UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

BUSCA DISTRIBUÍDA UTILIZANDO COMUNICAÇÃO EM GRUPO PARA A RESOLUÇÃO DO PROBLEMA 8-PUZZLE

ANDERSON RODRIGO JANKE

BLUMENAU
2004

2004/2-05

ANDERSON RODRIGO JANKE

BUSCA DISTRIBUÍDA UTILIZANDO COMUNICAÇÃO EM GRUPO PARA A RESOLUÇÃO DO PROBLEMA 8-PUZZLE

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Paulo Fernando da Silva - Orientador

BLUMENAU
2004

2004/2-05

BUSCA DISTRIBUÍDA UTILIZANDO COMUNICAÇÃO EM GRUPO PARA A RESOLUÇÃO DO PROBLEMA 8-PUZZLE

Por

ANDERSON RODRIGO JANKE

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Paulo Fernando da Silva – Orientador, FURB
Membro:	Prof. Francisco Adell Péricas, FURB
Membro:	Prof. Jomi Fred Hübner, FURB

Blumenau, 17 de novembro de 2004

Dedico este trabalho aos poucos, mas bons amigos, próximos ou distantes, que estiveram sempre ao meu lado durante todo o tempo, comemorando comigo cada evolução mesmo sem entender nada do que eu falava.

A única coisa que interfere com meu aprendizado é a minha educação.

Albert Einstein

AGRADECIMENTOS

Ao meu orientador, Prof. Paulo Fernando da Silva, pela grande ajuda e apoio.

Ao Prof. Ademir Goulart, pelo apoio na elaboração da idéia e pela orientação inicial.

Ao Prof. José Roque Voltolini da Silva, pelo dinamismo na resolução dos problemas pertinentes à coordenação.

A todas as demais pessoas que direta ou indiretamente ajudaram neste trabalho.

RESUMO

Este trabalho tem por finalidade apresentar a especificação e implementação de um protótipo de sistema distribuído capaz de executar procedimentos de busca em estruturas de dados, utilizando um mecanismo de comunicação em grupo para a troca de mensagens entre os processos envolvidos. O protótipo foi desenvolvido incorporando técnicas para programação multiplataforma. A demonstração de seu funcionamento é apresentada pela resolução de diferentes níveis de dificuldade do problema *8-Puzzle*.

Palavras chaves: Sistemas distribuídos; Comunicação em grupo; Busca distribuída.

ABSTRACT

The purpose of this work is to present the specification and implementation of a distributed system prototype capable to execute search procedures in data structures, using a group communication mechanism for the message changes among the involved processes. The prototype was developed incorporating multiplatform programming techniques. The demonstration of its operation is presented by the resolution of different difficulty levels of the 8-Puzzle problem.

Key-Words: Distributed systems; Group communication; Distributed search.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE SIGLAS

AIAI - Artificial Intelligence Applications Institute

API - Application Program Interface

FIFO - First In First Out

GTK - GIMP Toolkit

GUI - Graphical User Interface

HTML - Hyper Text Markup Language

IP - Internet Protocol

LAN - Local Area Network

MFC - Microsoft Foundation Classes

UML - Unified Modeling Language

WAN - Wide Area Network

LISTA DE SÍMBOLOS

* - asterisco

- sustenido

SUMÁRIO

1 INTRODUÇÃO

Segundo JBOnline (2003), estimou-se pela Fundação Getúlio Vargas que o Brasil possuiria 22,4 milhões de computadores em 2003. Nas grandes empresas, havia em 1988 um computador para cada três usuários, enquanto que no final de 2002 havia um computador para cada 1,1 usuários.

Nas empresas e instituições, geralmente os computadores encontram-se conectados a uma rede interna, e com o advento da internet comercial, grande parte destes interligaram-se aos de pequenas empresas e até mesmo aos domésticos. A maioria destes computadores, mesmo estando interligados, trabalham de forma independente e passam a maior parte do tempo ociosos.

Agrupando os computadores de uma área geográfica, ou até mesmo de uma única empresa ou instituição, tem-se uma grande estrutura computacional que, tendo sua capacidade de memória e processamento compartilhada, pode oferecer um desempenho superior se comparado aos computadores mais robustos existentes hoje no mercado.

Através do conceito de sistemas distribuídos, é possível dividir as tarefas entre mais de um computador, oferecendo estrutura para aplicações que necessitam de um grande poder de processamento. Há inclusive projetos que buscam membros que possuam computadores conectados à internet e que através de um sistema distribuído fazem uso do poder de processamento não utilizado destes computadores. É o caso do projeto *Seti@home* (SETI@HOME, 2003), que busca por inteligência extraterrestre, e do *United Devices Cancer Research Project* (GRID, 2004), que busca a cura do câncer. Ambos possuem o código fonte fechado.

Um dos problemas neste tipo de contexto é a gerência da comunicação entre os processos, e é neste ponto que a tecnologia da comunicação em grupo oferece subsídios para abstrair este tipo de controle do restante da aplicação. Fazendo-se uso de um mecanismo de comunicação em grupo, o programador não terá que preocupar-se com o desenvolvimento de toda a estrutura de comunicação, bastando apenas optar pelo mecanismo que melhor atende suas necessidades e saber como integrá-lo com a aplicação.

Desta forma, propõe-se através deste trabalho o desenvolvimento de um protótipo que demonstre a resolução do problema *8-Puzzle*, clássico na área da inteligência artificial, cuja

busca pelo resultado depende de grande volume de memória e alto poder de processamento por realizar busca em volumosas estruturas de dados. Para alcançar este resultado de forma eficiente, empregar-se-á o conceito de sistemas distribuídos e a tecnologia de comunicação em grupo.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um protótipo que gerencie e realize busca em estruturas de dados em memória demonstrando através de um estudo de caso a resolução do problema *8-Puzzle*, utilizando para isso o conceito de sistemas distribuídos e a tecnologia de comunicação em grupo.

Os objetivos específicos do trabalho são:

a) demonstrar a busca pela solução de diferentes níveis de dificuldade do problema *8-Puzzle*, conhecido no campo da inteligência artificial pela sua grande necessidade de memória e poder de processamento;
b) comprovar o funcionamento do protótipo utilizando o mecanismo de comunicação em grupo *Spread* na plataforma *Windows*;
c) gerenciar de forma estável e robusta no mínimo 20 computadores interligados através de uma rede *Local Area Network* (LAN), trabalhando simultaneamente na busca de soluções para diversos níveis de dificuldade do problema *8-Puzzle*.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está dividido em quatro capítulos, sendo que o primeiro compreende a introdução ao trabalho, com a sua contextualização, seus objetivos e sua organização.

O segundo capítulo apresenta os conceitos, técnicas e ferramentas mais relevantes ao propósito do trabalho, bem como a descrição dos trabalhos correlatos.

O terceiro capítulo engloba a metodologia de desenvolvimento do trabalho, apresentando os requisitos principais do problema, a especificação, a implementação e a discussão dos resultados obtidos através de testes operacionais feitos a partir de um estudo de caso.

No quarto capítulo são apresentadas as conclusões obtidas com o desenvolvimento do trabalho, bem como suas vantagens, limitações e as sugestões de extensão para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados os principais conceitos a respeito das seguintes tecnologias: sistemas distribuídos; comunicação em grupo; mecanismo de comunicação em grupo *Spread*, técnicas de busca; o problema *8-Puzzle* e ferramentas para desenvolvimento multiplataforma.

2.1 SISTEMAS DISTRIBUÍDOS E COMUNICAÇÃO EM GRUPO

Segundo Coulouris, Dollimore e Kindberg (2001), um sistema distribuído é um conjunto de componentes de *hardware*, geralmente computadores independentes, e de *software* interligados através de uma infra-estrutura de comunicação, como uma rede de computadores ou um barramento especial, que cooperam e se coordenam entre si através da troca de mensagens, e eventualmente, pelo compartilhamento de memória comum, para execução de aplicações distribuídas pelos diferentes componentes.

Tanenbaum e Steen (2002) dizem ainda que um sistema distribuído geralmente está sempre disponível, mesmo que algumas partes estejam com problemas. Usuário e aplicação não são notificados que partes estão sendo trocadas ou consertadas, ou que novas partes foram adicionadas para servir mais usuários e aplicações.

Sendo assim, a utilização de sistemas distribuídos torna-se uma alternativa prática e geralmente econômica. É possível adicionar computadores ao longo de uma rede e integrá-los ao sistema distribuído aumentando seu poder de processamento ou desconectar computadores que necessitem de reparos, sem que o sistema seja interrompido ou que o usuário perceba estas alterações. É um sistema naturalmente redundante.

Os maiores problemas na distribuição de um sistema dizem respeito ao gerenciamento da estrutura computacional disponível e a maneira como toda a estrutura comunica-se. A estrutura de uma rede de computadores é dinâmica, novos computadores podem conectar-se a qualquer momento, bem como os computadores já presentes podem deixar de fazer parte da rede. Vários eventos precisam ser tratados, o que acaba tornando o sistema de gerenciamento e comunicação complexo.

Goulart (2002, p. 17) diz que o paradigma da comunicação em grupo foi criado com o intuito de atender a necessidade da comunicação de um para muitos com confiabilidade e

escalabilidade, abstraindo a complexidade de controle e gerenciamento das mensagens, da complexidade da aplicação, tratando os diversos computadores como sendo membros de um grupo. Assim, o projetista preocupa-se apenas com as funcionalidades, os procedimentos e os algoritmos particulares da aplicação, sem envolver-se com os problemas referentes à comunicação. Passa a ser de responsabilidade do mecanismo de comunicação em grupo o controle de fluxo, a confiabilidade, o seqüênciamento e a garantia de entrega das mensagens ao aplicativo final, bem como o controle de quais computadores estão fazendo parte deste grupo, gerenciando a entrada de novos membros no grupo e a saída voluntária ou involuntária devido à quebra de um computador ou particionamento da rede.

Um outro aspecto importante do mecanismo de comunicação em grupo observa-se quando uma aplicação distribuída executando em trinta computadores diferentes, cada qual com vinte processos distintos que necessitam comunicar-se com todos os demais processos, onde seriam necessárias centenas de conexões para cada um dos processos. Utilizando-se o paradigma da comunicação em grupo, este tipo de comunicação é possível com apenas uma conexão por processo (SPREAD, 2004).

2.2 MECANISMO DE COMUNICAÇÃO EM GRUPO SPREAD

Goulart e Friedrich (2002, p. 4) enumeram os seguintes mecanismos de comunicação em grupo existentes: *Isis*; *Phoenix*; *RMP*; *Totem*; *Ensemble*; *Transis*; *Newtop*; *Atomic*; *Intergroup*; *JGroup* e *Spread*. Dentre estes mecanismos, alguns evoluíram, mas em seguida permaneceram sem atualizações, enquanto outros, como *Intergroup* e *JGroup*, são recentes e baseados no *Java/RMI*. O *Spread* evoluiu e continua em desenvolvimento, e além disso possui boa documentação, código fonte aberto e já foi portado para diversas plataformas além do *Unix*, como *Windows* e *MacOS*.

Segundo Goulart (2002, p. 101), o mecanismo de comunicação em grupo *Spread* trata as dificuldades encontradas em redes *Wide Area Network* (WAN) através de três principais características:

a) uso de diferentes protocolos de baixo nível que provêm o envio confiável de mensagens de acordo com a configuração da rede;
b) uso da arquitetura cliente-servidor, que dentre muitos benefícios, o mais importante é a habilidade de pagar um preço mínimo para as diferentes causas de alterações de

membros do grupo, como o processo de entrada ou saída de um grupo que se resume a uma simples mensagem;

c) separar o mecanismo de disseminação e segurança local do protocolo de ordenação e estabilidade, permitindo o envio imediato de mensagens à rede, independente de outros controles de perda e ordenação.

Amir e Stanton (1998, p. 3) explicam ainda que o *Spread* baseia-se no modelo cliente-servidor, onde um ou mais servidores, ou *daemons*, são responsáveis pela formação da rede de disseminação de mensagens e pelo fornecimento dos serviços de ordenação e gerência de membros dos grupos; e os clientes por sua vez são formados por aplicações que possuem uma pequena biblioteca introduzida em seu código que permite a conexão ao servidor *Spread*. Estes clientes podem estar em qualquer ponto da rede e irão conectar-se ao servidor mais próximo para fazer uso dos serviços.

Complementam ainda, Amir e Stanton (1998, p. 4), que o mecanismo de comunicação em grupo *Spread* é configurável, permitindo que seja utilizado um único servidor para uma rede inteira, ou um servidor para cada computador que esteja executando a aplicação de comunicação em grupo. A utilização de um servidor em cada computador contribui para a melhora do desempenho, entretanto o uso de poucos servidores, devidamente posicionados, diminui o custo da recuperação em caso de falha.

Outra característica do mecanismo de comunicação em grupo *Spread* é possuir uma estrutura tolerante a falhas que garante a continuidade da troca de mensagens mesmo nas situações em que vários nodos da rede deixam de funcionar. Sua arquitetura possibilita também que novos computadores sejam adicionados ao sistema distribuído sem que seja necessário alterar o código da aplicação (ZAWODNY, 2003).

Giordano e Jansch-Pôrto (2000) descrevem esta característica como implementada na camada de roteamento do mecanismo *Spread*, fazendo com que o sistema permaneça estável, provendo confiabilidade, robustez e consistência. O mecanismo *Spread* compreende uma camada de rede composta pelo protocolo *Ring*, para domínios *multicast* e redes do tipo LAN; e o protocolo *Hop*, para conexões ponto-a-ponto e redes do tipo WAN. Ambos os protocolos são integrados em um sistema baseado em árvores de roteamento que garantem um melhor desempenho em uma rede de comunicação confiável. Amir e Stanton (1998, p. 8) e Goulart (2002, p. 109) explicam detalhadamente o funcionamento dos protocolos *Ring* e *Hop*.

Como citado anteriormente, as aplicações que constituem os clientes possuem uma biblioteca integrada ao seu código, que por sua vez fornece uma *Application Program Interface* (API) para a comunicação com o servidor *Spread*. Stanton (2002, p. 23) descreve detalhadamente esta API, que resume-se nas seguintes funções:

a) sp_connect: estabelece a comunicação com o servidor *Spread*;
b) sp_disconnet: finaliza a conexão com o servidor *Spread*;
c) sp_join: entra em um determinado grupo;
d) sp_leave: sai de um determinado grupo;
e) sp_multicast: envia uma mensagem aos membros de um grupo;
f) sp_receive: recebe mensagens enviadas por outros membros do grupo ou pelo servidor *Spread*. Um parâmetro da função informa se a mensagem é constituída por dados ou se contém informações de alteração nos grupos, como por exemplo, a entrada ou saída de um usuário.

