Implementação de uma Biblioteca para Busca Informada e Não-Informada em Espaço de Estados

SILVA, D. M., FREITAS, V. M., FERNANDES JR., G. M., UCHÔA, J. Q., SCHNEIDER, B. O.

UFLA - Universidade Federal de Lavras
DCC - Departamento de Ciência da Computação
Cx Postal 37 - CEP 37200-000 Lavras (MG)
{dmsilva,vfreitas,mfernandes,joukim,bruno}@comp.ufla.br

Resumo. O objetivo desse trabalho é apresentar a implementação de uma biblioteca composta por um conjunto de classes genéricas para busca de soluções em problemas de busca informada e não-informada. Tais classes fornecem soluções rápidas para a implementação de busca em largura, busca em profundidade, busca em profundidade limitada, busca com aprofundamento iterativo, busca "gulosa" e busca A* ao programador sem que esse tenha o trabalho de "reinventar a roda", bastando para isso apenas definir algumas informações intrínsecas ao problema a ser resolvido. A fim de verificar a funcionalidade dessas classes o problema do quebra-cabeça de 16 peças foi escolhido para ser solucionado, fornecendo informações importantes para a realização desse trabalho.

Palavras-Chave: busca informada, busca não-informada, OOP, heurística, espaço de estados

1 Introdução

Alguns problemas de interesse prático em computação podem ser modelados como problemas de busca em espaço de estados. Tal abordagem apresenta métodos para solucionar o problema dado através da adoção de um objetivo a satisfazer e de uma seqüência de ações disponíveis para atingí-lo. [Russel95] apresenta uma série de exemplos de problemas que podem ser resolvidos utilizando essa abordagem.

Em geral modelar um problema de busca em espaço de estados implica na definição de algumas características inerentes ao problema. Embora tais problemas variem em complexidade e natureza, algumas informações são comuns a grande maioria deles. Tais características compreendem:

Estado inicial : consiste em uma representação de que configuração de estado inicial temos para o problema. Se o problema for encontrar o menor caminho partindo da cidade de São Paulo para a cidade de Belo Horizonte, o estado inicial deve ser a representação da cidade de São Paulo. Se o problema a ser resolvido for o problema dos missionários e canibais, então o estado inicial deve compreender a configuração inicial de missionários e canibais em cada uma das margens de um rio. Cada problema deve ter uma representação particular de seu estado inicial.

Estado final ou objetivo : uma representação da configuração de estado aceita como solução para o problema. No caso acima, o estado final para o problema dos missionários e canibais pode ser dado pela seguinte configuração: nenhum missionário e canibal na margem esquerda do rio, e todos os missionários e canibais na margem direita do rio.

Operadores : consiste em um conjunto de operações que podem ser aplicadas aos estados do problema para gerar novos estados derivados desses. Tais operações representam ações que podem ser tomadas de forma a alcançar o estado objetivo partindo do estado corrente. Novamente para o caso dos missionários e canibais existem algumas ações possíveis de serem tomadas, como por exemplos transportar um missionário para a margem direita ou esquerda do rio, transportar um canibal para a margem direita ou esquerda do rio, ou transportar duas pessoas que podem ser tanto missionários e/ou canibais para a margem esquerda ou direita do rio. Dependendo da configuração do estado corrente alguns operadores não podem ser aplicados - se a margem esquerda do rio não possuir nenhum missionário o operador "mover missionário para a

margem direita" irá gerar um estado impossível tornando a árvore de busca inconsistente.

Custo do caminho : corresponde a uma função que atribui um custo para um caminho. Geralmente o custo de um caminho é dado pela soma do custo das ações individuais que formam esse caminho. Um caminho é dado por uma seqüência de nós que se inicia no nó raiz e termina no nó corrente da árvore de busca.

Esse trabalho apresenta uma biblioteca de programação desenvolvida para a linguagem C++ capaz de permitir uma rápida implementação de programas que resolvem problemas de busca em espaço de estados com aplicações. Tal biblioteca fornece um conjunto de classes genéricas capazes de resolver o problema dado utilizando as seguintes estratégias de busca informada e não-informada: busca em largura, busca em profundidade, busca em profundidade limitada, busca com aprofundamento iterativo, busca "gulosa" e busca A*.

Através do mecanismo de herança fornecido pela linguagem o usuário define apenas as informações intrínsecas ao problema a ser solucionado, como por exemplo a representação dos estados e as informações do problema - estado inicial, estado final, operadores disponíveis e o custo do caminho. O uso de classes abstratas pela ferramenta orienta o programador a implementar cada uma dessas características necessárias a realização do processo de busca selecionado.

A biblioteca foi implementada utilizando-se a linguagem de programação C++ e testada com o compilador `GNU g++` versão `2.96` presente na distribuição `Linux Red Hat 7.2` com kernel versão `2.4.9-31`. Os fundamentos teóricos utilizados na implementação foram encontrados em [Russel95] e [Nilsson98]. Detalhes sobre a implementação serão dados na seção 3.3.

2 Estratégias de Busca

2.1 Busca Não-Informada

As estratégias de busca não-informada, também conhecida como busca cega ou *blind search* são estratégias de busca baseadas em tentativas de solução por força bruta onde o único conhecimento que pode ser aplicado ao problema para determinar o próximo passo rumo a uma solução é dado por uma função de enfileiramento. Tais estratégias podem encontrar uma solução para o problema simplesmente gerando novos estados e testando-os contra o objetivo.

Dentre estas estratégias as mais importantes são conhecidas como busca em largura (*breadth search*), busca em profundidade (*depth search*), busca em profundidade limitada (*depth limited search*) e busca com aprofundamento iterativo (*iterative deepening search*). As próximas seções descrevem em mais detalhes cada tipo de busca não-informada, apresentando características que diferenciam-nas e que desempenham papel importante na determinação do quão rápido o processo de busca converge para uma provável solução.

2.1.1 Busca em Largura

Nesse tipo de estratégia, o nó raiz é o primeiro nó a ser expandido. Após gerar todos os nós filhos possíveis, a partir do estado pertencente ao nó raiz, esses são expandidos. Somente após a expansão de todos os nós pertencentes ao mesmo nível de profundidade na árvore de busca é que os nós do próximo nível serão expandidos. Um exemplo deste processo pode ser visto na Figura 1.

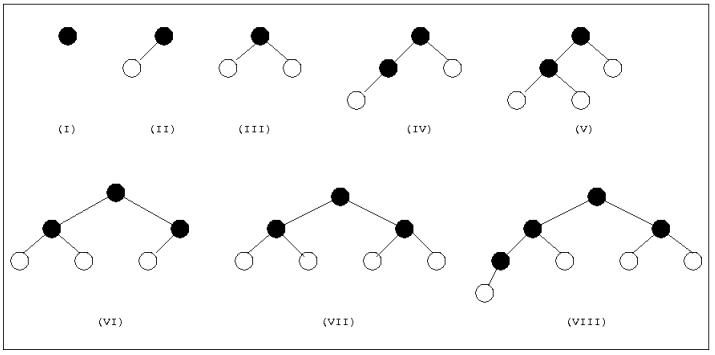

Figura 1: Expansão dos nós gerados pela estratégia de busca em largura

O detalhe mais importante de sua implementação está na função de enfileiramento adotada. A técnica utilizada consiste em armazenar os nós filhos gerados por um determinado nó em uma fila. Seguindo a convenção de que um nó somente pode ser removido do início da fila, a ordem de expansão dos nós corresponderá à especificada pela busca em largura. O processo de busca continua até que um nó cujo estado corresponde ao estado final seja removido do início da fila, encerrando esse processo de busca.

