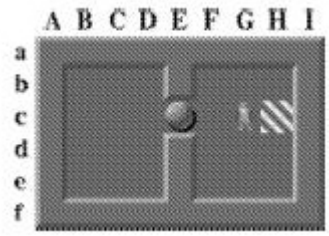
As duas caixas devem passar obrigatoriamente por Ce. [1] Junghanns, A.

Observe a figura acima, para as caixas alcançarem suas metas eas tem que passar obrigatoriamente pelo quadrado Ce, que chamaremos de entrada, E. Então ao invés da combinação de menor custo calcular a distância das caixas até suas respectivas metas, ela calculará a distância das caixas até E, pois a distância real de cada caixa até sua respectiva meta não importa se a soma das distâncias de cada caixa até sua meta é uma constante, distancia(P, M) denota a distancia da pedra P até a meta M.
distancia (P1, M1) | distancia (P2, M2) =
(distancia (P1, E) | distancia (E, M1)) + (distancia (P2, E) + distancia (E, M2)) =
= (distancia (P1, E) + distancia (E, M2)) + (distancia (P2, E) + distancia (E, M1)) =
= distancia (P1, M2) + distancia (P2, M1)

Mesmo depois dessa modificação feita, a nossa heurística de limite inferior para single agent search ainda é a mais custosa da literatura para ser calculada.
Esse truque não melhora a estimativa da nossa heurística, só melhora a performance da computação.

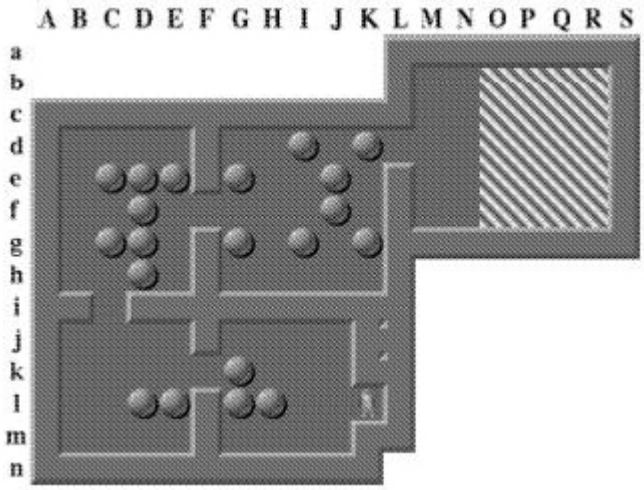
Posição do Homem

Quando a distância das caixas até suas metas são calculadas um fato muito importante estava sendo ignorado, a posição do agente no mapa, até agora estava sendo considerado que o agente poderia 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 precisará empurrar a caixa para fora de sua trajetória ótima para posicionar-se atrás da caixa e então empurrá-la em direção a uma meta [Junghanns, 1999].



A distância depende da posição do homem. [1] Junghanns, A.

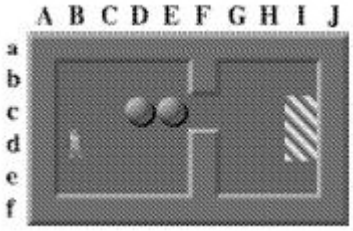
Na figura acima o agente tem que empurrar a caixa dois quadrados para trás e então empurrá-la na direção da meta. A função combinação de menor custo não levou em consideração esses dois empurrões para trás que a caixa precisou, a capacidade de detectar essa necessidade e conseguir melhorar a heurística de limite inferior é chamada de "conflito deve voltar".



O conflito deve voltar ajuda a melhorar a heurística de limite inferior para esse problema. [1] Junghanns, A.