Stanton (2002, p. 4) descreve ainda que o mecanismo de comunicação em grupo *Spread* oferece diferentes tipos de serviços de mensagens à aplicação, formados por padrões de ordenação e confiança, conforme apresentado na Tabela 1.

Tabela 1 - Tipos de serviços de mensagens

Tipo de serviço	Ordenação	Confiança
UNRELIABLE_MESS	Nenhuma	Sem confiança
RELIABLE_MESS	Nenhuma	Com confiança
FIFO_MESS	Fifo	Com confiança
CAUSAL_MESS	Causal	Com confiança
AGREED_MESS	Ordenação total	Com confiança
SAFE_MESS	Ordenação total	Seguro

Fonte: Stanton (2002, p. 5)

Os padrões de ordenação oferecidos pelo *Spread* são descritos como:

a) nenhum: não há garantia de ordenação, ou seja, uma mensagem enviada com este padrão de ordenação pode ser entregue tanto antes quanto depois de uma segunda mensagem enviada utilizando este mesmo padrão;
b) fifo: as mensagens serão entregues de acordo com a ordenação *First In First Out* (FIFO) estabelecida pelo remetente;
c) causal: todas as mensagens enviadas por todos os remetentes são consistentes com a definição de ordem causal de *Lamport* (MACÊDO, 1995);
d) ordenação total: todas as mensagens enviadas por todos os remetentes são entregues exatamente na mesma ordem para todos os destinatários.

Da mesma forma, os padrões de confiança são descritos como:

a) sem confiança: a mensagem pode ser descartada ou perdida e não será recuperada pelo mecanismo *Spread*;
b) com confiança: caso a mensagem seja descartada ou perdida, o mecanismo *Spread* irá recuperá-la;
c) seguro: a mensagem somente será entregue quando houver certeza absoluta de que todos os destinatários irão recebê-la, mesmo que ocorra uma mudança nos membros.

Quanto ao processo de configuração do servidor *Spread*, deve-se determinar o endereço *broadcast* da rede onde o servidor irá atuar, bem como a porta em que irá aguardar por conexões. Deve-se também configurar um nome para o servidor e o endereço *Internet Protocol* (IP) de cada computador que estiver executando uma instância do servidor *Spread*. O Quadro 1 ilustra a parte do arquivo de configuração, *spread.conf*, onde são determinadas estas opções para um contexto onde exista apenas um servidor.

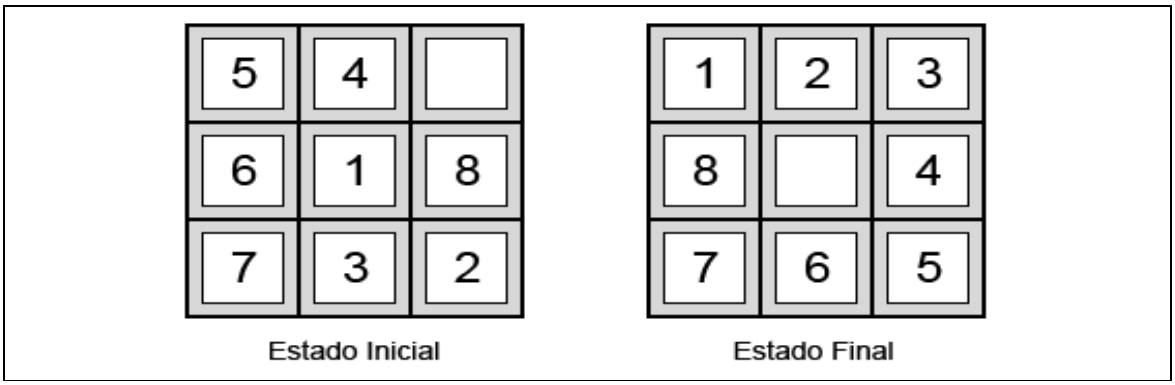
```
Spread_Segment  192.168.0.255:4803 {
      localhost         192.168.200.1
}
```

Quadro 1 - Arquivo de configuração do servidor *Spread*

No exemplo do Quadro 1, vê-se a configuração do endereço *broadcast 192.168.0.255* com a porta *4803*. A identificação do servidor é *localhost* e o endereço IP do computador onde está sendo executado o servidor *Spread* é *192.168.200.1*.

2.3 O PROBLEMA 8-PUZZLE

O problema *8-Puzzle*, também conhecido como jogo das oito fichas, é clássico no campo da inteligência artificial. Consiste, segundo Russel e Norvig (1995, p. 63), de um tabuleiro quadrado com nove divisões, onde são colocadas oito fichas quadradas, numeradas de 1 à 8, e um espaço vazio. A ficha adjacente ao espaço vazio pode ser deslocada para aquele espaço. O objetivo é obter a partir de uma posição inicial, uma posição meta, conforme a Figura 1, deslocando as peças até que atinjam a posição desejada.



Fonte: Russel e Norvig (1995, p. 63)

Figura 1 - Uma instância típica do problema *8-Puzzle*

A principal dificuldade em solucionar o problema *8-Puzzle* através de algoritmos computacionais é a grande necessidade de memória e de poder de processamento. Cada nodo da estrutura de dados irá gerar novos nodos de forma sucessiva, e a velocidade de criação destes novos nodos depende do poder de processamento do computador em uso. Conforme a configuração do estado inicial e do estado final, a busca pelo caminho entre estes estados pode gerar uma estrutura de nodos muito grande, o que torna necessário a disponibilidade de um grande volume de memória.

2.4 TÉCNICAS DE BUSCA

Problemas de busca compõem um ramo conhecido, principalmente no âmbito da inteligência artificial, e existem vários algoritmos que os resolvem das mais diversas formas. Particularmente, um dos problemas que mais se destaca é o da busca por caminhos, e consequentemente o menor caminho, entre dois nodos de um grafo.

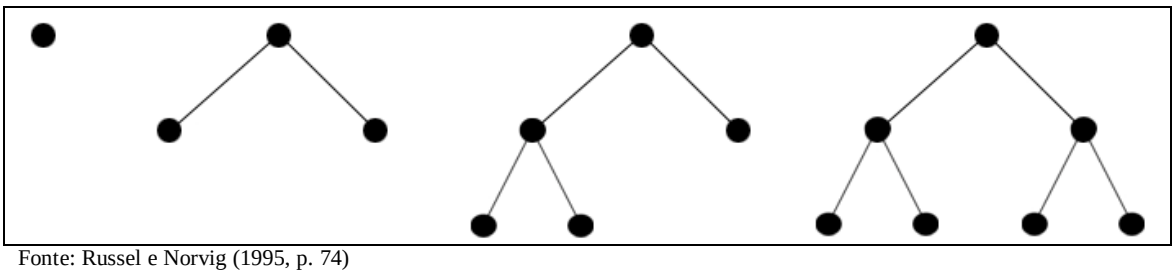
Dentre os algoritmos utilizados na tentativa de solucionar este tipo de problema, destacam-se os algoritmos de busca em largura, busca em profundidade e busca heurística. Cada um destes algoritmos possui as suas particularidades e sendo assim, deve-se concentrar parte do trabalho na escolha da estratégia de busca correta para o problema. Para facilitar este processo de escolha, segundo Russel e Norvig (1995, p. 73), os algoritmos podem ser classificados por meio de quatro características básicas, que são:

a) completo: um algoritmo de busca é completo quando é capaz de encontrar a solução para o problema, caso exista;

b) ótimo: um algoritmo de busca é considerado ótimo quando a solução encontrada é a melhor possível. Por exemplo, a solução possui o menor caminho ou o caminho de menor custo entre dois nodos;
c) tempo: caracteriza-se pelo tempo que o algoritmo pode levar para encontrar a possível solução;
d) espaço: indica a quantidade de memória necessária para que um algoritmo possa encontrar a solução em determinada situação.

2.4.1 BUSCA EM LARGURA

Segundo Russel e Norvig (1995, p. 74), o algoritmo de busca em largura é considerado uma das estratégias de busca mais simples. Nesta estratégia, o nodo raiz é expandido primeiro e em seguida são expandidos todos os seus sucessores, que por sua vez também são expandidos. Basicamente, a busca em largura consiste na expansão de todos os nodos de profundidade *d*, e em seguida a expansão de todos os nodos de profundidade *d+1*, e assim sucessivamente, como mostra a Figura 2.



Fonte: Russel e Norvig (1995, p. 74)

Figura 2 - Ilustração da busca em largura

O algoritmo de busca em largura é completo e ótimo, ou seja, encontrará sempre a melhor resposta, mesmo havendo mais de uma solução, pois devido a sua característica de expandir primeiro todos os nodos de uma determinada profundidade, garante que o primeiro nodo encontrado, que corresponda com o nodo meta, será o nodo que possui o menor caminho em relação ao nodo raiz.

Porém, esta característica que garante que a primeira solução encontrada será a melhor, é a mesma que compromete o algoritmo em relação ao tempo e espaço para grandes estruturas de dados. Para avaliar melhor o nível de comprometimento, é possível calcular a quantidade de nodos que serão expandidos até uma determinada profundidade, e para isso deve-se saber o

número de ramificações dos nodos, ou *branching factor*, que caracteriza-se pela quantidade de novos nodos que serão gerados a partir da expansão do nodo anterior.

Considerando que o número de ramificações de um nodo é igual a b, tem-se que sua expansão irá gerar b^2 novos nodos, que por sua vez irão expandir b^3 novos nodos, e assim sucessivamente até a profundidade d, onde ter-se-á b^d. Logo, o total de nodos expandidos até a profundidade d pode ser representado pela fórmula do Quadro 2.

$$1 + b + b^2 + b^3 + \bullet\bullet\bullet + b^d$$

Fonte: Russel e Norvig (1995, p. 74)

Quadro 2 - Fórmula de cálculo do total de nodos

Para exemplificar melhor o comprometimento do algoritmo de busca em largura em relação ao tempo e espaço, a Tabela 2 ilustra uma simulação onde cada nodo ocupa 100 bytes, cada expansão gera 10 novos nodos e 1000 nodos são processados por segundo.

Tabela 2 - Relação de tempo e espaço do algoritmo de busca em largura

Profundidade	Nodos	Tempo	Espaço
0	1	1 milisegundo	100 bytes
2	111	100 milisegundos	11 kilobytes
4	11.111	11 segundos	1 megabyte
6	10^6	18 minutos	111 megabytes
8	10^8	31 horas	11 gigabytes
10	10^{10}	128 dias	1 terabyte
12	10^{12}	35 anos	111 terabytes
14	10^{14}	3500 anos	11.111 terabytes

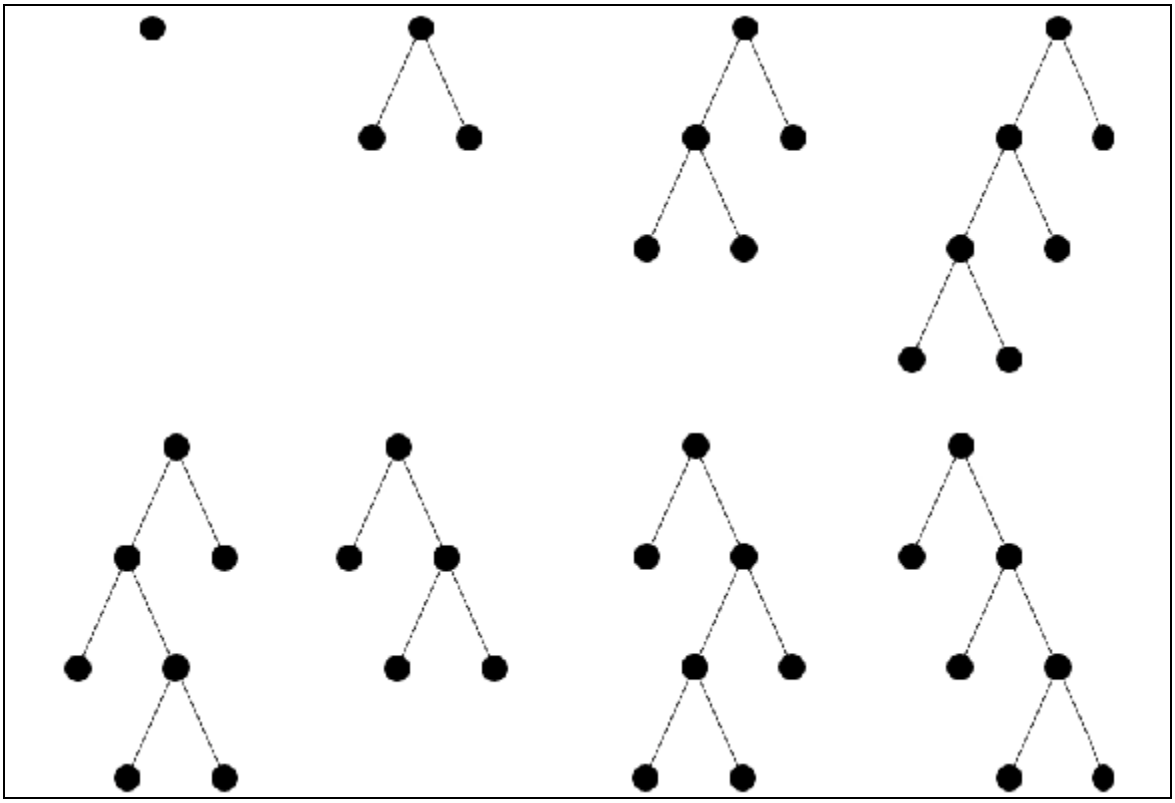
Fonte: Russel e Norvig (1995, p. 75)

Considerando os valores apresentados na Tabela 2, vê-se que a utilização do algoritmo de busca em largura é útil para buscas em estruturas de dados pequenas ou de pouca profundidade. Porém, requer atenção em estruturas maiores ou de maior profundidade, pois pode gerar intenso processamento e inclusive tornar-se impraticável.

Entretanto é possível adaptar ao algoritmo de busca em largura um mecanismo de poda, que armazena os nodos anteriormente processados e impede que novos nodos similares àqueles sejam novamente gerados. Desta forma, tem-se uma árvore menor, apenas com nodos únicos, mas com o atraso provocado pela constante verificação entre os novos nodos e os nodos já visitados.

2.4.2 BUSCA EM PROFUNDIDADE

O algoritmo de busca em profundidade baseia-se no principio de expandir um único nodo com a maior profundidade possível. Geralmente, o critério de escolha do nodo a ser expandido baseia-se na utilização do nodo mais a esquerda ou mais a direita dentre os nodos gerados. Quando o algoritmo atinge um nodo que não é a meta e que não pode mais ser expandido, o algoritmo retorna ao nível anterior e expande os nodos subseqüentes, como ilustra a Figura 3.