Esse tipo de estratégia possui a particularidade de sempre encontrar uma solução se ela existir. Caso existam diversas soluções, uma busca em largura no espaço de estados sempre encontra a melhor solução possível sendo classificada como *completa* por encontrar um caminho que leva a solução e *ótima* caso o custo do caminho for uma função decrescente da profundidade de um nó. A condição de ótima pode ser explicada com base na característica de considerar primeiro todos os caminhos de tamanho 1, depois todos os caminhos de tamanho 2, e assim sucessivamente de forma que quando encontrar uma solução ela será a que possui menor seqüência de ações para atingir o estado objetivo.

Embora seja uma estratégia de busca completa e ótima, utilizá-la nem sempre é uma boa idéia se forem consideradas as quantidades de tempo e memória consumidas pelo processo de busca. Considerando a hipótese de que cada estado expandido pela busca gera b novos estados então o nó raiz irá gerar b estados no primeiro nível da árvore de busca. Cada nó expandido também irá gerar b nós dando um total de b^2 nós no segundo nível da árvore de busca. Da mesma forma cada um destes nós irá gerar novamente b nós totalizando b^3 nós no terceiro nível da árvore de busca, e assim sucessivamente. Supondo que a solução para este problema possua caminho de tamanho d então o número máximo de nós que devem ser expandidos antes de encontrar a solução será:

$$1 + b + b^2 + b^3 + ... + b^d \quad (1)$$

Desta forma, a análise de complexidade desta estratégia de busca mostra que a complexidade de tempo é dada por $O(b^d)$. Podemos considerar a mesma complexidade para o espaço uma vez que cada nó expandido deve estar armazenado em memória. Em [Russel95] pode-se encontrar uma tabela comparativa de caminhos que levam a uma solução com diferentes tamanhos, dando uma noção do quanto pode ser custoso utilizar esta estratégia quando a busca demora a convergir para uma solução com base no estado inicial do problema.

2.1.2 Busca em Profundidade

Nessa estratégia de busca um dos nós do nível mais profundo da árvore de busca sempre é expandido. Somente quando a busca atinge um nó cujos filhos possuem estados que já foram expandidos outrora é que a busca expandirá algum dos nós pertencentes aos níveis acima deste, processo este conhecido como *backtrack*. A Figura 2 ilustra essa estratégia de busca.Uma forma de garantir esta ordem de expansão é adicionar os nós filhos que serão expandidos em uma pilha, onde o primeiro nó removido será o primeiro filho gerado de onde a busca continua aprofundando-se até não mais ser possível ou retirar da pilha um nó cujo estado seja correspondente ao estado objetivo.

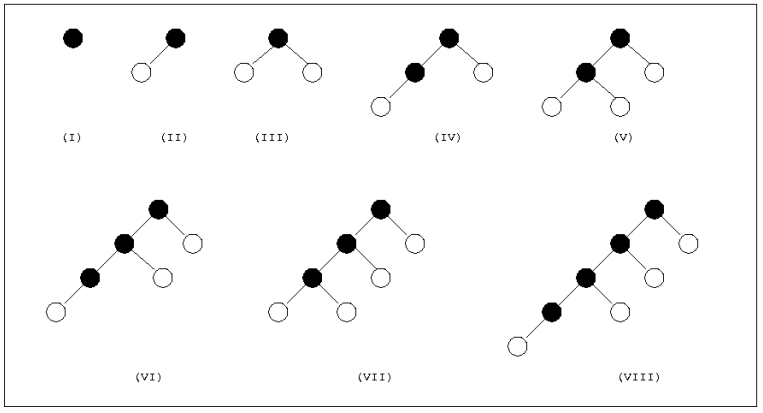

Figura 2: Expansão dos nós gerados pela estratégia de busca em profundidade

Diferente da estratégia de busca em largura, a busca em profundidade utiliza menor quantidade de memória para armazenar os nós expandidos do que a busca anterior uma vez que armazena somente um único caminho do nó raiz até um nó folha mantendo os demais nós não expandidos enquanto aprofunda pelo caminho. Segundo [Nilsson98], se cada nó expandido gera b nós e a árvore de busca possui profundidade máxima m então a busca em profundidade requer que estejam armazenados bm nós na memória fornecendo uma complexidade de espaço da ordem de $O(bm)$. A complexidade de tempo é da ordem de $O(b^m)$ criando um grande contraste com a complexidade de tempo de $O(b^d)$ dado pela busca em largura.

Para problemas que possuem muitas soluções a busca em profundidade tende a encontrar uma solução mais rápido que a busca em largura devido a possibilidade de procurar uma solução explorando apenas uma pequena parte do espaço de estados. Porém, como sempre aprofunda em um determinado ramo da árvore de busca pode ocorrer de existir uma melhor solução nos ramos que possuem menor profundidade na árvore de busca e que dificilmente serão explorados, fazendo com que as soluções encontradas pela busca em profundidade não sejam na maioria das vezes soluções ótimas. Outro problema consiste na possibilidade do método de busca fazer uma escolha não muito boa de forma que a busca continue sempre aprofundando sem retornar para os ramos de menor profundidade mesmo que uma solução exista em outra região do espaço de estados levando a uma *busca infinita*. As características anteriormente citadas fazem com que a busca em profundidade seja um método mais econômico em termos de requerimento de tempo e memória, porém não se trata de uma estratégia ótima ou completa podendo encontrar soluções de custo elevado ou simplesmente não encontrar solução, mesmo se existir alguma.

2.1.3 Busca com Profundidade Limitada

Essa estratégia de busca possui um funcionamento semelhante ao da busca em profundidade. A diferença consiste em colocar um "corte" na profundidade máxima de um caminho na árvore de busca. Quando um nó removido da pilha estiver na máxima profundidade permitida ele não gera filhos, fazendo com que a pilha de nós não seja modificada. Assim o próximo estado a expandir é aquele contido no nó presente no topo da pilha que pertence a um nível de profundidade menor do que o nó anterior na árvore de busca.

Devido ao corte imposto pela busca esta estratégia garante encontrar uma solução se ela existir dentro da profundidade limite. Porém não é garantido que esta solução seja a melhor possível tornando este método de busca completo desde que a solução esteja dentro das restrições impostas, mas não ótimo. A complexidade de tempo e espaço são similares as complexidade da busca em profundidade padrão, sendo $O(b^l)$ a complexidade de tempo e $O(bl)$ a complexidade de espaço para l igual a profundidade limite.

2.1.4 Busca com Aprofundamento Iterativo

Uma busca com aprofundamento iterativo tenta encontrar a solução para o problema dado realizando uma busca com profundidade máxima de i. Caso não exista solução para essa profundidade, a busca é reiniciada tendo agora profundidade máxima $i + 1$, prosseguindo desta forma até que uma solução seja encontrada.