Conflitos Lineares

Até agora parar calcular a distância de uma caixa até sua meta sempre consideramos que conseguimos empurrar a caixa, não é considerado as caixas que estão ao redor da caixa que estamos tentando movimentar, mas será que ela realmente pode sempre ser empurrada? Precisamos considerar as caixas ao redor dela?
Observe a figura abaixo:

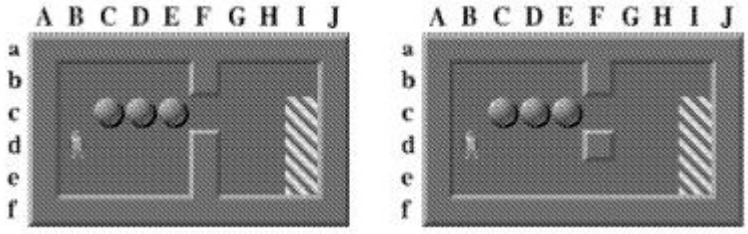


Exemplo de conflito linear. [1] Junghanns, A.

Para calcular a distância da caixa até a meta a combinação de menor custo empurraria a caixa da posição Ec para Fc, porém esse movimento é inválido, para conseguir empurrar a primeira caixa em direção à sua meta temos que tirar a caixa que está no quadrado Dc de sua trajetória ótima e então continuar com nosso plano. Esse é um exemplo de conflito linear. Quando uma situação dessas é detectada essa técnica penaliza o cálculo da heurística em 2, pois elá precisa sair de sua trajetória, e depois voltar, ou seja, fez dois movimentos a mais que o calculado pela combinação de menor custo.

Problema com Conflitos Lineares

Quando dois conflitos lineares é encontrado não podemos simplesmente penalizar o cálculo da heurística em 4. Considere a imagem abaixo:



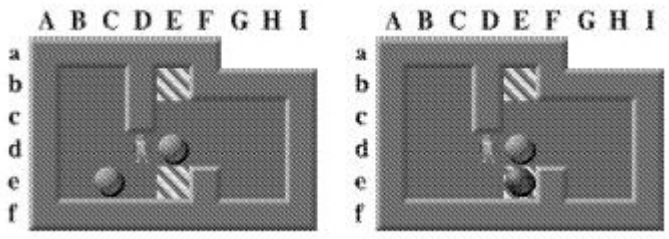
Problemas com conflitos lineares. [1] Junghanns, A.

Na situação ilustrada a esquerda 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 por causa da entrada extra que foi colocada embaixo no tabuleiro. Se a caixa do meio for empurrada para baixo as outras estarão desbloqueadas. Mas somente uma caixa pode ser empurrada para baixo, pois só existe uma meta que podemos alcançar 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.

Atualizações Dinâmicas

A combinação de menor custo é rodada antes da busca IDA*, essas distâncias representam a distância de uma caixa até outro quadrado qualquer no tabuleiro, elas

são calculadas de maneira otimista pois não levam em consideração as interações entre as caixas, só levam em consideração as paredes existentes no tabuleiro. Quando uma caixa é empurrada para um esquina ela fica lá para sempre, pois as caixas não podem ser puxadas. Se essa esquina não for uma meta então essa situação é de deadlock. Por outro lado se essa esquina for uma meta, essa caixa pode ser considerada como se fosse uma parede. Mas as paredes tem interferência no cálculo da distância das caixas até outros quadrados, por isso toda vez que uma caixa se torna fixa, em uma esquina por exemplo, o *Rolling Stone* recalcula as distâncias das caixas até os outros quadrados e faz a atualização dessas distâncias dinamicamente.



Exemplo de atualização dinâmica. [1] Junghanns, A.

Resultados do Algoritmo Rolling Stone

A tabela abaixo mostra a eficácia das técnicas desenvolvidas até agora para a heurística de limite inferior. A tabela mostra os resultados obtidos com a combinação de menor custo CC,
inclusão do conflito deve voltar CV, inclusão dos conflitos lineares CL, e todas as técnicas juntas TUDO. O limite superior LM, foi retirado do arquivo global de pontos do Sokoban, esses resultados representam a melhor solução que um humano conseguiu encontrar, por isso é um limite superior da solução ótima. A última coluna, Dif, representa a diferença entre os resultados obtidos com a heurística de limite inferior, com todas as técnicas adicionadas, e o Limite superior retirado do arquivo global, essa diferença na realidade pode ser menor, já que existe a possibilidade de uma pessoa ainda não ter alcançado a solução ótima para aquele tabuleiro. Se a diferença estiver como 0, quer dizer que a solução ótima para aquele tabuleiro não é conhecida.