Fonte: Russel e Norvig (1995, p. 77)

Figura 3 - Busca em profundidade em uma árvore binária

Russel e Norvig (1995, p. 77) explicam que o algoritmo de busca em profundidade necessita de pouca memória, pois precisa armazenar um único caminho entre a raiz e o nodo atual da árvore de dados. Em uma situação em que o número de ramificações de um nodo é igual a b e a profundidade máxima é m, o algoritmo irá gerar bm nodos.

Comparado ao algoritmo de busca em largura na simulação apresentada na Tabela 2, em uma profundidade igual a 12, o algoritmo de busca em profundidade precisará de 12

kilobytes ao invés de 111 *terabytes* do anterior, ou seja, uma quantidade de memória dez bilhões de vezes menor.

Porém, quando a árvore de dados é muito grande ou até mesmo infinita, o algoritmo de busca em profundidade irá atingir uma profundidade sempre maior e não retornará aos níveis anteriores, ficando em um *loop* infinito e sem retornar nenhuma solução, ou retornando uma solução que dificilmente será ótima. Por esta razão, o algoritmo de busca em profundidade não é completo e não é ótimo.

Uma maneira de evitar a possibilidade de que o algoritmo de busca em profundidade entre em um *loop* infinito, é limitar a profundidade máxima que ele pode atingir. Porém, limitar a profundidade de busca pode fazer com que o algoritmo percorra toda a árvore sem encontrar solução, sendo que a mesma existe, mas está em níveis mais profundos do que o limite estabelecido.

Outra maneira de melhorar o desempenho do algoritmo de busca em profundidade é a utilização de um mecanismo de poda, similar ao do algoritmo de busca em largura. Com a utilização deste mecanismo, garante-se que quando houver um nodo cujo todos os descendentes que serão gerados já tenham sido visitados, o algoritmo irá retornar ao nível anterior e processar os nodos subseqüentes, impedindo que o algoritmo entre em um estado de *loop* infinito.

2.4.3 BUSCA HEURÍSTICA

A utilização dos algoritmos de busca em largura e profundidade pode ser viável em pequenas estruturas de dados e com pequenos níveis de profundidade, mas em problemas mais complexos, envolvendo grandes estruturas de dados, sua utilização passa a apresentar riscos, mesmo fazendo-se uso de um mecanismo de poda eficiente.

Há também certos problemas e situações com restrições de tempo e espaço que impedem que sejam utilizados algoritmos de busca que explorem todos os nodos, e neste caso, é preciso encontrar uma estratégia eficiente que encontre a solução sem que seja necessário percorrer toda a árvore de busca.

Por estas razões, passou-se a utilizar algoritmos mais específicos para cada tipo de problema e que possuem um método heurístico que auxilia na escolha dos melhores nodos

para serem visitados. Geralmente, um método heurístico consiste na atribuição de valores aos nodos, como por exemplo, o custo de utilização do caminho sugerido por aquele nodo ou um valor gerado a partir de uma estimativa de quão próximo aquele nodo está do nodo meta.

Sendo assim, Santos (2002, p. 22) explica que o algoritmo A* (pronuncia-se "A-Estrela") apresenta-se como uma solução mais apropriada aos problemas de busca de menor caminho em curto espaço de tempo, pois encontra o caminho de menor custo entre dois nodos examinando apenas os vizinhos mais promissores do nodo atual da busca.

O algoritmo A* mantém duas listas de nodos: abertos e fechados. Para nodos a explorar e já explorados, respectivamente. No início, fechados está vazia e abertos contém apenas o nodo de origem. A cada iteração, o melhor nodo é removido da lista de abertos para ser explorado. Para cada vizinho do nodo, são feitas as seguintes operações: se o vizinho está em fechados, é ignorado; se está em abertos, suas informações são atualizadas se o custo do caminho atual é menor do que o custo calculado anteriormente para ele; se não está em nenhuma das duas listas, é um novo nodo que é então colocado na lista de abertos. Depois de serem processados todos os seus vizinhos, o nodo removido da lista de abertos é colocado na lista de fechados.

Como descrito anteriormente, o critério para escolher o melhor nodo a ser processado é uma estimativa da distância entre cada nodo e o nodo de destino, calculado através da fórmula apresentada no Quadro 3, onde: $f(v)$ representa o custo total do caminho do nodo de origem ao nodo de destino passando pelo vértice v; $g(v)$ representa o custo calculado do caminho do nodo de origem ao nodo v, e $h(v)$ representa o custo estimado do caminho do nodo v ao nodo de destino. A cada iteração, o nodo da lista de abertos com o menor custo estimado será escolhido pelo algoritmo.

$f(v) = g(v) + h(v)$

Fonte: Santos (2002, p. 23)

Quadro 3 - Fórmula da estimativa heurística do algoritmo A*

Russel e Norvig (1995, p. 99) garantem que o algoritmo A* é ótimo e que não há outro algoritmo que encontre a solução ótima de um mesmo problema expandindo uma menor quantidade de nodos, pois qualquer tentativa de não expandir todos os nodos vizinhos ao melhor nodo gera o risco de perder a solução ótima.

Hübner (2004) diz que a complexidade de tempo e espaço do algoritmo de busca heurística A* é, no pior dos casos, similar ao da busca em largura.

2.5 BIBLIOTECA WXWIDGETS

Segundo Schneebeli (2000?), o desenvolvimento de aplicações para ambientes gráficos, como o *Microsoft Windows*, é uma tarefa complexa e que, não sendo bem elaborada, pode levar a um software com problemas de qualidade. Em geral, uma API consiste de um conjunto de rotinas, funções e procedimentos, e de um conjunto de definições de estrutura de dados, necessárias para o uso destas rotinas. Isto leva a uma programação procedural complexa e repetitiva.

O uso da programação orientada a objetos simplifica consideravelmente a tarefa de programação, através do encapsulamento e do reaproveitamento de código, principalmente se uma biblioteca de classes já testada for utilizada. Para o desenvolvimento de aplicações para a plataforma *Windows*, a biblioteca de classes padrão é a *Microsoft Foundation Classes* (MFC). No entanto, a MFC permite somente o desenvolvimento para este ambiente e qualquer tentativa de conversão para uso em outras plataformas implica em grandes modificações do código.

Visando permitir portabilidade, outras bibliotecas foram criadas para o uso neste tipo de aplicação. GUIToolkit (2003) fornece uma relação das principais bibliotecas existentes atualmente. Algumas delas possuem a característica de serem multiplataforma, isto é, o mesmo código-fonte pode ser compilado em outro ambiente gerando um código executável nativo. Dentre estas bibliotecas, destacam-se:

a) fox: desenvolvido por Jeroen van der Zijp, é estável e oferece vários recursos, embora a documentação seja ruim, especialmente para iniciantes. O software gerado apresenta boa aparência, parecida com os programas da plataforma *Windows*;
b) qt: desenvolvida pela Trolltech, é livre para uso não comercial. Apresenta boa documentação e boa aparência. Tem se revelado quase que um padrão para desenvolvimento na plataforma *Linux*. Porém, uma desvantagem é que o código a ser compilado é obtido através do processamento do código-fonte por um software especial, que substitui certas construções por comandos *C++*. Isto é, não se programa diretamente em *C++*, mas sim em uma extensão da linguagem, o que

pode confundir os iniciantes bem como os ambientes de programação. Uma de suas vantagens é a interface gráfica para geração de aplicativos, o *QtDesigner*;

c) wxWidgets: criada por Julian Smart, atual líder do projeto, a partir de 1992 no *Artificial Intelligence Applications Institute* (AIAI), na Universidade de *Edinburgh*, e posteriormente liberado pela instituição para livre desenvolvimento e distribuição pelo autor. Os programas gerados com o uso desta biblioteca tem aspecto nativo da plataforma para a qual foi compilado. Existem extensões para uso de *OpenGL* e também de outras linguagens como *Java* e *Python*. É bem documentada e sua única desvantagem é a falta de uma interface gráfica livre para geração de aplicativos. A *wxWidgets*, anteriormente chamada de *wxWindows*, teve seu nome alterado recentemente por solicitação formal da *Microsoft Corporation*.

Segundo Furtado, Heuseler e Lychowski (2003), a biblioteca *wxWidgets* constitui um conjunto de códigos que permitem que aplicações *C++* sejam compiladas e executadas em diferentes tipos de plataforma, conforme ilustrado pela Tabela 3, sem necessidade de mudança no código-fonte, desde que programada corretamente. Possui uma biblioteca *Graphical User Interface* (GUI) contendo os principais elementos necessários para a construção visual da aplicação, como: janelas, menus, diálogos e botões; assim como também provê uma API para funcionalidades dos sistemas operacionais comumente usadas, como desde copiar e excluir arquivos até o suporte a *sockets* e *threads*.

Tabela 3 - Portabilidade das aplicações com *wxWidgets*

Mesmo código fonte para todas as plataformas								
API wxWidgets								
wxMSW	wxX11	wxGTK	wxMotif	wxMax	wxBase			
GDI	Xlib/X11	GTK+	Motif	Mac	*Sem interface*			
Windows	Unix			MacOS	Windows	Unix	MacOS	OS/2

Fonte: Furtado, Heuseler e Lychowski (2003)

Possui uma API simples e de fácil uso, bastando que a biblioteca específica para a plataforma desejada seja *linkada* com a aplicação e um compilador compatível seja utilizado (quase todos os compiladores populares para *C++*). Depois de compilada, a aplicação terá a aparência nativa da plataforma. Oferece também recursos de ajuda *online*, *streams*, área de transferência, arrastar e soltar, manipulação de imagens, suporte a banco de dados, pré-

visualização e impressão de código *Hyper Text Markup Language* (HTML), dentre outros recursos. Estruturas básicas, tais como *strings*, *arrays*, listas e tabelas *hash* também são suportadas.

2.6 TRABALHOS CORRELATOS

Como trabalho correlato pode-se citar um aplicativo de troca de mensagens desenvolvido em *Java* (BERKET, 2000, apud GOULART, 2002, p. 86), que utiliza comunicação em grupo com o mecanismo *Intergroup*.

É um aplicativo simples, cuja finalidade é permitir a troca de mensagens de texto entre usuários. Tanto o servidor quanto o cliente foram implementados em *Java*, e por esta razão apresenta-se certa lentidão na execução.

Pode-se citar também um aplicativo de coleta de dados e gerência de computadores conectados a uma rede (GOULART, 2002, p. 134), desenvolvido utilizando a ferramenta *Microsoft Visual C++*.

O aplicativo foi construído com o objetivo de avaliar o mecanismo de comunicação *Spread*, no que diz respeito à sua conectividade, estabilidade, escalabilidade e desempenho em um ambiente de rede WAN.

Outro trabalho correlato é um aplicativo desenvolvido por Theo Schlossnagle e George Schlossnagle, que tem por finalidade unificar os arquivos de *log* gerados por um *cluster* de servidores *web Apache* que faz uso do mecanismo de comunicação em grupo *Spread* (MODLOG, 2000).

Há também o *AOL Communicator* (AOL, 2003), que caracteriza-se por um aplicativo de troca de mensagens instantâneas, gerência de caixa postal, filtros *anti-spam* e sintonizador de rádios *on-line*.

O *Audacity* (AUDACITY, 2004?) é um editor de áudio livre, com funções de gravação, edição e reprodução arquivos do tipo WAV, AIFF, MP3 e OGG. Possui também efeitos de amplificação e distorção do áudio.

Já o *AVG Antivírus* (GRISOFT, 2004) é um aplicativo antivírus com funções de atualização automática da lista de vacinas contra vírus, remoção de arquivos infectados para

quarentena e verificação de arquivos anexados a *e-mails*. O *AVG Antivírus* está disponível na versão paga para uso comercial, e gratuita para uso doméstico.

Tanto o *AOL Communicator*, quanto o *Audacity* e *AVG Antivírus* utilizam em seu desenvolvimento a biblioteca *wxWidgets*, para permitir portabilidade.

3 DESENVOLVIMENTO DO TRABALHO

Neste capítulo serão apresentados os requisitos do trabalho, bem como a sua especificação, as técnicas e ferramentas utilizadas para a implementação, o teste de operacionalidade e a discussão dos resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O protótipo proposto neste trabalho tem como requisitos funcionais (RF):

a) controle: o protótipo deverá possuir um aplicativo cliente que permitirá ao usuário controlar as configurações desejadas para o problema *8-Puzzle*, como por exemplo o nível de dificuldade;
b) tarefas: o aplicativo cliente deverá controlar a divisão da tarefa de busca da solução do problema entre os servidores disponíveis;
c) servidor: o protótipo deverá possuir um aplicativo servidor que receberá as configurações do aplicativo cliente e efetuará a busca pela solução do problema em uma determinada área da estrutura de dados definida pelo cliente;
d) estatística: o aplicativo servidor deverá informar ao cliente estatísticas de seu processo de busca quando em andamento;
e) resposta: o programa servidor deverá informar ao cliente a resposta do problema, caso encontre-a, especificando os passos necessários para alcançar o nodo meta a partir do nodo inicial.

Os requisitos não funcionais (RNF) resumem-se a:

a) arquitetura: o protótipo deverá possuir a arquitetura cliente/servidor;
b) comunicação: o protótipo deverá utilizar o mecanismo de comunicação em grupo *Spread* e funcionar em uma rede LAN;
c) plataforma: o protótipo deverá ser compatível com o sistema operacional *Microsoft Windows 2000 Professional* ou superior;
d) desempenho: o protótipo deverá oferecer estabilidade e velocidade de execução aceitáveis.

3.2 ESPECIFICAÇÃO

O método de ciclo de vida escolhido para o desenvolvimento do protótipo apresentado neste trabalho é o iterativo e incremental, por tratar-se de um método onde todo o trabalho é

dividido em partes menores que tem como resultado um incremento ao trabalho final. Este método oferece maior segurança no atendimento dos requisitos e maior flexibilidade durante todo o processo de desenvolvimento.

A especificação do problema apresenta-se através de um grupo de diagramas da linguagem *Unified Modeling Language* (UML), sendo estes: diagrama de classes, diagrama de seqüência e diagrama de casos de uso. Utilizou-se a ferramenta *Rational Rose* versão 2003.06 para a modelagem da especificação.

Considerando o fato de que o problema trata de dois aplicativos distintos, cliente e servidor, apresentam-se o conjunto de diagramas para cada aplicativo separadamente.

3.2.1 APLICATIVO CLIENTE

O diagrama de casos de uso do aplicativo cliente, mostrado na Figura 4, apresenta os casos de uso pertinentes ao usuário do aplicativo, que são: criar busca, executar busca, cancelar busca, visualizar estatística e visualizar solução.