Devido a necessidade de recomeçar a busca do nível 1 aumentando a profundidade limite a cada tentativa sem êxito de encontrar uma solução a busca com aprofundamento iterativo tende a consumir muitos recursos de tempo e espaço. Como visto na seção 2.1.3 para uma busca em profundidade limitada com profundidade d e número de nós gerados por uma expansão igual a b, considerando o pior caso onde a busca se aproxima a uma busca em largura padrão o número total de nós gerados será igual a:

$$1 + b + b^2 + ... + b^d \tag{2}$$

Na busca com aprofundamento iterativo os nós das regiões mais profundas da árvore de busca são expandidos uma vez, na próxima tentativa os nós desta região são expandidos duas vezes e assim a expansão continua até encontrar uma solução. Como o nó raíz é expandido $d + 1$ vezes o número total de expansões ocorridas será:

$$(d+1)1+(d)b+(d-1)b^2+...+3b^{d-2}+2b^{d-1}+1b^d \tag{3}$$

De [Nilsson98] encontra-se a seguinte fórmula fechada para o cálculo do número máximo de expansões realizadas por uma busca com aprofundamento iterativo. Da Equação 2, tem-se que o número máximo de expansões para uma busca em profundidade limitada quando a mesma se aproxima da busca em largura é:

$$1 + b + b^2 + ... + b^d = \frac{b^{d+1} - 1}{b - 1} \tag{4}$$

Supondo o número total de nós expandidos por uma busca em profundidade limitada até a profundidade j ser N_j, para uma solução encontrada na profundidade d

tem-se que o número total de nós expandidos por uma busca com aprofundamento iterativo será:

$$N_d = \sum_{j=0}^{d} \frac{b^{j+1}-1}{b-1} = \frac{1}{b-1}\Big[b\Big(\frac{b^{d+1}-1}{b-1}\Big) - (d+1)\Big] \tag{5}$$

Simplificando a expressão dada pela Equação 5 tem-se a fórmula fechada que é dada pela Equação 6.

$$N_d = \frac{b^{d+2} - 2b - bd + d + 1}{(b-1)^2} \tag{6}$$

Essa quantidade de nós que é expandida a mais a cada tentativa é pequena em relação a uma busca em profundidade limitada, fazendo com que a complexidade de tempo da estratégia de busca com aprofudamento iterativo seja da ordem de $O(b^d)$ enquanto que a complexidade de espaço é da ordem de $O(db)$. Tais ordens de complexidade fazem com que o método seja muito utilizado quando o espaço de busca é muito grande e a profundidade de uma solução não é conhecida.

2.2 Busca Informada

As estratégias de busca informada diferem das estratégias de busca não informada por acrescentarem uma informação a mais na determinação da ordem de expansão dos nós durante o processo de busca. Esta informação é chamada de *função de avaliação* e consiste em uma forma de mensurar a probabilidade de um nó convergir para uma solução baseado em seu estado corrente.

Esta medida de probabilidade pode ser dada por uma função $h(n)$ que mede a estimativa de um dado nó - em outras palavras, o quanto falta para atingir o estado objetivo partindo do estado deste nó. Outra medida associada é dada pela soma de $h(n)$ com $g(n)$, onde $g(n)$ mede o custo de percorrer o caminho que passa por este nó partindo do nó raiz.

As estratégias mais conhecidas de busca informada são a busca "gulosa" (*greedy search*) que utiliza somente a estimativa do nó como função de avaliação, e a busca A* (*A* search*) que utiliza a soma da estimativa do nó com seu custo para definir esta avaliação. Em geral tais estratégias são variações da busca em profundidade. A diferença consiste em determinar de qual nó folha deve-se continuar a busca - geralmente o nó que possui menor avaliação. A Figura 3 ilustra a diferença entre estes dois tipos de busca informada utilizando como exemplo a mesma instância para a árvore de busca. Os nós coloridos representam aqueles que já forma previamente expandidos, enquanto que os demais representam nós filhos que podem ser expandidos. Esses nós possuem como rótulo uma letra para melhor exemplificar a ordem presente na lista de nós abertos para cada uma das estratégias. O valor numérico presente dentro dos nós a expandir é o valor da *estimativa* para alcançar o estado objetivo. A árvore que representa a busca A* contém ainda outro valor numérico, representado o *custo* para gerar este caminho. Abaixo de cada árvore está a lista de nós abertos ordenada por meio da função de avaliação heurística para cada uma das estratégias. A ordem apresentada é a ordem na qual os nós filhos serão posteriormente expandidos.

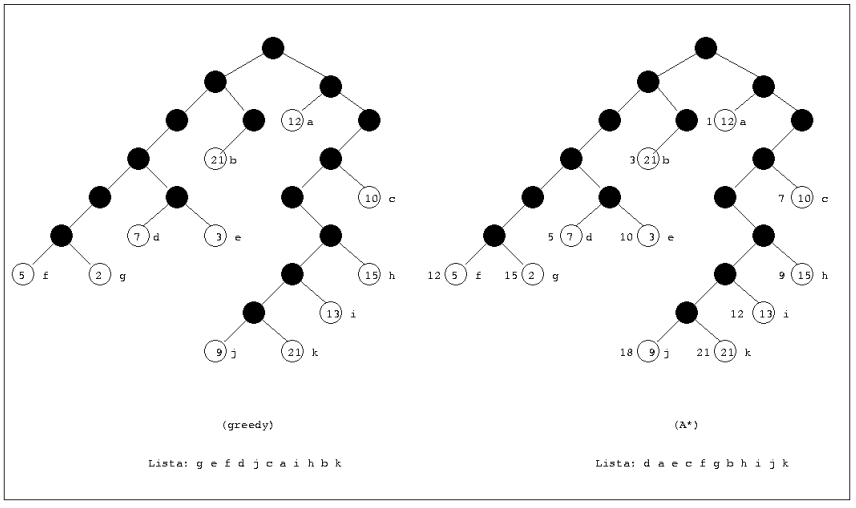

Figura 3: Ordem de expansão dos nós para a busca "gulosa" e A*

A estratégia de busca "gulosa" possui os mesmos defeitos de uma busca em profundidade padrão. Ela não é ótima pois leva em consideração apenas a informação da *estimativa* de um nó e não o *custo* ou *profundidade* que este possui, e também não é completa pois pode aprofundar-se em um caminho e nunca retornar para tentar outras possibilidades. A complexidade de tempo do pior caso para a busca gulosa é da ordem de $O(b^m)$ onde m é a profundidade máxima do espaço de busca. Sua complexidade de espaço também é da ordem de $O(b^m)$ pois retêm todos os nós expandidos na memória.

A estratégia de busca A* é considerada ótima e completa. Uma demonstração da otimalidade e completude deste método de busca é encontrado em [Russel95], assim como um breve comentário sobre a complexidade do método.

Dependendo de como a função de avaliação heurística é definida as complexidades de espaço e de tempo podem cair drasticamente ao fazer a busca convergir para uma provável solução de uma forma rápida e segura.

3 Implementação da Biblioteca

3.1 Estruturas de Dados Utilizadas

Diversas estruturas de dados foram utilizadas no decorrer da implementação desse conjunto de classes com diferentes propósitos. Citaremos as principais e sua importância no funcionamento da ferramenta.

3.1.1 Representação dos Estados

Como as buscas realizadas pela biblioteca baseiam-se em estados, é de fundamental importância que as estruturas de dados utilizadas para representá-los sejam as mais eficientes possíveis. Diversas estruturas de dados conhecidas podem ser utilizadas com diferentes propósitos na definição de um estado. Por exemplo, para o problema do quebra-cabeça de 16 peças implementado nesse trabalho, cada estado é representado por uma matriz de inteiros de dimensões 4x4, onde cada elemento da matriz representa uma peça do quebra-cabeça. Para outros tipos de problemas como é o caso do problema de encontrar o menor caminho entre duas cidades a abstração utilizada para representar cada estado possível pode ser um valor inteiro que indique o número da cidade se for feita a consideração de que cada cidade do mapa possui um valor único. Outro exemplo pode ser dado para o problema dos missionários e canibais onde cada estado representa uma configuração das duas margens do rio. Nesse caso, uma estrutura de dados adequada deve ser utilizada para representar a quantidade de missionários e de canibais em cada uma das margens.