#	MM	+BO	+LC	ALL	UB	Diff
51	118	118	118	118	118	0
55	118	120	118	120	120	0
78	134	136	134	136	136	0
53	186	186	186	186	186	0
83	190	190	194	194	194	0
48	200	200	200	200	200	0
80	219	225	225	231	231	0
4	331	355	331	355	355	0
1	95	95	95	95	97	2
2	119	129	119	129	131	2
3	128	128	132	132	134	2
58	189	197	189	197	199	2
6	104	104	106	106	110	4
5	135	137	137	139	143	4
60	148	148	148	148	152	4
70	329	329	329	329	333	4
63	425	425	427	427	431	4
73	433	437	433	437	441	4
84	147	149	147	149	155	6
81	167	167	167	167	173	6
10	494	506	496	506	512	6
38	73	73	73	73	81	8
7	80	80	80	80	88	8
82	131	131	135	135	143	8
79	164	164	166	166	174	8
65	181	199	185	203	211	8
12	206	206	206	206	214	8
57	215	215	217	217	225	8
9	215	227	217	229	237	8
14	231	231	231	231	239	8
62	235	237	235	237	245	8
72	284	284	288	288	296	8
77	360	360	360	360	368	8
54	177	177	177	177	187	10
56	191	193	191	193	203	10
76	192	192	194	194	204	10
47	197	197	199	199	209	10
8	220	220	220	220	230	10
27	351	351	353	353	363	10
86	122	122	122	122	134	12
44	167	167	167	167	179	12
17	121	201	121	201	213	12
59	218	218	218	218	230	12
87	221	221	221	221	233	12
43	132	132	132	132	146	14
34	152	152	154	154	168	14
71	290	290	294	294	308	14
40	310	310	310	310	324	14
35	362	362	364	364	378	14
36	501	501	507	507	521	14
41	201	219	203	221	237	16
45	274	282	276	284	300	16
19	278	282	280	286	302	16
22	306	306	308	308	324	16
20	302	444	304	446	462	16
18	90	106	90	106	124	18
21	123	127	127	131	149	18
13	220	220	220	220	238	18
31	228	228	232	232	250	18
64	331	367	331	367	385	18
25	326	364	330	368	386	18
90	436	442	436	442	460	18
49	96	104	96	104	124	20
42	208	208	208	208	228	20
61	241	241	243	243	263	20
28	284	284	286	286	308	22
68	317	319	319	321	343	22
39	650	650	652	652	674	22
46	219	223	219	223	247	24
67	367	375	369	377	401	24
23	286	424	286	424	448	24
32	111	111	113	113	139	26
16	160	160	162	162	188	26
85	303	303	303	303	329	26
89	345	349	349	353	379	26
24	442	516	442	518	544	26
15	94	94	96	96	124	28
33	140	150	140	150	180	30
26	149	163	149	163	195	32
11	197	201	201	207	241	34
75	261	261	263	263	297	34
29	124	122	124	122	164	42
74	158	172	158	172	214	42
37	220	242	220	242	290	48
88	306	334	308	336	390	54
52	365	365	367	367	423	56
30	357	357	359	359	465	106
66	185	185	187	187	325	138
69	207	217	209	219	443	224
50	96	96	96	100	370	270

Resultados das heurísticas admissíveis do Rolling Stone. [1] Junghanns, A.