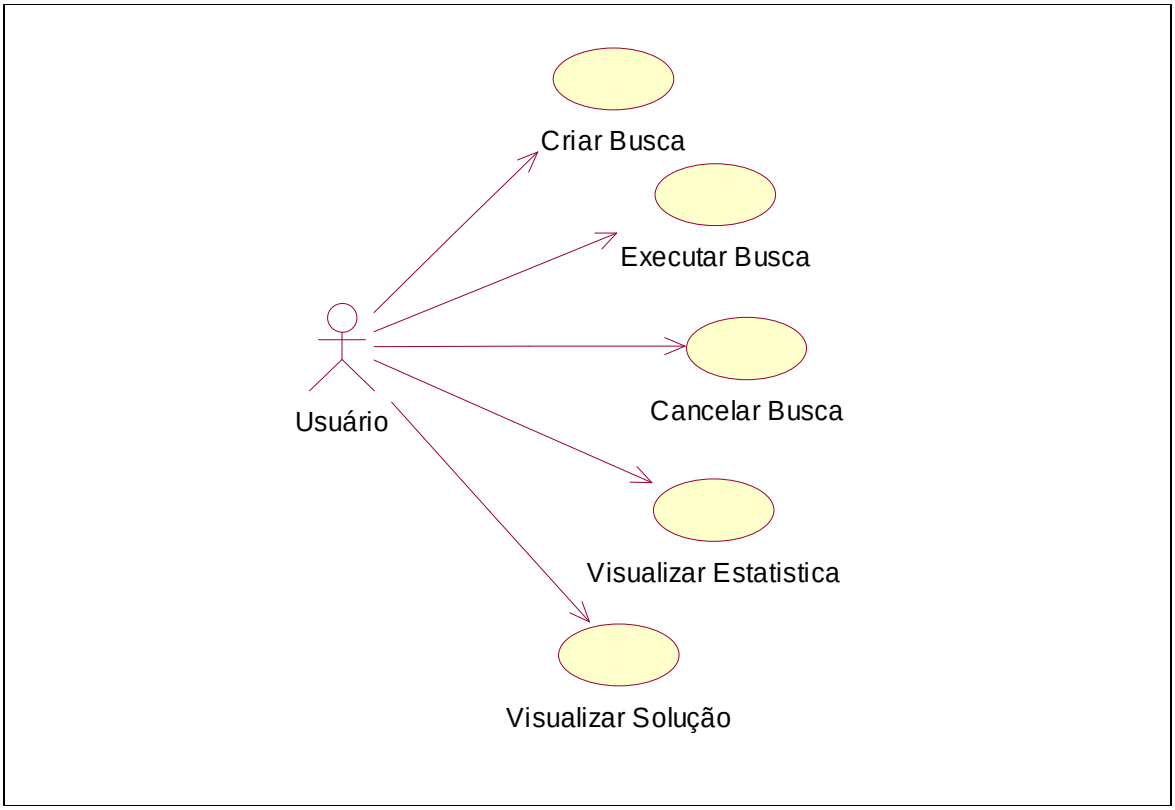


Figura 4 - Diagrama de casos de uso do aplicativo cliente

O caso de uso criar busca refere-se à funcionalidade que permite ao usuário configurar um novo processo de busca, definindo os estados do problema, quais servidores farão parte do processo de busca e demais opções, como definir qual algoritmo de busca deverá ser utilizado. Enquanto que o caso de uso executar busca refere-se ao inicio propriamente dito do processo, quando os servidores começam a buscar por uma solução para o problema. O caso de uso cancelar busca diz respeito à possibilidade do usuário cancelar o processo, seja após a criação de uma nova busca ou até mesmo após o início da execução da mesma.

Já o caso de uso visualizar estatística representa a funcionalidade do aplicativo que permite ao usuário ter conhecimento sobre o estado atual do processo, como por exemplo: busca em andamento, busca concluída ou solução encontrada. Similar a este, há o caso de uso visualizar solução, que apresenta a possibilidade do usuário consultar a solução de uma busca através dos passos necessários para se chegar ao estado final do problema, a partir do estado inicial.

A Figura 5 apresenta o diagrama de classes do aplicativo cliente, sendo estas as seguintes: *FramePrincipal*, *FrameBusca*, *ThreadControle*, *ThreadEstatistica*, *Servidor* e *Solucao*. A classe *FramePrincipal* é responsável pela aparência gráfica da aplicação, ou seja, o controle da janela principal, menu e demais componentes visuais necessários. Enquanto que a classe *FrameBusca* controla as características visuais referentes aos controles que tornam-se visíveis no momento em que uma nova busca é criada.

A classe *FrameBusca* trata também das demais funcionalidades do processo de busca, como: gerenciar o seu início e término, tratamento das mensagens enviadas por servidores do mesmo grupo, manipulação da solução caso algum servidor encontre-a. O relacionamento entres as classes *FramePrincipal* e *FrameBusca* é de um para zero ou muitos, respectivamente, pois cada novo processo de busca irá criar uma nova instância da classe *FrameBusca*, que tem como janela principal uma única instância da classe *FramePrincipal*.

Já a classe *ThreadControle* é responsável por todos os processos que envolvem o servidor *Spread*, como: conexão, troca de mensagens e gerência da entrada e saída de servidores do grupo principal. Uma instância da classe *ThreadControle* é na verdade uma *thread*, pois é necessário que a aplicação permaneça em *loop* verificando constantemente a existência de mensagens através da função *SP_receive* da API do mecanismo de comunicação em grupo *Spread*. A *thread* permanece ininterruptamente em execução, concorrentemente aos

demais processos, e somente é finalizada quando a aplicação é encerrada ou quando ocorre um erro na comunicação com o servidor *Spread*. A classe *ThreadControle* relaciona-se com a classe *FramePrincipal*, onde acessa recursos visuais, e com a classe *Servidor*.

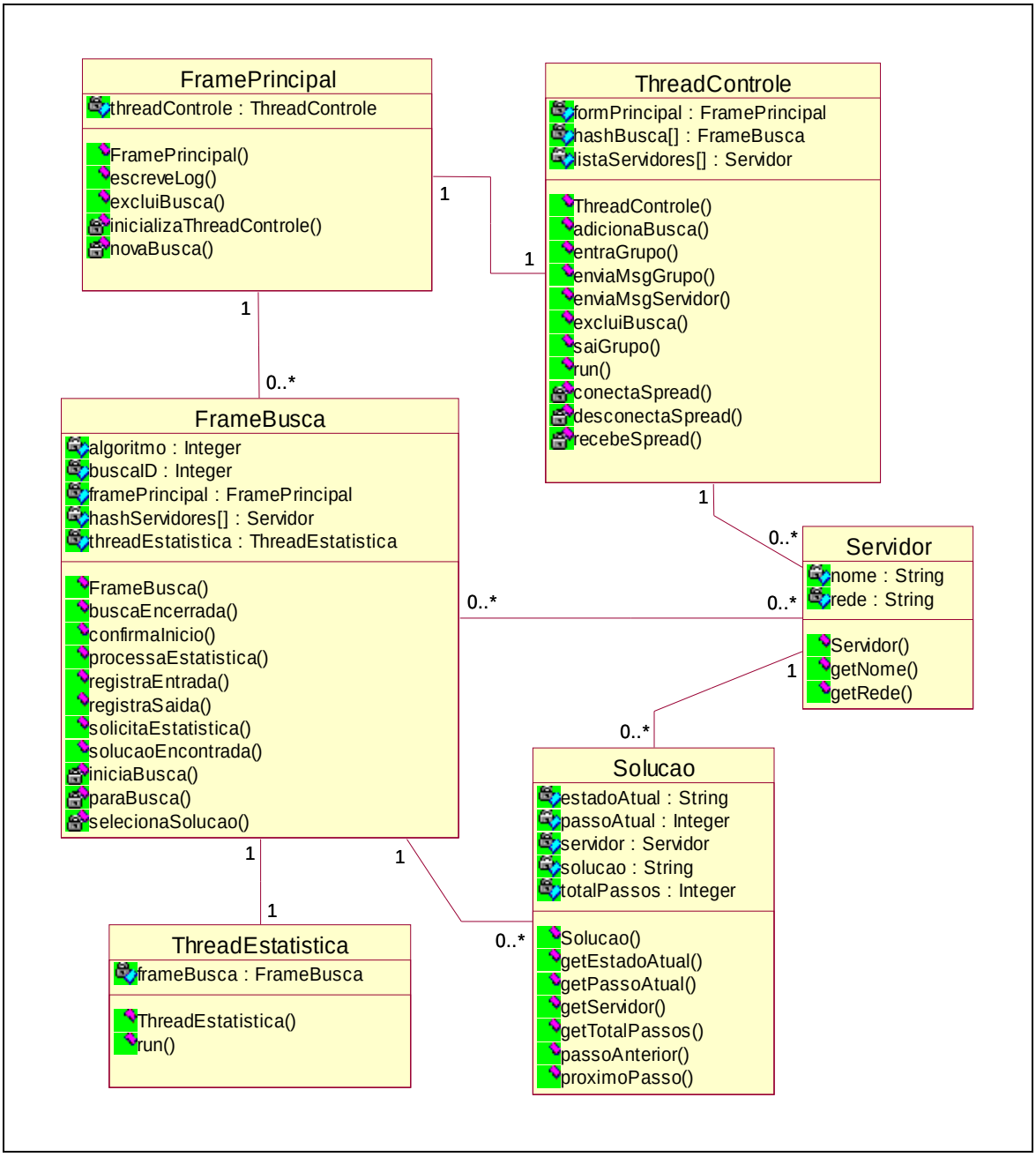


Figura 5 - Diagrama de classes do aplicativo cliente

A classe *ThreadEstatistica* também constitui uma *thread*, porém mais simples, pois sua única funcionalidade é solicitar, entre intervalos de tempo predefinidos, as estatísticas de situação do processo de busca em andamento a todos os servidores alocados naquele

momento. Cada instância da classe *FrameBusca* possui uma instância da classe *ThreadEstatistica*.

A classe *Servidor* é instanciada pela classe *ThreadControle* cada vez que um novo servidor torna-se disponível, armazenando os dados deste. Esta classe relaciona-se também com a classe *FrameBusca*, e com a classe *Solucao*, que é instanciada quando uma solução é encontrada e mantém referência para o objeto do servidor que encontrou a solução, bem como para o objeto da classe *FrameBusca* que controla o processo de busca para o qual foi encontrado a solução.

A classe *FrameBusca*, como descrito anteriormente, instancia o objeto que é responsável pelo controle de um processo de busca, além de prover os recursos visuais necessários ao usuário quanto a este. Vários objetos desta classe podem ser instanciados ao mesmo tempo, visto que o aplicativo cliente tem a característica de gerenciar vários processos de busca simultaneamente. O objeto desta classe é visto pelo usuário na forma de uma guia.

Quanto aos diagramas de seqüência, a Figura 6 apresenta o caso de uso criar busca, onde o objeto da classe *FramePrincipal* aciona o método *novaBusca()*, que irá criar um novo objeto da classe *FrameBusca*, responsável pelo controle do novo processo de busca.

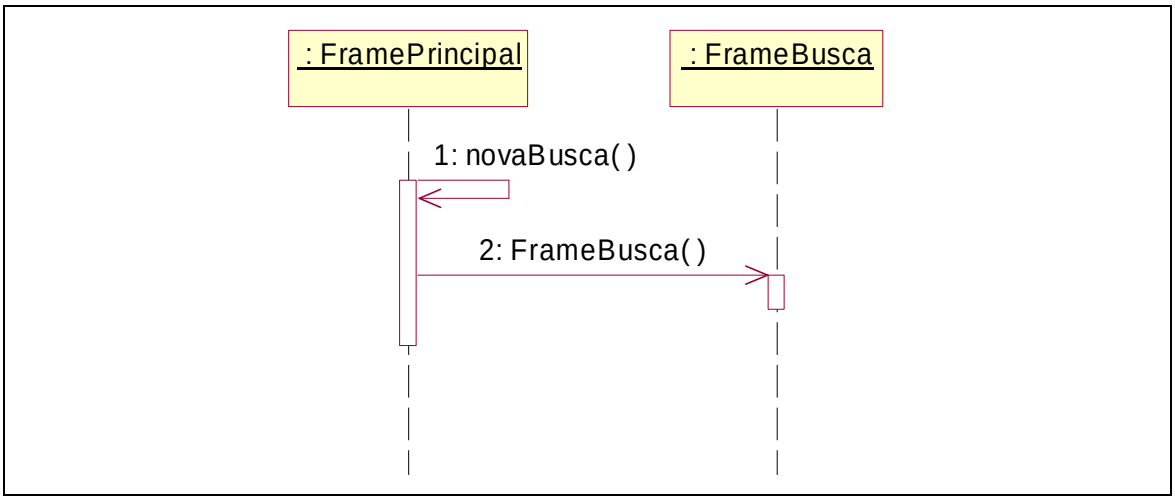


Figura 6 - Diagrama de seqüência do caso de uso criar busca

A Figura 7 ilustra o diagrama de seqüência do caso de uso executar busca, que ocorre quando o usuário já criou um novo processo de busca e deseja iniciar sua execução. O objeto da classe *FrameBusca* ativa o método *iniciarBusca()*, que por sua vez chama o método

entraGrupo(), que faz com que o aplicativo passe a fazer parte de um novo grupo de usuários, onde estarão conectados todos os servidores que fazem parte do novo processo de busca.

De cada servidor que foi selecionado para fazer parte do processo de busca, será obtido o nome do servidor e o nome da rede *Spread* da qual este pertence, através dos métodos *getNome()* e *getRede()* respectivamente. Para cada um destes servidores selecionados, será enviada uma mensagem única, separadamente, com dados de controle e a solicitação para que conecte-se ao grupo do novo processo de busca.