3.1.2 Representação do Problema

O problema define algumas informações vitais para a realização das buscas como por exemplo um estado inicial e um estado final, além do conjunto de operadores válidos que podem ser aplicados a um estado para gerar estados filhos. No entanto, além dessas informações básicas talvez seja necessário implementar alguma estrutura de dados adicional na classe que define o problema a fim de armazenar alguma nova informação instrínseca à ele. Por exemplo, para o problema de encontrar o menor caminho entre duas cidades seria interessante manter um grafo ou uma tabela que represente a distância entre as cidades vizinhas. Da mesma forma alguns problemas requerem custos diferentes para gerar um novo estado a partir de um já existente. Tais problemas devem possuir a informação sobre o custo de gerar este novo estado em alguma estrutura de dados, como por exemplo uma tabela ou *array*.

3.1.3 Estruturas de Dados Utilizadas pela Busca

Como visto, o usuário dessa biblioteca possui a flexibilidade de utilizar estruturas de dados como desejar para melhor definir seus problemas. No entanto essas estruturas não são visíveis pelas classes de busca e a única forma de comunicação entre o que foi definido pelo usuário e o que as classes de busca requerem é dado através de *interfaces* definidas por meio de *métodos virtuais*. Tais métodos devem ser implementados pelos usuários durante a construção de suas classes *problema* e *estado*.

As classes de busca apresentam algumas estruturas de dados que são a chave para o controle de quais nós já foram percorridos durante a busca. As estruturas de dados utilizadas para esse propósito são duas listas: a lista de nós abertos e a lista de nós fechados. Embora o protocolo dessas estruturas seja extremamente flexível no que diz respeito ao acesso e remoção de seus elementos, nessa implementação tais listas estarão restritas a possuir comportamento semelhante ao de estruturas de dados que utilizam aos protocolos `FIFO` e `LIFO` a fim de emular o funcionamento de uma fila ou pilha de acordo com a ocasião ou estratégia de busca utilizada.

A lista de nós fechados compreende uma lista onde são armazenados todos os nós que já foram expandidos durante a busca. É de vital importância mantê-los organizados nessa lista a fim de evitar a geração de nós que contenham estados já gerados, impedindo desta forma que a busca entre em *loop infinito*. Além disso, ao verificar que um determinado estado já existente em algum nó da lista de nós fechados pode ser gerado por uma seqüência menor de operações - isto é, a profundidade desse nó é menor que a profundidade de outro nó na lista

de nós fechados que contenha o mesmo estado - é possível atualizar essa nova informação no nó correspondente da lista de nós fechados, tornando a solução mais eficiente ao diminuir a quantidade de operações necessárias para atingir o estado objetivo.

A lista de nós abertos contém os nós que ainda não foram expandidos - ou seja, os nós folhas. Essa estrutura de dados não é uma lista propriamente dita, pois permite apenas extrair informações do primeiro nó da lista. No entanto não segue obrigatoriamente o protocolo `FIFO` no momento da inserção de novos nós. A ordem e posição que estes nós são inseridos definem as estratégias de busca implementadas como visto na seção 2. Os nós utilizados na árvore de busca merecem atenção especial em se tratando de estruturas de dados utilizadas para representá-los, tamanha a importância das informações que ele encapsula. Existem basicamente cinco informações geradas pelo processo de busca que cada nó deve conter, a citar:

Estado corrente : representa o estado no espaço de estados ao qual o nó corresponde. É necessário armazenar essa informação pois é através dela que a busca determina se o estado objetivo foi atingido ou não mediante comparação, encerrando o processo.

Nó pai : armazena qual nó da árvore de busca gerou esse nó. Por meio dessa informação aliada a um nó que contenha o estado final é possível montar uma seqüência de operações aplicadas a partir do estado inicial do problema que levam à solução ou estado objetivo.

Operador : operador definido no problema que foi aplicado para gerar esse nó. Essa informação é recuperada pela montagem da seqüência de passos comentada acima.

Profundidade : guarda o número de nós a percorrer da raiz até esse nó. Sua importância está na otimização da solução, comentada acima.

Custo : armazena o custo de geração desse estado pelo nó pai. Essa informação é de grande relevância para os métodos de busca informada A* e busca "gulosa".

3.2 Algoritmos Utilizados

Alguns algoritmos clássicos de ordenação tiveram que ser implementados devido ao fato de que algumas estratégias de busca necessitam que os nós da lista de nós abertos estejam ordenados pela estimativa. Uma primeira tentativa de implementação consistiu no uso do algoritmo de seleção. Devido à complexidade assintótica desse algoritmo ser da ordem de $O(n^2)$ e da quantidade de nós ser muito grande para problemas cujo estado inicial está muito distante do estado final em termos de estimativa, procurou-se uma solução capaz de realizar a ordenação em tempo melhor que $O(n^2)$. A escolha foi implementar o algoritmo *quicksort*. Embora em média o algoritmo *quicksort* apresente complexidade da ordem de $O(nlogn)$, o fato da lista de nós abertos ser reordenada cada vez que novos nós gerados são introduzidos na mesma levou a ordenação a apresentar comportamento semelhante a algoritmos com complexidade $O(n^2)$, além da quantidade de memória consumida para realizar a recursão crescer com o tamanho da lista de entrada. Outra solução proposta foi utilizar o algoritmo da *bolha* que, embora seja considerado um dos piores senão o pior em termos de complexidade, com uma pequena modificação no código padrão é possível obter uma ordenação otimizada - a lista de nós abertos está semi-ordenada pela última ordenação, de forma que o algoritmo da *bolha* ordena apenas os novos nós inseridos e aborta a operação ao verificar que o restante da lista está ordenada. Maiores informações sobre esses métodos de ordenação assim como outros algoritmos e estruturas de dados úteis podem ser encontrados em [Cormen99] e [Ziviani96].

Embora a biblioteca apresente diversas estratégias de busca implementadas, os algoritmos utilizados possuem a mesma estrutura básica que pouco varia de classe para classe. Abaixo é apresentado o algoritmo de busca genérico em pseudo-código, extraído de [Russel95]:

```
função BUSCA-GENERICA(problema, ENFILEIRA-FN) retorna uma solução ou falha

lista-abertos <- FAZER-FILA(FAZER-Nó(ESTADO-INICIAL[problema]))

fazer
    se lista-abertos está vazia então retorna FALHA

    nó <- REMOVER-PRIMEIRO(lista-abertos)
```

```
    se TESTAR-OBJETIVO[problema] aplicado em ESTADO[nó] obtêm sucesso então
        retornar nó

    lista-abertos <- ENFILEIRA-FN(lista-abertos,
                                  EXPANDIR(nó, OPERADORES[problema]))
fim fazer
```

O código acima mostra uma função de busca genérica que recebe um problema e uma função de enfileiramento. Inicialmente expande-se o estado inicial gerando os filhos do nó que contém o estado inicial. Após isso, uma série de testes é realizado dentro de um *loop* que levará ao estado final ou a uma situação onde não exista solução.