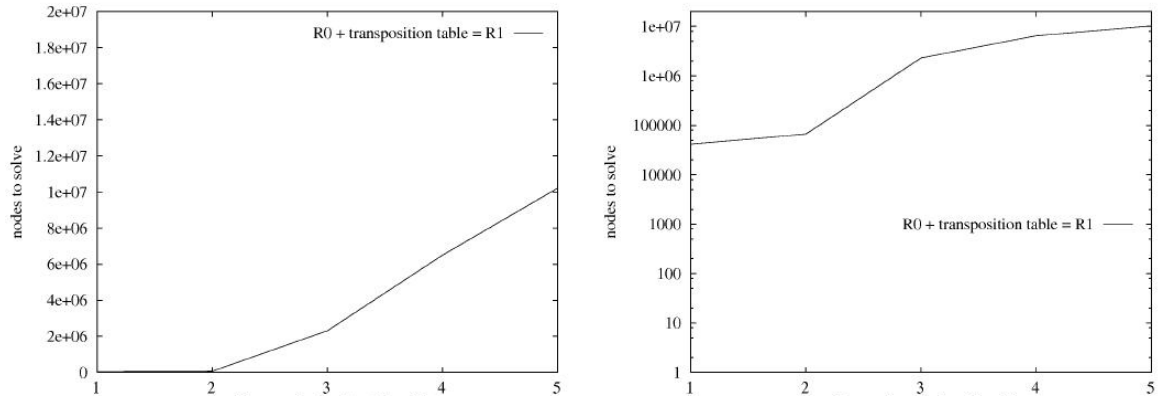
É importante dizer que mesmo depois da adição dessas técnicas avançadas de heurística de limite inferior, adicionadas a busca IDA*, nenhum dos 90 problemas propostos foi resolvido.
Um dos principais motivos é a geração de deadlock que não pode ser detectado pelo estimador e conseqüentemente faz com que a busca gaste muito tempo procurando soluções em partes gigantes da árvore de busca onde não existe solução.

Tabelas de Hash

Nesse trabalho as tabelas de hash servem para cumprir duas funções, evitar retrabalho, visitas em estados que já foram visitados, e evitar ciclos. A implementação dessa tabela tem chaves com tamanho de 64 bit usadas para criar um índice para uma tabela de hash enorme. Essa tabela de hash é organizada em dois níveis. No primeiro nível estão os estados que foram buscados profundamente, ou seja, vários filhos e netos dele já foram gerados e no segundo nível os estados gerados mais recentemente.
A chave das tabelas contém as posições exatas das caixas, para bater 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, as posições do agente não precisam ser idênticas, pois se precisassem menos estados iriam bater, e menos trabalho seria salvo com essa tabela de hash.
Uma entrada que foi buscada profundamente salvará muito trabalho se outro estado idêntico for encontrado, por outro lado as chances da busca gerar estados idênticos a estados que foram armazenados recentemente na árvore são bem maiores, armazenar esses dois tipos de estados separadamente ajuda usar essa tabela de hash de uma maneira mais eficiente.

Resultados

Com o acréscimo dessa técnica à busca IDA*, o programa Rolling Stone agora é capaz de resolver 5 problemas dos 90, foi dado um limite de expandir 20 milhões de nós. Para expandir 20 milhões de nós um computador leva cerca de 2 - 4 horas. Foi feito um segundo teste, ao invés de armazenar posições equivalentes do agente, foram armazenada posições equivalentes, mas com esse teste o número de colisões de estados caiu para menos de 10% e somente o problema número foi resolvido.



Resultados com o acréscimo das tabelas de hash no Rolling Stone. [1] Junghanns, A.

A figura acima representa o esforço que o programa fez para resolver os tabuleiros, O eixo vertical representa o número de nós expandidos para resolver o problema, e o eixo horizontal o número de problemas que foram resolvidos.
R0 representa o programa Rolling Stone com os estimadores de limite inferior mais a busca IDA*, R1 = R0 + Tabelas de Hash.

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 podemos 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.

Além disso essa técnica só é usada na última iteração do IDA* pois não faz sentido gastar processamento ordenando os movimentos dos nós internos já que teremos que percorrer eles de qualquer maneira, e como o tamanho da árvore cresce exponencialmente a cada iteração, teremos um grande ganho de performance se conseguirmos terminar a última iteração o mais cedo possível, por isso os movimentos da última iteração é que são ordenados.