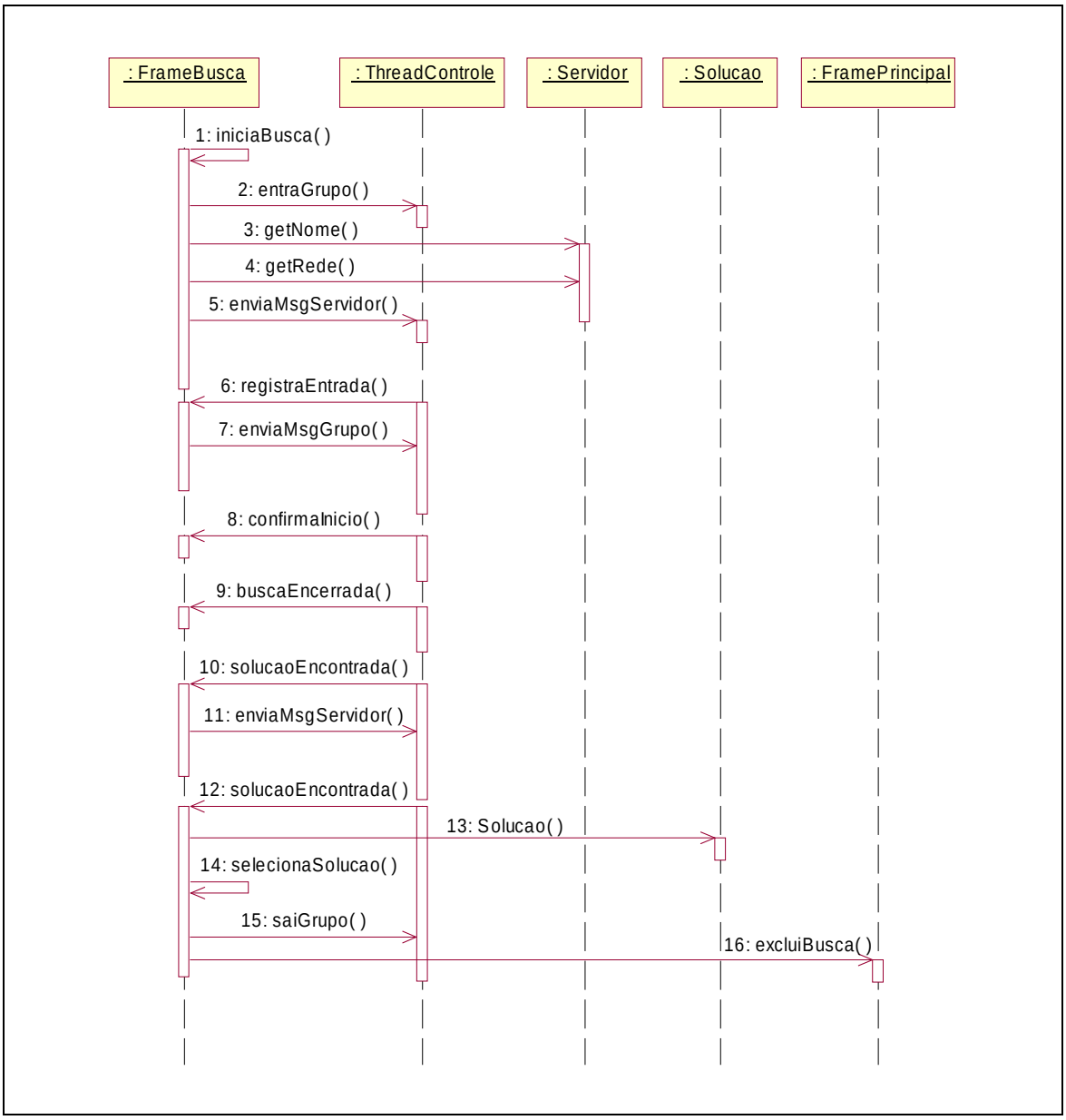


Figura 7 - Diagrama de seqüência do caso de uso executar busca

Cada vez que um servidor conectar-se ao grupo do novo processo de busca, o objeto da classe *ThreadControle* ativará o método *registraEntrada()*, que irá considerar aquele servidor como presente no grupo. Quando este método já tiver sido chamado para todos os servidores selecionados, será enviado a todos os integrantes do grupo, através do método *enviaMsgGrupo()*, uma única mensagem contendo o estado inicial e final do problema, e a ordem para o início do processo de busca.

Como o aplicativo cliente também faz parte do grupo de busca, receberá uma cópia da própria mensagem que enviou. O recebimento desta mensagem fará com que o objeto da classe *ThreadControle* ative o método *confirmaInicio()*, que irá considerar que o processo de busca foi devidamente iniciado em todos os servidores e está em execução.

Quando um servidor conclui o processo de busca sem encontrar uma solução, este envia uma mensagem para o aplicativo cliente, que será recebida pelo objeto da classe *ThreadControle*, que por sua vez irá ativar o método *buscaEncerrada()*, que registrará o término da parte do processo de busca atribuída àquele servidor.

Caso algum servidor tenha encontrado uma solução, este enviará uma mensagem que ativará, através do objeto da classe *ThreadControle*, o método *solucaoEncontrada()*. Este método irá responder à mensagem solicitando ao servidor que envie a resposta. Quando o aplicativo cliente receber a mensagem contendo a resposta, será novamente ativado o método *solucaoEncontrada()*, porém, com parâmetros diferentes, que irá então criar um novo objeto da classe *Solucao*, onde será armazenada a seqüência de passos da solução, bem como o servidor que a encontrou e a qual processo de busca a solução pertence. Posteriormente, será ativado o método *selecionaSolucao()*, que irá tornar visível ao usuário o passo inicial da solução encontrada.

O método *saiGrupo()* faz com que o aplicativo cliente desconecte-se do grupo referente ao processo de busca atual. Ele será ativado somente quando o usuário fechar a guia visual referente ao processo de busca. Posterior a isso, considerando que o objeto da classe *FrameBusca* constitui um controle visual incorporado à janela principal da aplicação, será chamado o método *excluiBusca()* no objeto da classe *FramePrincipal*, que irá destruir a instância do objeto da classe *FrameBusca* responsável pela guia visual e pelo controle do processo de busca encerrado pelo usuário. Este processo ocorre desta forma para que a janela

principal seja redesenhada corretamente quando o objeto da classe *FrameBusca* deixar de existir.

A Figura 8 apresenta o diagrama de seqüência do caso de uso cancelar busca, que ocorre quando o usuário decide cancelar o processo de busca, onde o método *paraBusca()* da classe *FrameBusca* irá chamar o método *enviaMsgGrupo()* da classe *ThreadControle*, que por sua vez irá enviar uma mensagem ao grupo do processo de busca informando do seu cancelamento. Posteriormente, o objeto da classe *FrameBusca* irá ativar o método *saiGrupo()*, que fará com que o aplicativo cliente desconecte-se do grupo do processo de busca atual.

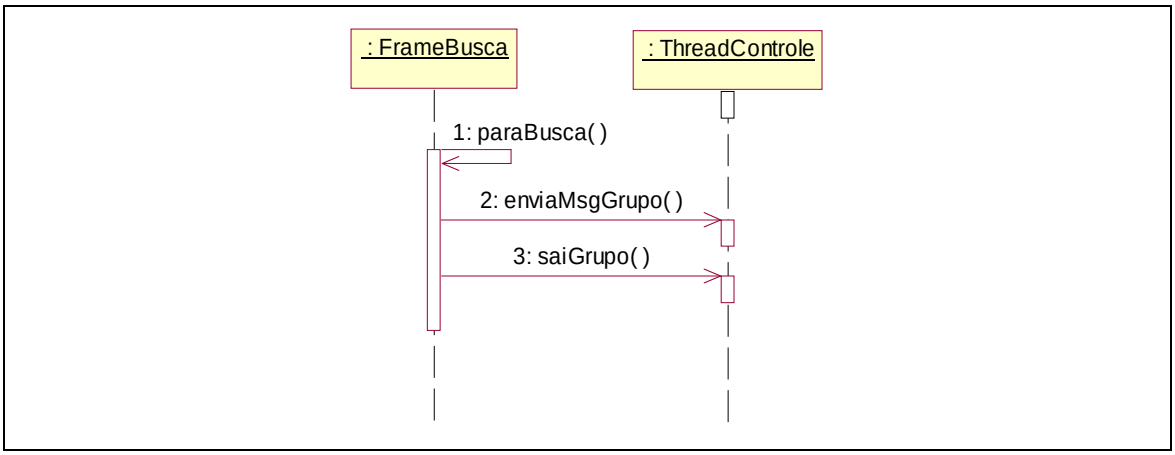


Figura 8 - Diagrama de seqüência do caso de uso cancelar busca

A Figura 9 apresenta o diagrama de seqüência do caso de uso visualizar estatística, onde o objeto da classe *ThreadEstatistica* ativa o método *solicitaEstatistica()* no objeto da classe *FrameBusca*, que por sua vez ativa o método *enviaMsgGrupo()*, responsável pelo envio da mensagem para o grupo do processo de busca solicitando dados estatísticos do mesmo.

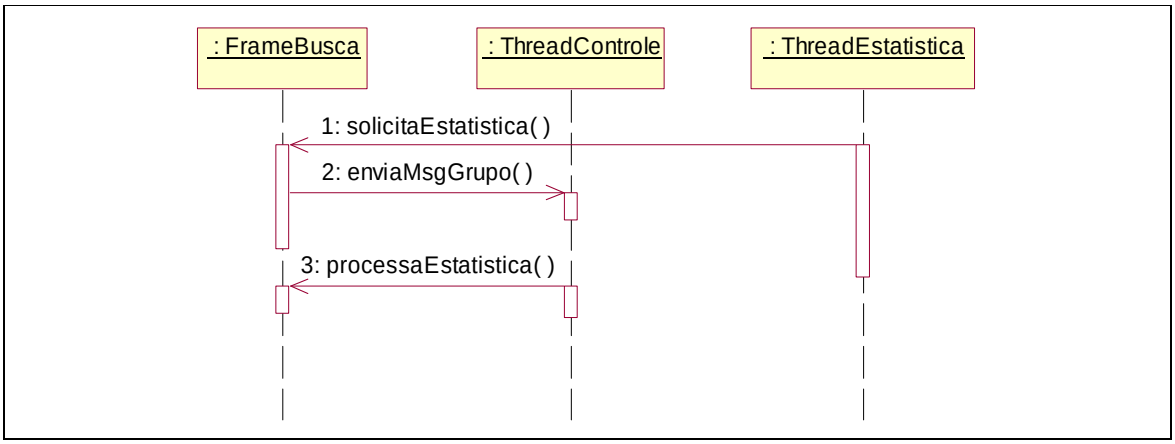


Figura 9 - Diagrama de seqüência do caso de uso visualizar estatística

Cada servidor conectado ao grupo irá responder à mensagem enviada pelo método *enviaMsgGrupo()*, e a resposta será recebida pelo objeto da classe *ThreadControle*, responsável pela ativação do método *processaEstatistica()* que irá apresentar os dados ao usuário.

O diagrama de seqüência do caso de uso visualizar solução é apresentado pela Figura 10, onde o primeiro passo da solução, equivalente ao estado inicial do problema, é apresentado ao usuário pelo método *selecionaSolucao()*.

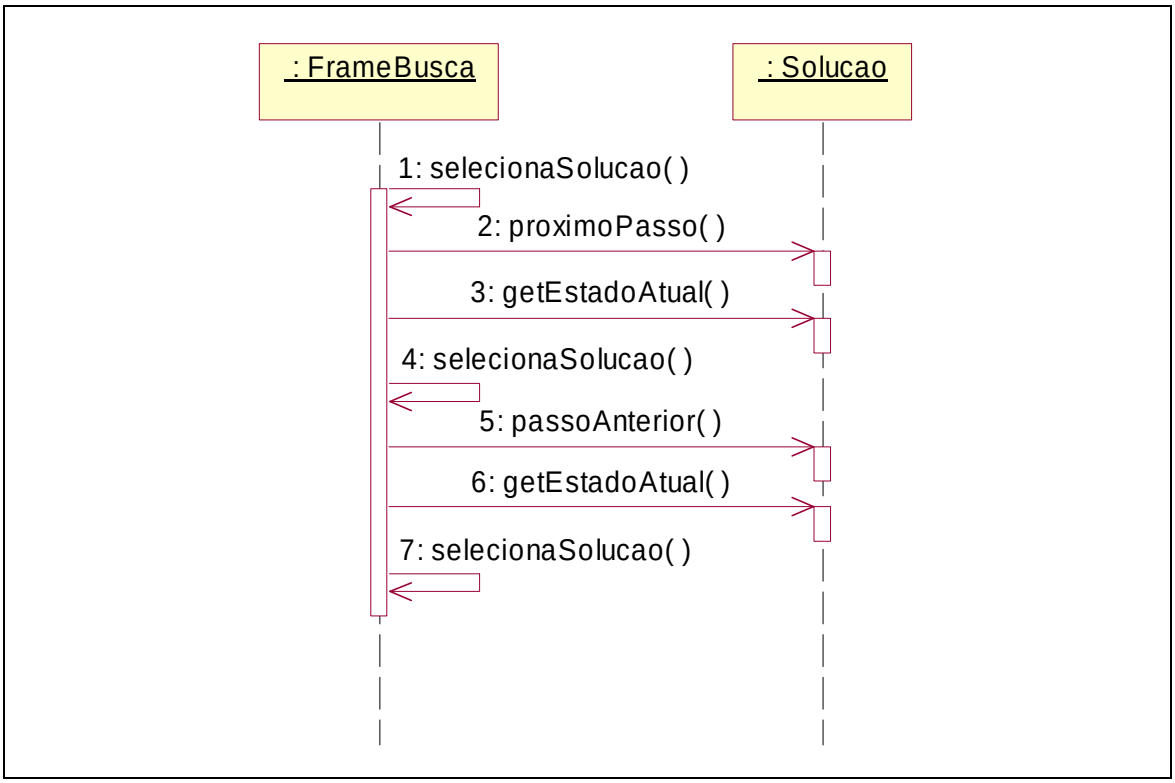


Figura 10 - Diagrama de seqüência do caso de uso visualizar solução

Caso o usuário queira visualizar o próximo passo da solução, poderá fazê-lo acionando um controle visual que ativa o método *proximoPasso()* no objeto da classe *Solucao*, o qual irá gerar o estado correspondente ao passo sucessivo ao atual. O novo estado é obtido através do método *getEstadoAtual()* e apresentado ao usuário pelo método *selecionaSolucao()*. O mesmo ocorre caso o usuário queira visualizar o passo anterior da solução, porém, o método ativado para gerar o estado de acordo com aquele passo é o *passoAnterior()*.

3.2.2 APLICATIVO SERVIDOR

A especificação do aplicativo servidor é apresentada inicialmente pelo diagrama de casos de uso, Figura 11, que ilustra os casos de uso: criar busca, executar busca, parar busca e solicitar estatística. O ator cliente representa o aplicativo cliente, visto que o aplicativo servidor não possui funcionalidades que possam ser manipuladas diretamente pelo do usuário.