Se a lista de nós abertos estiver vazia não existe solução para esse problema. No entanto, se for possível remover o primeiro nó da lista então um teste é realizado nesse nó para verificar se ele contém o estado final. Se positivo então esse nó é retornado pela busca, tornando possível trilhar o caminho inverso da solução através do atributo *nó pai*. Caso negativo, então esse nó é expandido e seus nós filhos são inseridos na lista de nós abertos utilizando a função de enfileiramento especificada. O processo se repete até que não existam mais nós na lista de nós abertos ou então um nó removido dessa lista contenha o estado final ou objetivo.

Da mesma forma para a implementação dos métodos de busca informada um novo algoritmo de busca extraído de [Russel95] foi utilizado:

```
função BUSCA-INFORMADA(problema, AVALIACAO-FN) retorna uma seqüência solução
    entradas: problema, o problema a resolver
              AVALIACAO-FN, uma função de avaliação

    ENFILEIRA-FN <- uma função que ordena nós por meio de AVALIACAO-FN

    retorne BUSCA-GENERICA(problema, ENFILEIRA-FN)
```

A função acima representa um algoritmo onde conhece-se um dado problema e uma função heurística de avaliação que será utilizada pela função de enfileiramento a fim de ordenar os nós inseridos na lista em ordem de *estimativa* ou *estimativa + custo*. O restante da busca é realizado como em `BUSCA-GENERICA()`, onde a função de enfileiramento irá ordenar todos os nós a expandir em ordem da função heurística de avaliação.

As implementações das busca informada e não informada baseiam-se nos algoritmos dados acima, alterando as características de cada um a fim de atender as diferentes estratégias de busca propostas.

3.3 Implementação em C++

A implementação dessa biblioteca utilizando a linguagem C++ merece especial destaque neste documento. O uso dessa linguagem foi adotado devido às características presentes na linguagem que puderam facilitar a implementação de uma estrutura de busca genérica. A facilidade de utilizar-se funções *virtuais puras* além do uso de *templates* e orientação à objetos - principalmente no que diz respeito ao mecanismo de herança - tornaram a implementação simples. Informações sobre orientação a objetos usando C++ são encontrados em [Deitel97].

A estratégia utilizada para implementar cada um dos tipos de busca foi definir algumas classes básicas que contenham atributos e métodos que todas as estratégias de busca utilizam - como por exemplo as listas de nós abertos e fechados. Dessa forma uma hierarquia de classes foi criada onde classes mais especializadas herdam as características das classes básicas, realizando poucas alterações na redefinição ou complementação de suas características - reutilização do código permitido pelo paradigma de orientação a objetos. As classes utilizadas nessa implementação serão discutidas mais detalhadamente nas seções de 3.3.1 até 3.3.10, sendo que agora será dado apenas uma breve descrição do que cada uma contém.

A classe `clGeneralSearch` contém o escopo básico utilizado por todas as demais classes que dela derivam. Nela são implementadas as estruturas de dados acima mencionadas além da maior parte dos métodos que atuam como interface entre as classes. Muitos métodos definidos nessa classe são *métodos virtuais puros* que devem ser implementados nas classes derivadas, fa-

zendo com que `clGeneralSearch` seja uma classe abstrata - ou seja, não permite que objetos sejam instanciados diretamente dela. Duas classes herdam diretamente a classe `clGeneralSearch`. Cada uma delas implementa uma estratégia de busca diferente, que é propagada para as classes que delas derivam. A classe `clBreadthSearch` implementa a estratégia de busca em largura, enquanto que a classe `clDepthSearch` implementa a estratégia de busca em profundidade. Ambas estratégias já foram citadas nas seções 2.1.1 e 2.1.2.

A classe `clDepthSearch` possui grande importância para essa biblioteca, pois dessa classe derivam quatro classes que implementam variações desse tipo de busca. A classe `clDepthLimitedSearch` implementa uma busca em profundidade limitada, onde impõe-se um limite máximo para o aprofundamento de cada ramo da árvore de busca. Da mesma forma a classe `clIterativeDeepeningSearch`, derivada da classe de busca `clDepthLimitedSearch`, implementa outra estratégia de busca semelhante a anterior com a diferença que a profundidade máxima permitida é incrementada a cada tentativa de solução sem êxito.

Existem ainda as classes `clGreedySearch` e `clAAsteriskSearch` que derivam de `clDepthSearch`. Essas duas classes implementam métodos de busca informada que por meio de heurísticas tendem a encontrar soluções para os problemas de busca com menor tempo e consumo de memória. O diagrama de herança das classes de busca é dado na Figura 4 para melhor ilustrar a implementação.

Existem ainda duas classes genéricas cuja estrutura é aproveitada pelos usuários desse conjunto de classes através do mecanismo de herança. Essas classes definem a interface entre o problema e os estados definidos pelos usuários e os algoritmos de busca propriamente ditos. Tais classes são `clGenericState` e `clGenericProblem` que também são classes abstratas como a `clGeneralSearch`. Para utilizar essa biblioteca, o usuário deve construir duas classes que herdam as características das classes genéricas que definem o problema e estado, redefinir os métodos abstratos de *interface* e instanciar um objeto da classe de busca escolhida conforme a estratégia desejada. Esse objeto possui dois parâmetros *templates* que compreeendem as classes problema e estado recém definidas pelo usuário. Após a instanciação a busca pode ser iniciada, pois todas as informações necessárias para sua realização já foram definidas.

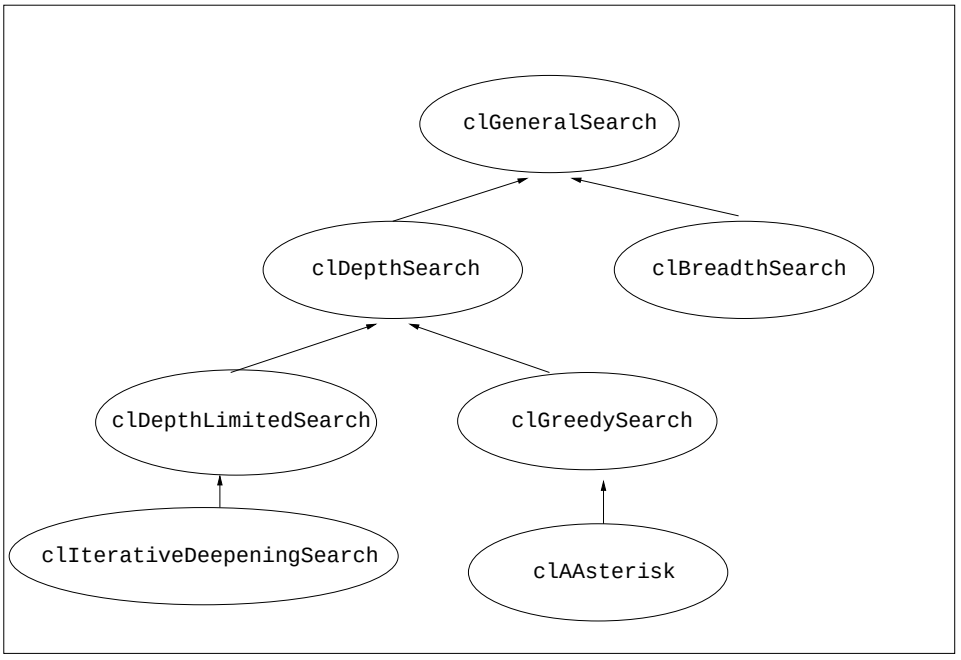

Figura 4: Hierarquia de Classes de Busca

A seguir uma descrição mais detalhada a respeito de cada uma das classes de busca é dada, incluindo comentários sobre os atributos e métodos além de algumas explicações sobre as decisões de implementação mais importantes.