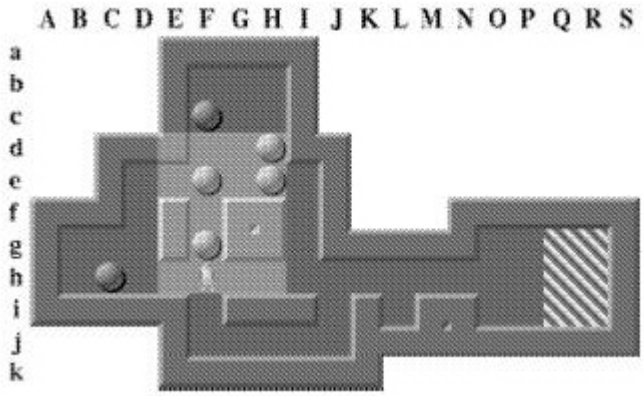
Tabelas de Deadlock

No início do desenvolvimento do Rolling Stone foi codificado várias lógicas para detectarem deadlocks que aconteciam enquanto a busca IDA* estava rodando. Porém essa técnica era capaz de pegar somente deadlocks triviais, além disso, a cada nova configuração de tabuleiros estava ficando mais difícil de codificar essas lógicas de detecção de deadlock, 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 procura nesse banco de dados para checar se na verdade não está gerando um deadlock, se não for esse movimento é gerado.
Esse programa que roda offline é usado para enumerar todas as combinações de paredes, caixas e espaços vazios em um tamanho fixo, nesse caso 5x4, para cada combinação montada, uma busca era feita para verificar se era um padrão de deadlock ou não. Essa informação é armazenada 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. Cana 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.
Duas tabelas de deadlock foram construídas para regiões de 5x4, contendo

aproximadamente 22 milhões de entradas. O que diferencia uma tabela da outra é a ordem em que os quadrados do tabuleiro são buscados, com duas maneiras diferentes de criar padrões de deadlock, mais padrões podem ser encontrados, uma vez que conflitos com quadrados de metas podem às vezes ser evitados.

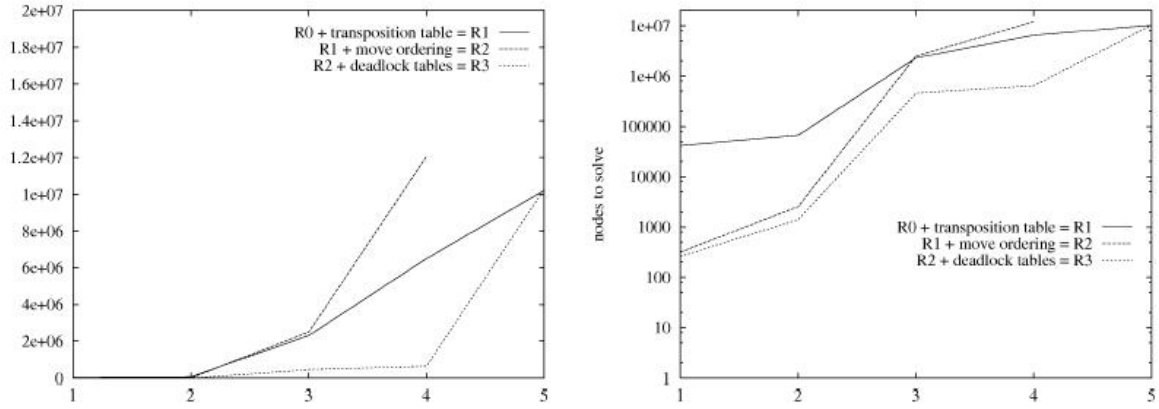
Como usar as tabelas de deadlock

Quando um movimento Ac - Ad vai ser gerado, o quadrado de destino, Ad, é usado como base na tabela de deadlock, e a direção que a caixa foi empurrada é usada para rotacionar a região. Se uma caixa for empurrada do quadrado Fh para o Fg, na figura abaixa, então a tabela de deadlock deve cobrir a área de 5x4 limitadas por Hh, Hd, Ed e Eh. Notem que a tabela pode ser usada para fazer a cobertura de outras regiões também. Para maximizar o uso das tabelas, espelhamentos de padrões assimétricos são feitos na direção em que a pedra foi empurrada.