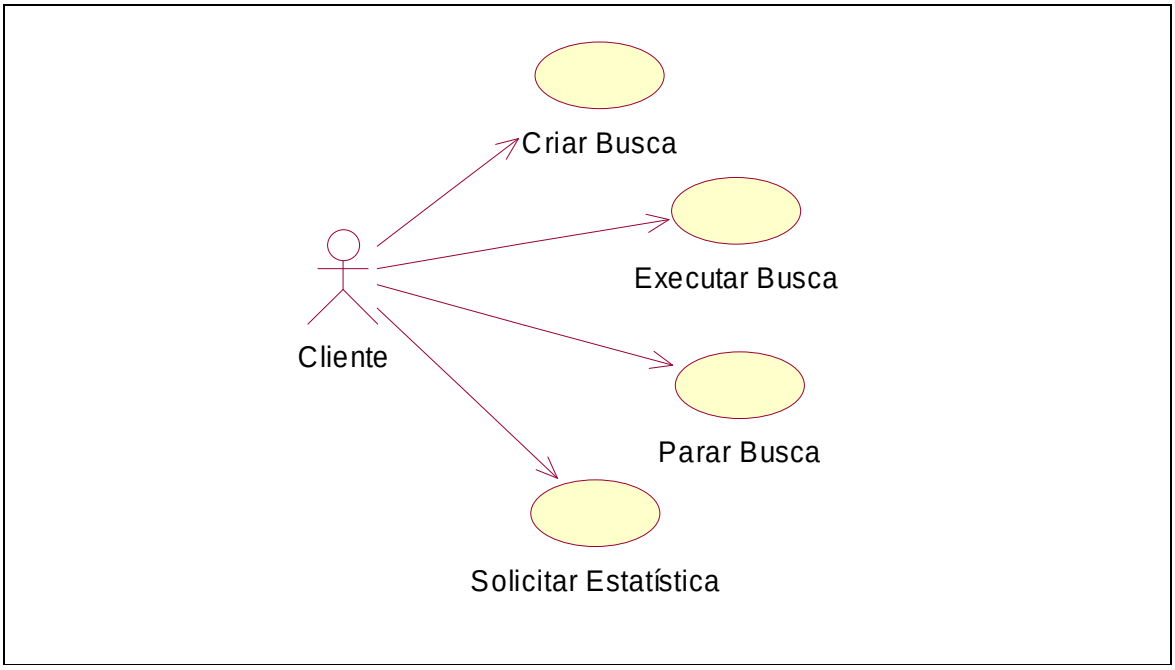


Figura 11 - Diagrama de casos de uso do aplicativo servidor

O caso de uso criar busca refere-se aos procedimentos envolvidos na instanciação e configuração de um novo processo de busca. Enquanto que o caso de uso executar busca limita-se ao início propriamente dito da execução do processo. O caso de uso parar busca refere-se à funcionalidade que permite que uma busca seja cancelada quando apenas criada ou mesmo quando já em execução. O caso de uso solicitar estatística refere-se à requisição de dados estatísticos sobre o processo de busca por parte do aplicativo cliente.

O aplicativo servidor, conforme apresenta a Figura 12, é composto pelas seguintes classes: *FramePrincipal*, *ThreadControle*, *HeapBinaria*, *Nodo*, *ThreadBusca*, classe abstrata *Busca*, e as classes de especialização: *BuscaAEstrela* e *BuscaLargura*. As classes *FramePrincipal* e *ThreadControle* são similares às classes de mesmo nome do aplicativo cliente. A diferença é que a classe *FramePrincipal* não oferece controles adicionais ao

usuário, e a classe *ThreadControle* armazena a lista de buscas em execução no servidor atual e não guarda referencias sobre os demais servidores participantes do processo de busca.

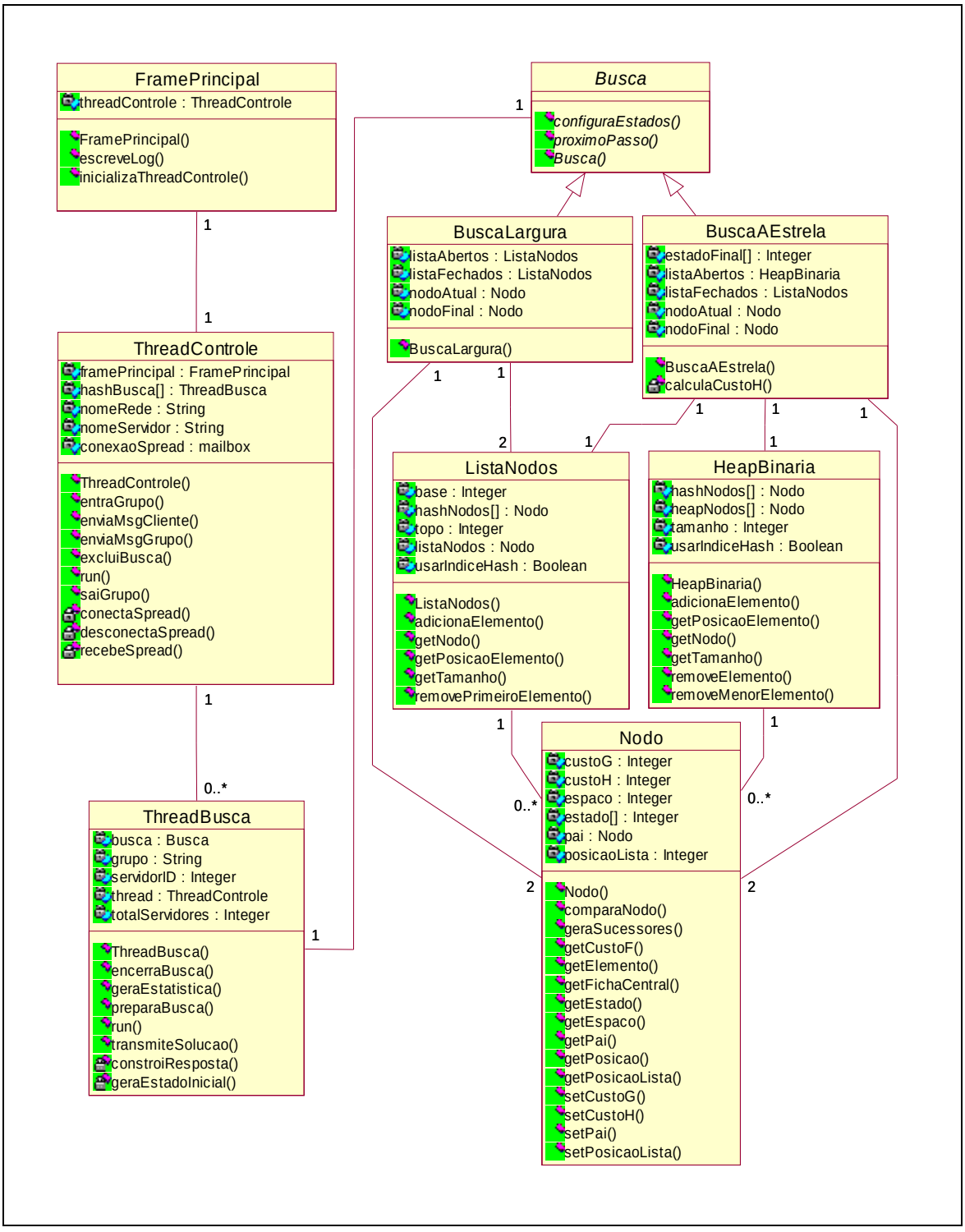


Figura 12 - Diagrama de classes do aplicativo servidor

A classe *ThreadBusca* constitui uma *thread*, cuja função é controlar o processo de busca. Relaciona-se com as classes *BuscaAEstrela* e *BuscaLargura*, que especializam a classe abstrata *Busca*. Uma das duas classes de especialização será instanciada através da classe abstrata, de acordo com a opção do usuário quanto ao algoritmo que deverá ser utilizado, sendo que cada uma das classes implementa um algoritmo de busca diferente, busca heurística A* ou busca em largura, respectivamente.

As classes *BuscaAEstrela* e *BuscaLargura* relacionam-se com a classe *Nodo*, que armazena dados sobre um determinado estado do problema e é instanciada para cada novo estado gerado. Relacionam-se também com a classe *HeapBinaria*, que contém a implementação de uma lista de ordenação binária, que oferece um mecanismo mais rápido do que vetores ou listas ordenadas, para o armazenamento de estados; e com a classe *ListaNodos*, que possui a mesma finalidade, porém sem ordenação. Os objetos mantidos pelas classes *HeapBinaria* e *ListaNodo* são instâncias da classe *Nodo*.

O diagrama de seqüência do caso de uso criar busca é apresentado na Figura 13, onde o objeto da classe *ThreadControle*, após receber uma mensagem do aplicativo cliente, cria uma nova instância da classe *ThreadBusca*, que será responsável pelo novo processo de busca. Em seguida, o novo objeto ativa o método *entraGrupo()*, que faz com que o aplicativo servidor conecte-se ao grupo referente ao novo processo de busca.

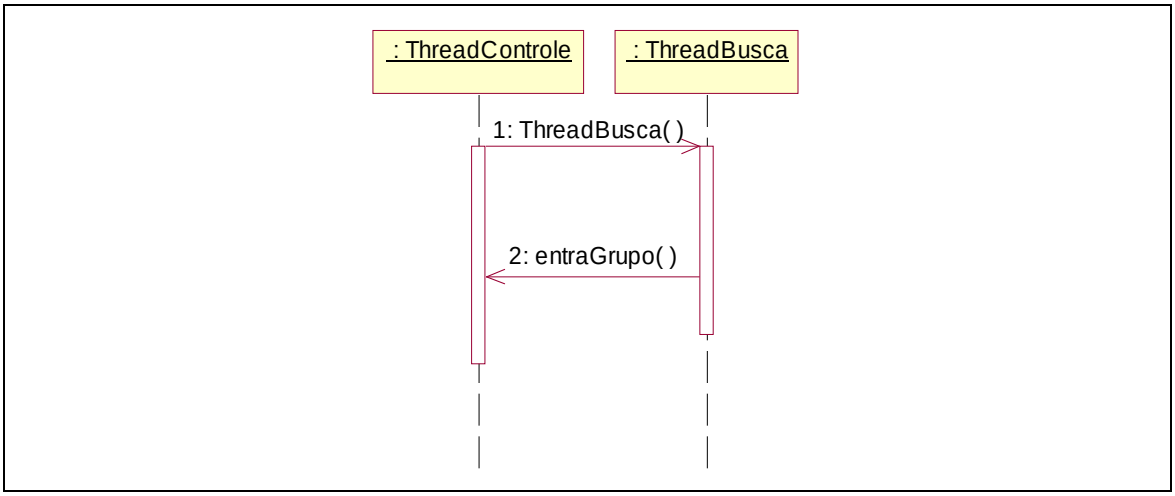


Figura 13 - Diagrama de seqüência do caso de uso criar busca

A Figura 14 ilustra o diagrama de seqüência do caso de uso executar busca, onde o objeto da classe *ThreadControle*, após receber a mensagem do aplicativo cliente, ativa o

método *preparaBusca()* no objeto da classe *ThreadBusca*, que irá criar um objeto da classe *BuscaAEstrela* ou *BuscaLargura* através da classe abstrata *Busca*, e em seguida irá ativar o método *geraEstadoInicial()*, responsável pela distinção de qual parte do problema o servidor deve tratar. Posteriormente, é ativado o método *configuraEstados()* no objeto da classe de busca, que por sua vez, irá configurar o estados inicial e final no novo objeto.

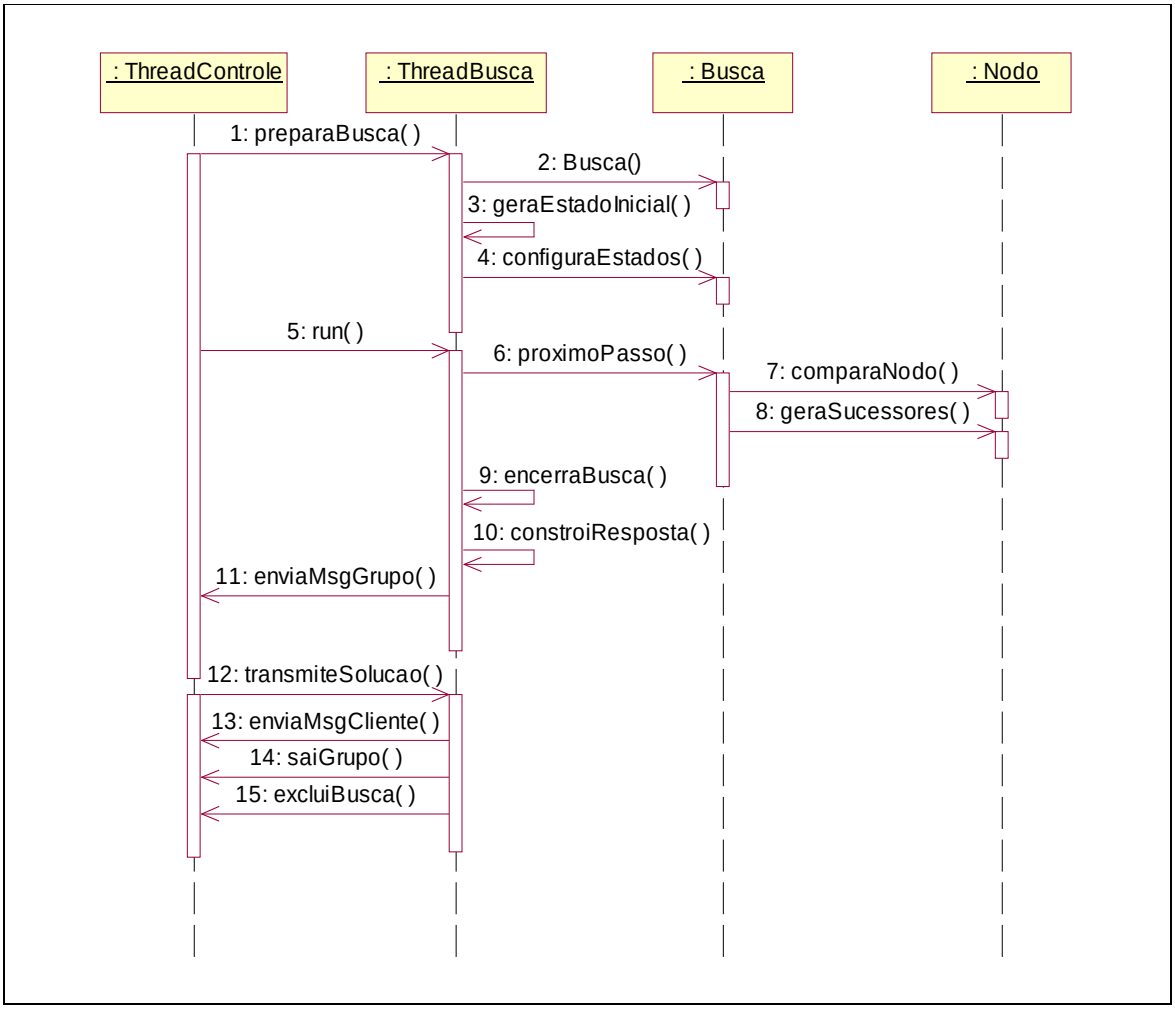


Figura 14 - Diagrama de seqüência do caso de uso executar busca

Em seguida, o objeto da classe *ThreadControle* inicia a execução da *thread* de busca através do método *run()*. Este por sua vez, irá chamar o método *proximoPasso()* no objeto de busca, que irá fazer a pesquisa pela solução do problema, usando para isso, métodos como *comparaNodo()* e *geraSucessores()* da classe *Nodo*. O método *proximoPasso()* corresponde a um passo completo do algoritmo de busca e é executado repetidamente até que o processo de busca termine, encontrando ou não uma solução.