3.3.1 A classe `clGeneralSearch`

A classe `clGeneralSearch` é a classe mais geral de toda a biblioteca. Compreende um conjunto básico de funções de *interface* entre o usuário e os métodos de busca. É uma classe abstrata, impedindo que um objeto seja instanciado diretamente dela. Alguns métodos virtuais são definidos e implementados nessa classe, porém são redefinidos posteriormente para melhor representar o comportamento dos diferentes métodos de busca que dessa classe dependem.

Existem três atributos na classe `clGeneralSearch` - a lista de nós abertos, a lista de nós fechados e o problema. A lista de nós abertos é um objeto da classe `vector` presente na biblioteca `STL (Standard Template Library)`. A lista de nós fechados é uma tabela *hash* implementada pela classe `clHashTable`, enquanto que o problema é um tipo de dados definido pelo usuário na classe que herda `clGenericProblem`, discutida adiante. A lista de nós abertos armazena os nós que ainda não foram expandidos na árvore de busca - ou seja, os nós folhas - enquanto que a lista de nós fechados armazena os nós que já foram expandidos a fim de evitar a duplicação dos nós gerados e otimizar a busca. O problema define algumas informações importantes na busca pela solução. Nele é encontrado informações sobre os estados inicial e final do problema, os operadores que podem ser aplicados a um determinado estado e o custo de gerar um estado a partir de outro. Essas informações são de grande relevância tanto para a busca informada quanto para a busca não-informada.

A classe `clGeneralSearch` possui três construtores que compreendem um construtor padrão, um construtor de cópia e um construtor parametrizado. O construtor padrão apenas inicializa o objeto da classe problema sem interagir com o usuário. O construtor de cópia constrói um objeto com os mesmos atributos de outro, sem que esses objetos sejam os mesmos. O construtor parametrizado permite ao usuário instanciar um objeto da classe de busca escolhida passando como parâmetro o nome do arquivo que contém informações sobre o estado inicial do problema. Nesse caso o construtor instancia um objeto estado a partir de informações desse arquivo, inserindo-o no atributo problema utilizado na busca. Cada usuário ao definir sua classe estado implementa um construtor parametrizado que lê as informações do arquivo e instancia corretamente essas informações no objeto estado recém decladado e instanciado pelo construtor.

Existem alguns métodos públicos definidos nessa classe que são visíveis a qualquer classe derivada da classe `clGeneralSearch`. Tais métodos incluem formas de definir e recuperar o problema de um objeto pertencente a uma classe de busca instanciado. Além disso existe o método `initSearch()` que é utilizado para inicializar a busca com base nas informações presentes no atributo problema. Esse método retorna um vetor de inteiros onde cada inteiro corresponde a um operador definido na classe problema, como será visto adiante. A ordem dos valores no vetor é a ordem de aplicação dos operadores no estado inicial para atingir o estado final. Se o vetor não possui elemento nenhum, significa que a busca não apresenta solução dependendo das restrições impostas pelo método de busca utilizado - como por exemplo, limitar o aprofundamento em um dado nível da árvore de forma que nenhum dos nós possíveis de serem gerados apresente como estado corrente o estado objetivo.

Os métodos protegidos definidos nessa classe são métodos visíveis apenas para as classes derivadas da classe `clGeneralSearch`. Tais métodos possuem funcionalidades variadas, como por exemplo verificar se existe algum nó na lista de nós fechados que possui estado igual ao estado de um nó que será expandido para evitar duplicação e busca infinita, inserir um nó recém expandido na lista de nós fechados, atualizar o pai de um nó cujo estado pode ser gerado pela aplicação de um conjunto menor de operadores otimizando a solução, remover o primeiro nó da lista de nós abertos, além das funções auxiliares de busca utilizadas pelo método público `initSearch()`, seguindo os algoritmos dados na seção 3.2.

Alguns desses métodos são métodos virtuais, que possibilitam uma reimplementação mais adequada em algumas das classes derivadas da classe `clGeneralSearch` de forma a melhor representar o comportamento de um tipo particular de busca. Existe também o método `insertInOpenNodes(nodes)`, responsável por inserir nós que foram recentemente gerados por uma expansão na lista de nós abertos. Como foi visto nas seções 2.1.1 e 2.1.2 dependendo da localização em que esses nós são inseridos na lista de nós abertos diferentes estratégias de busca são implementadas. Dessa forma a decisão de implementação foi definilo como método virtual puro para ser implementado em duas classes - a classe `clBreadhtSearch` e a classe `clDepthSearch`. A partir dessas classes todas a classes que delas derivam incorporam a estratégia de busca

em profundidade ou em largura com pequenas modificações quando necessário. A Figura 4 ilustra todas as classes de busca implementadas a partir da busca em largura e da busca em profundidade.

3.3.2 A classe `clBreadthSearch`

A classe `clBreadthSearch` implementa a estratégia de busca em largura padrão. Tal classe herda a classe `clGeneralSearch` incorporando todos os métodos acima citados. Apenas um dos métodos protegidos foi implementado. Ele consiste no método virtual denominado `insertInOpenNodes(nodes)` que é utilizado para inserir os nós filhos gerados na lista de nós abertos. Como a busca em largura implica em expandir todos os nós pertencentes ao mesmo nível da árvore de busca, a inserção desses novos nós é realizada no final da lista de nós abertos garantindo que todos os nós irmãos de um nó expandido sejam expandidos antes de qualquer nó filho desse, uma vez que o método `removeFrontNode()` apenas remove o primeiro nó da lista de nós abertos como se utilizasse o protocolo `FIFO` de uma fila. Os demais métodos incluem construtores equivalentes aos construtores da classe `clGeneralSearch`. Cada um desses construtores inicializa os atributos da classe derivada se ela possuir e chama o construtor correspondente da classe pai.

3.3.3 A classe `clDepthSearch`

A classe `clDepthSearch` é a classe derivada da classe `clGeneralSearch` que implementa a busca em profundidade padrão. Cada construtor chama o construtor correspondente na classe pai. O método protegido `insertInOpenNodes(nodes)` implementa a busca em profundidade de forma que todos os nós filhos de um dado nó são expandidos antes de qualquer nó vizinho com a mesma profundidade na árvore de busca. Para realizar essa estratégia tal método insere os nós filhos gerados no início da lista de nós abertos a fim de garantir o aprofundamento na árvore de busca.

3.3.4 A classe `clDepthLimitedSearch`

A classe `clDepthLimitedSearch` é uma classe derivada da classe `clDepthSearch` que implementa a busca com profundidade limitada. Além dos atributos adquiridos da classe pai através do mecanismo de herança essa classe inclui um novo atributo utilizado para armazenar o valor da profundidade máxima permitida para a geração de novos nós filhos na árvore de busca. Dois métodos públicos permitem o acesso e modificação desse novo atributo. Como existe um limite para o aprofundamento na árvore o método protegido `initAuxiliarSearch()` herdado da classe `clGeneralSearch` foi redefinido de forma a apresentar um comportamento mais coerente com a restrição imposta no aprofundamento da árvore de busca.