Exemplo da cobertura das tabelas de deadlock. [1] Junghanns, A.

Resultados da adição das tabelas de deadlock:



Nós expandidos (vertical) X número de problemas resolvidos (horizontal)

Limitações

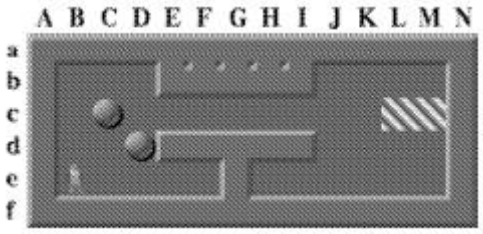
Pode parecer razoável o tamanho dos padrões de deadlock detectados por essa técnica 5x4 já que o tamanho do tabuleiro é 20x20. Mas muitos deadlocks são encontrados durante a busca IDA* e não são cobertos por essas tabelas, por outro lado é impraticável construir tabelas de deadlock com padrões maiores.
Grande parte da eficácia dessas tabelas é perdido se um padrão de deadlock cobre parte de uma área que contem metas. Uma vez que uma caixa está em uma meta ela não precisa se movimentar mais, além disso muitas regras de deadlock não se aplicam quando se trata de um quadrado meta.

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.

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.

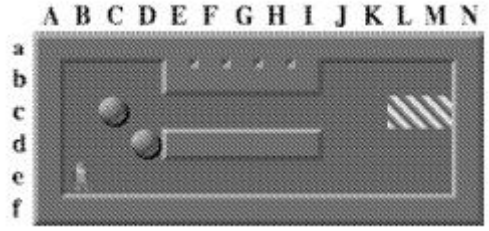


Túnel unidirecional

Se uma caixa deve ser empurrada até o fim do túnel quando ela entrar nele, não há motivos para os movimentos intermediários serem gerados. Na figura acima por exemplo a técnica de macro túnel substitui os movimentos Dc-Ec, Ec-Fc, Fc-Gc, Gc-Hc, Ic-Jc, Jc-Kc por Dc-Kc, e o contrário também Jc-Ic... é substituído por Jc-Dc.
Antes de fazer essa substituição uma checagem se o túnel não está vazia é feita, se o túnel não estiver vazio a substituição não é feita pois seria um movimento inválido. Além disso o movimento inicial também é apagado pois geraria um deadlock no tabuleiro e não seria considerado pela busca. O uso dessa técnica além de diminuir a profundidade da árvore de busca evita a geração de vários deadlocks.

Túneis bidirecionais

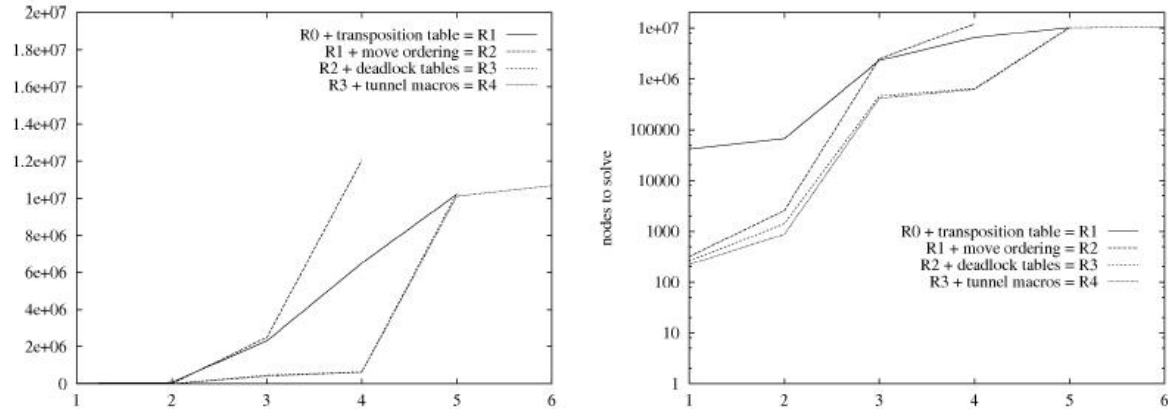
Túneis bidirecionais podem ser usados como uma área de armazenamento, ao contrário dos túneis unidirecionais que não podem, pois há pelo menos dois caminhos diferentes que ligam suas extremidades, dessa maneira o agente pode deixar a caixa dentro de um túnel bidirecional, usar o outro caminho alternativo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