Quando o processo de busca termina por ter ou não encontrado a solução, é ativado o método *encerraBusca()* no objeto da classe *ThreadBusca*. Se uma solução foi encontrada, o método *constroiResposta()* é ativado e irá gerar a seqüência de passos necessários para se chegar no estado final do problema a partir do estado inicial.

Em seguida, o objeto da classe *ThreadBusca* ativa o método *enviaMsgGrupo()* que envia uma mensagem para o grupo informando que aquele servidor terminou o processo de busca. Se uma solução foi encontrada e o aplicativo cliente deseja conhecê-la, uma mensagem é enviada por este e quando recebida pelo servidor, ativa o método *transmiteSolucao()*. O método *enviaMsgCliente()* é ativado e envia uma mensagem com a solução para o aplicativo cliente.

Após a transmissão da solução ou caso esta não tenha sido encontrada, é ativado o método *saiGrupo()*, que faz com que o aplicativo servidor desconecte-se do grupo referente a busca atual e logo após o método *excluiBusca()* é ativado no objeto da classes *ThreadControle*, que encerra a execução do processo de busca e destrói a instancia do objeto da classe *ThreadBusca*, responsável pelo processo de busca atual.

A Figura 15 apresenta o diagrama de seqüência do caso de uso parar busca, que ocorre quando o objeto da classe *ThreadControle* recebe uma mensagem do aplicativo cliente solicitando o cancelamento da busca. Neste processo, é ativado o método *encerraBusca()* no objeto da classe *ThreadBusca*, que finaliza os procedimentos de busca e ativa o método *saiGrupo()*, e em seguida o método *excluirBusca()*, ambos mencionados anteriormente.

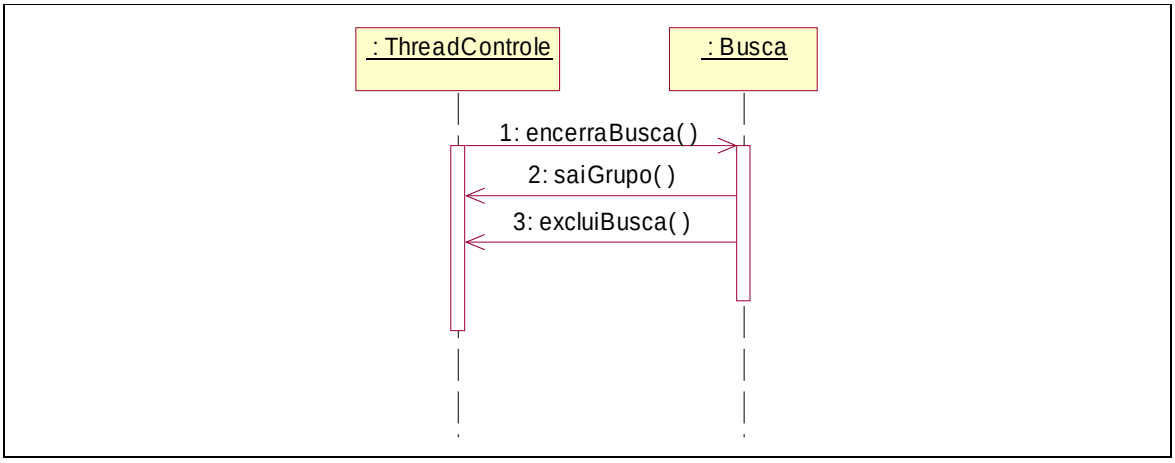


Figura 15 - Diagrama de seqüência do caso de uso parar busca

O diagrama de seqüência do caso de uso solicitar estatística é ilustrado pela Figura 16, onde o objeto da classe *ThreadControle*, após receber uma mensagem do aplicativo cliente solicitando estatística, ativa o método *geraEstatistica()* no objeto da classe *ThreadBusca*, que por sua vez monta a estatística da pesquisa.

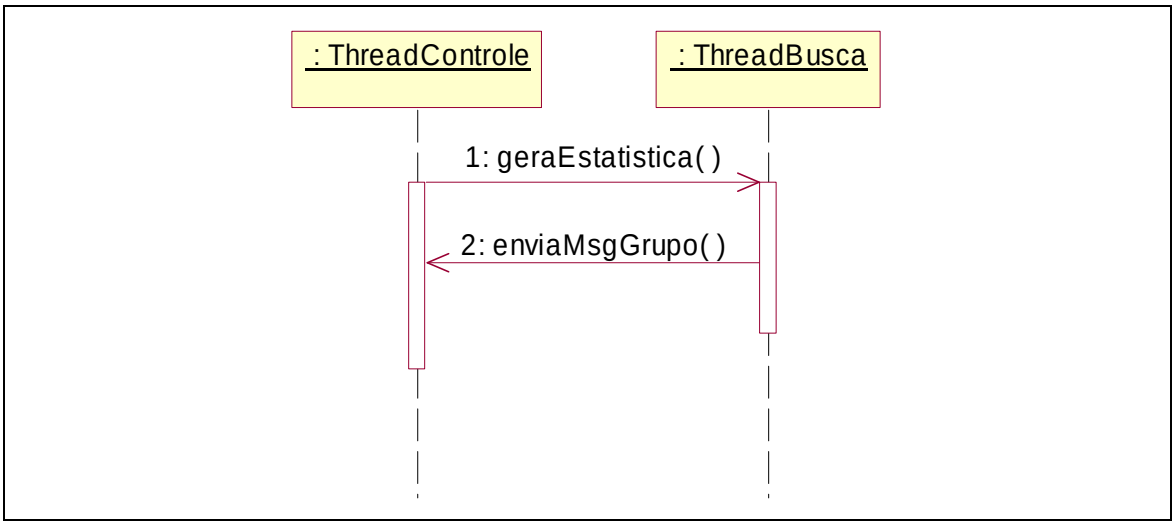


Figura 16 - Diagrama de seqüência do caso de uso solicitar estatística

Subseqüente à montagem da estatística de pesquisa pelo método *geraEstatistica()*, é ativado o método *enviaMsgGrupo()* que envia uma mensagem para o grupo de busca com a estatística daquele servidor no processo de busca atual.

3.2.3 DISTRIBUIÇÃO DA BUSCA

Quando um processo de busca é iniciado, primeiramente cada servidor selecionado pelo usuário para fazer parte do novo processo de busca recebe uma mensagem contendo a identificação da busca e o comando solicitando que conecte-se ao grupo de usuários daquele novo processo de pesquisa. Juntamente com esta mensagem, segue um número seqüencial ordenado, de forma que cada servidor tenha um número diferente.

Quando todos os servidores estiverem conectados ao grupo daquele processo de busca, o cliente irá enviar uma única mensagem para todo o grupo, informando o estado inicial e o estado final do problema, visto que é uma informação comum a todos, juntamente com o número de servidores que fazem parte do grupo.

A partir deste ponto, cada servidor irá gerar novos sucessores até que a quantidade de nodos abertos seja igual ou superior à quantidade de servidores no processo de busca atual.

Supondo que um processo de busca tenha cinco servidores, a quantidade de nodos abertos deverá ser igual ou superior a 5, como mostra a Figura 17.

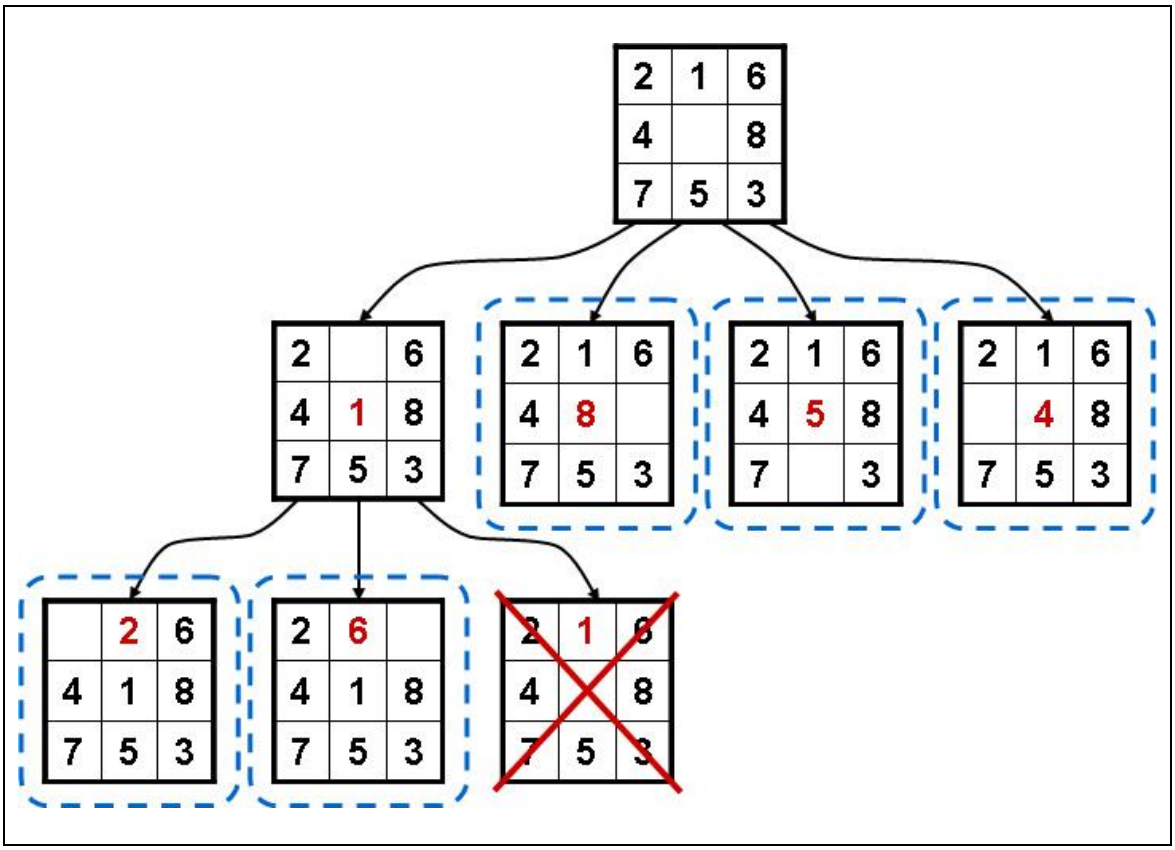


Figura 17 - Distribuição do processo de busca

Inicialmente são gerados os sucessores do estado inicial da busca, contabilizando 4 novos nodos. Como a quantidade de nodos abertos é inferior ao necessário, os sucessores do próximo nodo são gerados. Totalizam-se então 6 nodos abertos, porém o último nodo gerado é igual ao estado inicial e por esta razão é anulado pelo mecanismo de poda. Tem-se então a quantidade necessária de nodos abertos para o processo de busca.

Em seguida, cada servidor irá utilizar o número seqüencial que recebeu do aplicativo cliente como índice para obter da lista de nodos abertos o nodo que deve utilizar como estado inicial, ou seja, o servidor que recebeu o número 1 utilizará o primeiro nodo como estado inicial, enquanto que o servidor com o número 2 utilizará o segundo nodo, e assim sucessivamente.

3.3 IMPLEMENTAÇÃO

Este tópico apresenta as técnicas e ferramentas utilizadas no desenvolvimento do trabalho, como: método de desenvolvimento e implementação, ambiente e linguagem de programação. É apresentada também a explanação, com partes do código-fonte, das funcionalidades mais relevantes do sistema, como: interligação com o mecanismo de comunicação em grupo *Spread*, troca de mensagens entre o cliente e os servidores, mecanismo de divisão de tarefas distribuídas, distinção dos objetos de busca, função heurística para cálculo de custo de um nodo, comparação entre dois nodos, geração de sucessores de um nodo, algoritmo de busca A*, listas binárias ordenadas com índice *hash*, mecanismo de montagem e apresentação da solução e mecanismo de controle do término de um processo. Por fim, é apresentado também um teste operacional do sistema através de um estudo de caso.

3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

O processo de implementação deu-se ao término da especificação de cada nova parte do trabalho, seguido por uma fase de testes do tipo caixa aberta, para verificar a funcionalidade da parte implementada. E por fim, uma fase de testes tipo caixa fechada, para comprovar o funcionamento da nova parte com o restante da aplicação, bem como averiguar se a nova parte implementada não afetaria nenhuma outra parte já existente na aplicação.

A linguagem de programação escolhida para o desenvolvimento do trabalho apresentado é a linguagem *C++*. Esta escolha deu-se principalmente pelo fato de que o mecanismo de comunicação em grupo *Spread*, utilizado no trabalho, é escrito e oferece uma API completa nesta mesma linguagem.

Verificou-se ainda durante a revisão bibliográfica, a existência desta mesma API para a linguagem *Java*, bem como bibliotecas de adaptação da API original do *Spread* para plataformas como o *Delphi*, porém estas linguagens não foram utilizadas para que não houvesse riscos de comprometimento de um dos objetivos do trabalho, que é o alto desempenho da aplicação, visto que o uso de uma biblioteca de adaptação da API para outra linguagem causaria um acréscimo no tempo de execução, e pela linguagem *Java* não ser o padrão da API do mecanismo *Spread*.

Verificou-se também durante a revisão bibliográfica, a possibilidade da utilização de determinadas bibliotecas que possibilitam que o mesmo código fonte seja compilado em múltiplas plataformas, como é o caso da biblioteca *wxWidgets* apresentada anteriormente. Desta forma, visto que atualmente há uma crescente utilização de outras plataformas que não a *Windows,* por instituições e usuários finais, optou-se pela utilização desta biblioteca a fim de permitir portabilidade.

Quanto ao ambiente de desenvolvimento, foi selecionado o *Microsoft Visual C++* para a plataforma *Windows*, por oferecer um bom sistema de depuração. Na plataforma *Linux* utilizou-se o compilador *gcc* por recomendação dos desenvolvedores da biblioteca *wxWidgets.* O mesmo código-fonte foi compilado em ambas as plataformas, sem nenhuma alteração.

3.3.2 PROCESSO DE IMPLEMENTAÇÃO

Quanto às funcionalidades implementadas, a interligação com o mecanismo *Spread* dá-se por meio de uma *thread* que é executada no momento em que a aplicação inicia e só finaliza em caso de erro na conexão ou quando a aplicação é encerrada, tanto no aplicativo cliente quando servidor. Inicialmente a *thread* tenta estabelecer uma conexão com o servidor *Spread*, e caso ocorra algum erro grave, como nome de usuário inválido, a *thread* é encerrada. No caso de um erro por inatividade do servidor *Spread*, a *thread* continua em execução e tenta novamente estabelecer uma conexão em intervalos de tempo predefinidos.