3.3.5 A classe `clIterativeDeepeningSearch`

A classe `clIterativeDeepeningSearch` implementa uma estratégia de busca semelhante à busca em profundidade limitada. Ao herdar as características da classe `clDepthLimitedSearch` essa classe reimplementa alguns de seus métodos protegidos a fim de executar uma busca com aprofundamento iterativo rumo à solução do problema a resolver. Basicamente essa estratégia incrementa a profundidade máxima que a busca com profundidade limitada deve realizar a cada tentativa de obter a solução sem êxito. A busca é reiniciada e o processo continua até encontrar uma solução no *i-ésimo* nível da árvore de busca. Existe um limite simbólico para o número máximo de iterações permitidas que é dado por um atributo limite acessado por métodos públicos definidos na classe. Caso nenhum valor seja atribuído ao limite máximo esse possuirá o valor da constante `INFINITY` definido nessa classe.

3.3.6 A classe `clGreedySearch`

A classe `clGreedySearch` deriva da classe `clDepthSearch` e implementa uma estratégia de busca informada conhecida como busca "gulosa" onde após inserir os nós na lista de nós abertos esses são reordenados em ordem de *estimativa* por meio da função `BestFit()` que realiza uma chamada a um dos três algoritmos de ordenação citados na seção 3.2. Após a ordenação ser realizada o primeiro nó da lista de nós abertos é o que possui a maior probabilidade de levar a uma solução. Algumas modificações foram realizadas nessa classe como é o caso do método `initAuxiliarSearch()` que agora incorpora uma chamada ao método `BestFit()` após a inserção dos nós filhos gerados por uma expansão no nó presente no início da lista de nós abertos através do método `insertInOpenNodes(nodes)` da classe `clDepthSearch`. A avaliação da estimativa dos nós é realizada pelo método `evaluationFunction()` presente na classe estado derivada da classe `clGenericState` implementada pelo usuário.

3.3.7 A classe `clAAsteriskSearch`

A classe `clAAsterisk` consiste em uma classe derivada da classe `clGreedySearch` que implementa a estratégia de busca informada conhecida como A*. Nessa estratégia a avaliação de cada nó é dado pela soma

de sua estimativa (probabilidade de convergência para uma solução) com o custo para chegar ao estado objetivo a partir do estado do nó corrente. A diferença básica de implementação entre as classe `clGreedySearch` e `clAAsterisk` consiste no cômputo de qual é o nó que possui menor avaliação utilizado pelos algoritmos de ordenação citados.

3.3.8 A classe `clSearchNode`

A classe `clSearchNode` é utilizada para encapsular as informações de cada nó da árvore de busca. Essas informações possuem grande relevância para o processo de busca, independente de qual estratégia esteja sendo utilizada. Uma listagem de tais informações pode ser encontrada na seção 3.1.3. Essa classe apenas implementa métodos de acesso a esses atributos que funcionam como interface entre o método de busca propriamente dito e as informações geradas pela busca, como por exemplo o estado corrente do nó.

3.3.9 A classe `clGenericState`

A classe `clGenericState` define a estrutura básica de onde o usuário implementa sua classe estado de forma que a mesma possa ser utilizada pela biblioteca. Os métodos virtuais definidos nessa classe devem necessariamente ser implementados pelo usuário da biblioteca para que a busca seja ajustada para as necessidades de cada usuário. Dessa forma novas estruturas de dados e métodos podem ser implementados para melhor definir o que venha a ser o estado para o problema a ser solucionado sem que esses atributos e métodos adicionais interfiram no funcionamento da busca.

3.3.10 A classe `clGenericProblem`

A classe `clGenericProblem` apresenta os métodos que funcionarão como interface entre o problema definido e implementado pelo usuário e o que os métodos de busca necessitam para encontrar a solução para o problema especificado. Alguns métodos merecem destaque como é o caso do método `applyOperator(state, operator)` que retorna o estado que é gerado quando aplicando o operador `operator` no estado `state`. O método `getValidOperators()` retorna todos os operadores que podem ser aplicados a `state` sem gerar estados inválidos e possui extrema importância na geração dos estados pertencentes aos nós filhos que foram expandidos de um nó da árvore de busca e serão inseridos na lista de nós abertos. O método `getPathCost(parent, child)` retorna o custo para alcançar o nó `child` a partir do nó `parent`. Os demais métodos apenas acessam os atributos obrigatórios dessa classe que podem ser ampliados durante a implementação através da inserção de novos atributos e métodos à classe problema definida pelo usuário que herda as características dessa classe.

4 Exemplo de Uso da Biblioteca

Essa seção apresentará o problema do quebra-cabeça de 16 peças utilizado nessa implementação assim como comentários sobre as decisões tomadas no decorrer da implementação das classes *problema* e *estado* que representam o jogo. Esse problema foi escolhido para apresentar o uso da biblioteca pois trata-se de um problema de fácil compreensão. Os detalhes definidos para as classes que representam esse problema podem ser facilmente estendidos para qualquer outro problema a resolver.

4.1 O problema do quebra cabeça de 16 peças

O problema do quebra cabeça de 16 peças consiste em um jogo composto de um quadro de dimensões 4x4 com 15 peças numeradas de 1 a 15 que inicialmente se encontram espalhadas pelo quadro e um espaço vazio que possibilida a movimentação das demais peças fazendo-as deslizar para o local desse espaço. O jogo termina quando a configuração dada pela Figura 5 é atingida. As únicas operações permitidas para esse jogo são a movimentação das peças vizinhas ao espaço vazio.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figura 5: Configuração que representa o estado objetivo para o quebra-cabeça de 16 peças

Esse problema será utilizado com a finalidade de apresentar as etapas básicas de uso das classes genéricas demonstrando como um usuário poderá utilizá-las implementando as informações necessárias para o processo de busca. A seguir serão dados detalhes sobre a implementação das classes `clGameState` que representa um estado para o jogo do quebra-cabeça de 16 peças e da classe `cl16Puzzle` que representa o problema a ser resolvido com todos os parâmetros necessários para a realização da busca.

4.1.1 A classe `clGameState`

A classe `clGameState` consiste em uma classe que herda as características da classe abstrata `clGenericState`. Como comentado na seção 3.3.9 essa classe apresenta alguns métodos que devem ser implementados pelo usuário das classes de busca. O principal método presente nessa classe é o método `evaluationFunction()` que representa a função de avaliação heurística utilizada pelos métodos de busca informada na determinação de qual dos nós folhas deve ser expandido na próxima etapa rumo a uma solução.

Para o jogo do quebra-cabeça de 16 peças duas funções heurísticas foram propostas e implementadas. A primeira consiste em verificar a quantidade de peças que estão fora do lugar correto na configuração que representa o estado final. Como visto anteriormente, quanto menor a avaliação de um estado maiores são as chances de que expansões sucessivas nos estados que dele derivam levem a solução. Seguindo essa lógica, um estado que possui 13 peças fora do lugar tem probabilidade menor de convergir para uma solução do que um estado que possua apenas 2 peças fora do lugar. No entanto essa não é a melhor função heurística para uso nesse problema. Basta imaginar que as duas peças que estão fora do lugar no exemplo anterior são justamente as correspondentes às posições 1 e 15 na configuraçao do estado final. A quantidade de passos necessários para trocá-las de lugar pode ser infinitamente maior do que a quantidade de passos necessários para colocar as peças 13, 10 e 15 nos devidos lugares. Apesar disso segundo essa função de avaliação a primeira situação tem maior probabilidade de convergência, o que de fato não é verdade em algumas situações como demonstrado acima.