Túnel bidirecional

Como um túnel bidirecional pode ser usado como uma área de armazenamento, pelo menos uma parada dentro desse túnel deve ser permitida. O lugar mais indicado para estacionar a caixa dentro do túnel seria a área mais próxima a entrada pois a estratégia é voltada para tentar fazer o menor número de movimentos com as caixas. No caso dos túneis bidirecionais também é preciso fazer uma checagem antes de substituir os movimentos intermediários dentro do túnel por um macro túnel pois uma caixa já pode estar estacionada dentro dele.

Resultados da adição do macro túnel:



Nós expandidos (vertical) X número de problemas resolvidos (horizontal)

Com a adição dos macro túneis um problema a mais foi resolvido comparado com a versão anterior do Rolling Stone.

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://docs.google.com/Doc?id=ddtqqfbt_26cwxjh9d9&hl=pt_BR

Sokoban Automatic solver, feito por Ken'ichiro Takahashi.
http://www.ic-net.or.jp/home/takaken/e/soko/index.html

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] Andreas Junghanns, Jonathan Schaeffer. Sokoban: Enhancing General Single-Agent Search Methods Using Domain Knowledge.
Artificial Intelligence, vol. 129, no. 1-2, pp. 219-251, 2001.

[3] Andreas Junghanns, Jonathan Schaeffer. Sokoban: Improving the Search with Relevance Cuts.
Journal of Theoretical Computing Science, vol. 252, no. 1-2, pp. 151-175, 2001.

[4] Andreas Junghanns, Jonathan Schaeffer. Domain-Dependent Single-Agent Search Enhancements.
Proceedings of IJCAI-99, pp. 570-575, Stockholm, Sweden, 1999.

[5] Andreas Junghanns, Jonathan Schaeffer. Relevance Cuts: Localizing the Search.
Proceedings of CG-98, Tsukuba, Japan, 1998.

[6] Andreas Junghanns, Jonathan Schaeffer. Single-Agent Search in the Presence of Deadlock.
Proceedings of AAAI-98, pp. 419-424, Madison WI, USA, July 1998.

[7] Andreas Junghanns, Jonathan Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock.
AI'98: Advances in Artificial Intelligence (R. Mercer and E. Neufeld, eds), Springer Verlag, pp. 1-15, 1998.

[8] Andreas Junghanns. Are there practical alternatives to alpha-beta?
ICCA Journal, 21(1):14-32, March 1997.

[9] Andreas Junghanns, Jonathan Schaeffer. Sokoban: A Challenging Single-Agent Search Problem.
Workshop on *Using Games as an Experimental Testbed for AI Research*, *Proceedings IJCAI-97*, Nagoya, Japan, August 1997.

[10] 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)

[11] J. Culberson. Sokoban is PSPACE-complete.
Technical Report TR97-02, Department of Computing Science, University of Alberta, 1997.

[12] Y. Murase, H. Matsubaras and Y. Hiraga. Automatic Making of Sokoban Problems.
Proceedings to PRICAI-96.

[13] D. Dor and U. Zwick. Sokoban and other motion planning problems, 1995

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

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

[16] Learning Dead Ends in Sokoban (2001)