A segunda função heurística proposta corresponde mais a realidade do que a primeira função proposta. Ela consiste em calcular o somatório do número de passos necessários para colocar cada peça do estado atual na posição correspondente a do estado objetivo representado pela Figura 5. Considerando o esforço necessário para ordenar essas peças no quadro temos uma função que expressa melhor a probabilidade de convergência para uma solução.

O construtor parametrizado da classe `clGameState` é definido para ler um arquivo indicado pelo usuário e montar o estado corrente com as informações contidas nesse arquivo. No caso desse jogo a representação é feita por uma matriz de números inteiros, onde o espaço vazio é dado pela constante 0. Esse construtor abre o arquivo indicado e lê 16 valores inteiros colocando-os em posições adjascentes na matriz que representa o estado. O construtor parametrizado da classe estado deve ser implementado devido ao construtor parametrizado da classe `clGeneralSearch` instanciar um objeto estado utilizando como parâmetro o nome de um arquivo.

4.1.2 A classe `cl16Puzzle`

A classe `cl16Puzzle` é a classe responsável por encapsular as informações utilizadas pelos métodos de busca informada e não informada. Ela armazena informações vitais para a busca no espaço de estados. Essa classe consiste em uma classe derivada da classe `clGenericProblem`. Dois métodos devem ser implementados pelo usuário da biblioteca para que o processo de busca possa ocorrer.

O primeiro método consiste no método `getValidOperators()` que retorna todos os operadores válidos para o estado corrente. Para essa implementação pode-se pensar nos operadores de duas formas possíveis. A primeira seria considerar como operador cada movimento possível de cada peça do jogo. Assim pode-se ter operadores como `MOVER_2_DIREITA` ou `MOVER_15_ACIMA`. Essa abordagem é muito complexa, pois admite cerca de 60 operadores diferentes ao todo - os quatro movimentos possíveis para cada peça. Uma forma mais otimizada de resolver esse problema consiste em utilizar uma abordagem que considera os operadores do espaço vazio apenas. Dessa forma, considerando que o espaço “move-se” para as quatro direções permitidas diminui-se a representação dos operadores e a complexidade de verificar qual é válido em cada estado. O método acima comentado retorna justamente quais são os operadores válidos para cada estado. Seguindo a segunda abordagem nota-se que quando o espaço vazio estiver no canto superior direito do quadro apenas os movimentos para a direita e abaixo serão permitidos. Se essa função for implementada de forma coerente com a situação ela deverá retornar somente os operadores `MOVER_DIREITA` e `MOVER_ABAIXO`.

As classes genéricas tratam os operadores como números inteiros positivos. Cada operador deve ter um único inteiro que o represente. Nessa implementação foram utilizadas as

constantes 0, 1, 2 e 3 para representar os operadores `LEFT_MOVEMENT`, `RIGHT_MOVEMENT`, `UP_MOVEMENT` e `DOWN_MOVEMENT`, respectivamente. Quando o método `getValidOperators()` é utilizado nas classes que implementam as estratégias de busca ele retorna um vetor de inteiros com o número dos operadores válidos em cada posição do vetor.

Outro método importante implementado nessa classe pelo usuário é o método `applyOperator()`. Esse método é utilizado para gerar novos estados a partir do estado corrente aplicando um dos operadores válidos dado pela função `getValidOperators()`. Ele recebe um número inteiro que corresponde ao identificador de um dos operadores definidos para o problema e retorna o estado gerado quando aplicando esse operador. A implementação coerente desses dois métodos garantirá que a busca seja feita corretamente no espaço de estados.

Existem ainda outros métodos utilizados para diferentes propósitos, como por exemplo para definir o custo de geração de um nó. Como a maioria dos problemas de busca em espaço de estados admite que o custo de geração de um nó é unitário, então o valor padrão de custo é 1. Caso o usuário esteja implementando um problema como o do caixeiro viajante onde o custo pode ser dado pela distância entre cidades vizinhas o método correspondente deve ser reimplementado. Estruturas de dados também podem ser inseridas na classe derivada de `clGenericProblem` uma vez que sua presença não afetará o processo de busca.

5 Considerações Finais

Implementar os métodos clássicos de busca informada e não-informada nessa biblioteca utilizando a linguagem orientada a objetos C++ foi uma tarefa simples pois os próprios métodos consistem em variações de outros pré-existentes, organizando-se em uma hierarquia reproduzida pela hierarquia de classes presente na Figura 4.

Implementar problemas de busca utilizando essa biblioteca também é uma tarefa relativamente simples, uma vez que cada classe que representa o *problema* e *estado* é uma classe filha das classes `clGenericProblem` e `clGenericState`. Essas classes possuem métodos abstratos que devem ser redefinidos pelo usuário da biblioteca para melhor representar o comportamento do problema a resolver. Assim sendo, o usuário da biblioteca apenas necessita estender as classes genéricas, implementar os métodos abstratos dessas classes e compilar um programa que usa qualquer um dos métodos de busca implementados passando como parâmetros as novas classes recém definidas.

Algumas melhorias podem ser feitas na biblioteca de forma a acelerar o processo de busca e torná-la útil para o programador que deseja implementar programas que resolvam problemas de busca em um tempo satisfatório. A proposta para trabalhos futuros nessa biblioteca consiste em realizar as seguintes modificações:

- implementar um algoritmo de ordenação que possua complexidade média melhor do que os implementados nesse trabalho para diminuir o tempo gasto pela ordenação da lista de nós abertos quando utilizando algum método de busca informada. A proposta é implementar o algoritmo *mergesort* que possui complexidade $O(nlogn)$ em qualquer situação melhorando o tempo gasto para realizar cada expansão.

- aumentar a funcionalidade da biblioteca implementando outros tipos de busca informada e não-informada. A proposta é implementar os métodos de busca com custo uniforme, busca bi-direcional e busca com aprofundamento iterativo A*.

- incluir novas categorias de busca como é o caso de busca para solucionar problemas de satisfação de restrições e métodos de busca competitiva, estendendo a hierarquia de classes atual da biblioteca.

Dessas melhorias, a fundamental para um melhor desempenho da busca é implementar o algoritmo *mergesort* pois as operações de ordenação são realizadas para cada expansão de nós na árvore de busca, com tempo $O(n^2)$. Implementando essa melhoria espera-se atingir tempo $O(nlogn)$ para a ordenação, melhorando substancialmente o desempenho da busca para situações onde o número de nós gerados é muito grande.

Referências

[Russel95] Russel, S. & Norvig, P. "Artificial Inteligence - A Modern Approach", Prentice Hall, 1995.

[Nilsson98] Nilsson, N. J., "Artificial Inteligence: A New Synthesis", Morgan Kaufmann, 1998.

[Cormen99] Cormen, T. H., Leiserson, C. E., Rivest, R. L. "Introduction to algorithms", McGrall-Hill, 1999.

[Ziviani96] Ziviani, N. "Projeto de Algoritmos com implementações em Pascal e C", Pioreira, 1996.

[Schildt98] Schildt, H. "C++: The Complete Reference. Third Edition", McGrall-Hill, 1998.

[Deitel97] Deitel, H. M. & Deitel, P. J., "C++ How To Program", Prentice-Hall, 1997.