A verificação de erro é realizada através do teste do valor de retorno da chamada da função da API do *Spread* que está sendo utilizada em um dado momento, conforme mostra o código-fonte do Quadro 4, onde inicialmente é feita a chamada da função *SP_connect*, responsável pela conexão com o servidor *Spread*. Caso a conexão seja estabelecida, é feita a chamada à função *SP_join*, para a entrada no grupo geral de usuários, ou grupo de controle.

```
// chamada para a função de conexão da API Spread
iRetConexao = SP_connect(SPREAD_HOST, C_SPREAD_NOME, 0, 1, &conexaoSpread,
grupo);

// testa se a conexão está OK
if (iRetConexao == ACCEPT_SESSION) {
      // entra no grupo principal
      iRetJoin = SP_join(conexaoSpread, SPREAD_GRUPO);
}
```

Quadro 4 - Retorno de chamada das funções da API do *Spread*

Neste trabalho também foram utilizadas as funções de saída de um grupo e de desconexão do servidor *Spread*, *SP_leave* e *SP_disconnect*, respectivamente, conforme mostrado pelo código-fonte do Quadro 5. Um ponto em comum entre as funções da API do *Spread*, é necessidade de fornecer um identificador de conexão, que sempre é passado como parâmetro para a função, e que no caso dos exemplos, é a variável *conexaoSpread*.

```
// sai de um grupo
SP_leave(conexaoSpread, grupo);

// desconecta do servidor Spread
SP_disconnect(conexaoSpread);
```

Quadro 5 - Função de saída de um grupo e de desconexão do servidor *Spread*

Ainda quanto a utilização das funções da API do *Spread*, há também a função *SP_multicast*, responsável pelo envio das mensagens, e a função *SP_receive*, responsável pelo recebimento das mesmas. A função *SP_multicast* pode ser utilizada para o envio de mensagens a um grupo de usuários ou apenas para um usuário em específico, conforme mostram os métodos implementados no código do Quadro 6, onde *mu_vEnviaMsgCliente* implementa o envio de mensagens para o cliente e *mu_vEnviaMsgGrupo* implementa o método que envia mensagens para um grupo.

A distinção entre um grupo ou usuário é feita através do formato do parâmetro de destino da mensagem passado para a função da API do *Spread*. No caso de um grupo, o parâmetro conterá, por exemplo, *busca1*. No caso de um usuário em específico, o parâmetro irá conter o nome do usuário seguido pelo nome do segmento ao qual ele pertence, ambos precedidos pelo sinal de sustenido (#), por exemplo: *#usuario#localhost*.

```
// envia uma mensagem para o cliente
void ThreadControle::mu_vEnviaMsgCliente(int iTipo, wxString sMensagem)
{
      SP_multicast(conexaoSpread, SAFE_MESS, wxT("#" + C_SPREAD_NOME + "#"
+ ai_sNomeRede), iTipo, sMensagem.size(), wxT(sMensagem));
}

// envia uma mensagem para um grupo
void ThreadControle::mu_vEnviaMsgGrupo(wxString sGrupo, int iTipo, wxString
sMensagem)
{
      SP_multicast(conexaoSpread, SAFE_MESS, wxT(sGrupo), iTipo,
sMensagem.size(), wxT(sMensagem));
}
```

Quadro 6 - Métodos de envio de mensagem para um grupo e para um usuário

Um outro ponto importante é a maneira como cada servidor sabe em qual parte da árvore de busca deve pesquisar, o que é determinado por um mecanismo de divisão de tarefas distribuídas. Este mecanismo é caracterizado pelo método *geraEstadoInicial()* da classe *ThreadBusca*.

O método *geraEstadoInicial()* consiste na implementação do algoritmo de busca em largura, apresentado no Quadro 7, que irá gerar novos sucessores até que o número de nodos abertos seja igual ou superior ao número de servidores trabalhando no processo de busca. Há também um mecanismo de poda, que anula nodos com estados iguais, e o teste do nodo meta, que verifica se a solução não está em um dos primeiros sucessores do estado inicial do problema.

```
// enquanto a quantidade de nodos abertos for menor que a
// quantidade de servidores na busca, gera novos nodos
while ((!bEncontrouSolucao) && (aoListaAbertos->mu_iGetTamanho() <
ai_iTotalServidores)) {
      // pega o primeiro nodo da lista de abertos
      oNodoAtual = aoListaAbertos->mu_vRemovePrimeiroElemento();
      // gera os sucessores do nodo atual
      // o vetor de sucessores é passado como referência
      iSucessores = oNodoAtual->mu_iGeraSucessores(aoListaSucessores);
      // testa cada um dos sucessores
      iIndSucessores = 0;
      while ((!bEncontrouSolucao) && (iIndSucessores < iSucessores)) {
            // captura um dos sucessores
            oNovoNodo = aoListaSucessores[iIndSucessores];
            // testa se o sucessor está na lista de abertos
            if (aoListaAbertos->mu_iGetPosicaoElemento(oNovoNodo) < 0) {
                  // não está na lista de abertos
                  // testa se não é o nodo meta
                  if (!oNovoNodo->mu_bComparaNodo(oNodoInicial)) {
                        // define o nodo atual como pai
                        oNovoNodo->mu_vSetPai(oNodoAtual);
                        // adiciona na lista de abertos
                    aoListaAbertos->mu_vAdicionaElemento(oNovoNodo);
                  } else {
                        // encontrou a solução
                        bEncontrouSolucao = TRUE;
                  }
            }
            iIndSucessores++;
      }
}
```

Quadro 7 - Mecanismo de divisão de tarefas distribuídas

Feita a geração dos sucessores em quantidade suficiente, cada servidor irá obter o nodo que deve utilizar como estado inicial, conforme implementa o código-fonte apresentado no Qaudro 8. Cada servidor utilizará o número seqüencial que recebeu do aplicativo cliente como

índice, para obter da lista de nodos abertos o nodo que deve considerar como estado inicial. Porém, como determinados estados podem gerar dois, três ou quatro sucessores, pode ocorrer de que sejam gerados mais sucessores do que servidores, e neste caso, o servidor com o maior número seqüencial irá se responsabilizar pelo processamento de todos os nodos remanescentes.

```
// testa se a solução não foi encontrada
if (!bEncontrouSolucao) {
      // testa se o servidor é o último da lista
      if (ai_iServidorID == ai_iTotalServidores) {
            // o servidor deve processar os nodos remanescentes
            for (iNodos = (ai_iServidorID - 1); iNodos < aoListaAbertos-
>mu_iGetTamanho(); iNodos++) {
                  aoListaRemanescentes[iTamRemanescentes] = aoListaAbertos-
>mu_oGetNodo(iNodos);
                  iTamRemanescentes++;
                  }
      } else {
            // nodo correspondente ao ID  do servidor da lista de abertos
            aoListaRemanescentes[0] = aoListaAbertos-
>mu_oGetNodo(ai_iServidorID - 1);
            iTamRemanescentes++;
      }
} else {
      // a solução foi encontrada
      // define o nodo inicial como ponto de início
      aoListaRemanescentes[0] = oNodoInicial;
      iTamRemanescentes++;
}
```

Quadro 8 - Obtenção do estado inicial em uma busca distribuída

Caso a solução do problema seja encontrada dentre os primeiros sucessores do estado inicial ainda na divisão do processo de busca, é atribuído o estado inicial recebido do cliente como estado inicial para o processo de busca. Desta forma garante-se que o melhor caminho até a solução será encontrado, independente do algoritmo que está sendo utilizado no processo de busca.

No contexto de gerenciamento dos objetos de busca, destaca-se a utilização de tabelas *hash* para armazenar os objetos, visto que tanto o cliente pode controlar quanto os servidores podem processar mais de uma busca simultaneamente. Ambos os aplicativos utilizam tabelas *hash* para referenciar os objetos de busca, pois sempre que ocorre uma troca de mensagem, é fornecido juntamente com esta o identificador da busca, um nome gerado pelo cliente, que caracteriza o índice na tabela *hash*.

Desta forma, tem-se uma melhoria no desempenho de acesso aos objetos de busca, pois descarta-se a necessidade de se fazer uma varredura seqüencial em um vetor de objetos procurando pelo objeto que deve tratar o conteúdo da mensagem. O Quadro 9 ilustra o código-fonte onde é instanciado um novo objeto de busca no aplicativo servidor, em que cada objeto constitui uma *thread*, que em seguida é criada.

```
// cria um novo índice na tabela hash
ai_oHashBusca.insert(sGrupoBusca);
// cria um novo objeto de busca
ai_oHashBusca[sGrupoBusca] = new Busca(sGrupoBusca, iIDBusca, iIDServidor,
this);

// testa a criação da thread
if (ai_oHashBusca[sGrupoBusca]->Create() != wxTHREAD_NO_ERROR) {
     wxLogError(wxT("Não foi possível criar a Thread para a busca " +
sGrupoBusca + "!"));
}
```

Quadro 9 - Utilização de tabela *hash* no armazenamento de objetos de busca

Quanto ao processo de busca propriamente dito, no caso do algoritmo de busca heurística A*, tem-se a necessidade de utilização de um mecanismo de comparação entre nodos que permita ao processo optar pelo nodo que possua maior probabilidade de alcançar a resposta do que os demais. A parte mais importante deste mecanismo diz respeito à função heurística, ou seja, uma função que gere um custo probabilístico em relação à suposta distância entre o nodo atual e o nodo meta do problema.

O aplicativo servidor implementa um método de cálculo heurístico baseado em duas fórmulas de custo, conforme mostra o código apresentado pelo Quadro 10. Inicialmente é utilizado o conceito de distância linear, também conhecido como distância *Manhattan* (LESTER, 2004), caracterizado pelo deslocamento das peças em relação ao ponto em que deveriam estar. Em seguida utiliza-se o conceito da seqüência de *Nilsson* (JONES, 2004), que testa se o primeiro elemento ao lado de uma ficha após a borda do tabuleiro em sentido horário é o elemento correto, exceto para espaço central. Para cada ficha fora do lugar, são atribuídos custos que ao final são somados e geram o custo heurístico total.

Há também a necessidade da implementação de um mecanismo de comparação de estados entre dois nodos que seja eficiente e devolva a resposta da maneira mais ágil possível, visto que é um processo que se repete muitas vezes dependendo dos estados do problema, e por tal razão pode demandar muito tempo de processamento.

```
int   BuscaAEstrela::mi_iCalculaCustoH(Nodo *oNodo)
{
      // declaração de variáveis
      int iResS = 0, iResH = 0, iPosicao = 0, iLoop, iDeslX, iDeslY,
iCentro;
      // vetor de índice dos sucessores
      int aiSucessor[9] = {1, 2, 5, 0, -1, 8, 3, 6, 7};
      // captura a ficha central do estado
      iCentro = oNodo->mu_iGetFichaCentral();
      // testa se a ficha central é a mesma do estado final
      if (ai_aiEstadoFinal[4] != iCentro) {
            // não é - peso 1
            iResS = iResS + 1;
      }
      // calcula o deslocamento de cada ficha ignorando o espaço -> 0
      for (iLoop = 1; iLoop < 9; iLoop++) {
            // HEURISTICA - H
            // captura o indice da ficha i
            iPosicao = oNodo->mu_iGetPosicao(iLoop);
            // calcula a posição X
            iDeslX = abs(ai_aiPosicao[iLoop][0] - (iPosicao % 3));
            // calcula a posição Y
            iDeslY = abs(ai_aiPosicao[iLoop][1] - (iPosicao / 3));
            // soma tudo
            iResH = iResH + (iDeslX + iDeslY);

            // DESLOCAMENTO - S (score)
            // testa se não é a ficha central e o sucessor está correto
            if ((iLoop != iCentro) && (oNodo-
>mu_iGetElemento(aiSucessor[iPosicao]) !=
ai_aiEstadoFinal[aiSucessor[ai_aiPosicao[iLoop][2]]])) {
                  // soma peso 3 ao deslocamento
                  iResS = iResS + 3;
            }
      }
      // soma os valores
      iResH = iResH + iResS;
      // retorna o valor calculado
      return iResH;
}
```

Quadro 10 - Função de cálculo do custo heurístico de um nodo

Sendo assim, implementou-se um comparador de estados que retorna a resposta negativa de igualdade assim que encontra a primeira diferença, e que só verifica todas as posições do estado no caso de nodos iguais. A implementação do mecanismo de comparação é apresentada no Quadro 11.

Outro mecanismo que necessita ser eficiente é o de geração de sucessores de um nodo, que precisa, inicialmente, conhecer o estado atual para então analisar quais estados podem ser gerados a partir daquele e por fim criar fisicamente os nodos correspondentes. Visando obter um melhor desempenho, optou-se por armazenar nos objetos de nodo a posição do espaço em

branco naquele estado, dispensando a necessidade de testar a sua posição a cada repetição do processo.

```
bool Nodo::mu_bComparaNodo(Nodo *oNodo)
{
	int i = 0;
	bool igual = TRUE;

	// passa por todos os elementos dos nodos
	while ((i < 9) && (igual)) {
		// testa se os elementos são diferentes
		if (m_pEstado[i] != oNodo->getElemento(i)) {
			// são diferentes
			igual = FALSE;
		} else {
			i++;
		}
	}
	// retorna o resultado da comparação
	return igual;
}
```

Quadro 11 - Mecanismo de comparação de estados de dois nodos

Desta forma, o processo de geração simplificou-se em testar a posição conhecida do espaço em branco, e de acordo com esta, gerar os sucessores do estado atual. Utilizou-se uma estrutura de comparação seqüencial, onde de acordo com a posição do espaço em branco, os sucessores já são criados de imediato, conforme ilustra o Quadro 12, onde é apresentada a geração dos sucessores de um estado onde o espaço em branco encontra-se no canto superior esquerdo do tabuleiro, estado que acarreta a geração de dois novos sucessores, que são devolvidos pelo método na forma de um vetor.

Conforme os conhecimentos obtidos no processo de revisão bibliográfica a respeito dos algoritmos de busca, optou-se por utilizar neste trabalho o algoritmo A*, por permitir através do uso da heurística um melhor desempenho no processo de busca, garantindo a obtenção da solução, caso exista, geralmente no melhor tempo. O aplicativo servidor implementa o algoritmo A* com o uso de *threads*, o que extingue do método o uso de uma estrutura de repetição principal, que engloba os demais procedimentos, visto que a *thread* fornece esta funcionalidade. Foi implementado também o algoritmo de busca em largura para fins de comparação com o algoritmo A*.

```
// testa a posição do espaço em branco
if (m_pEspaco == 0) {
      // gera 2 sucessores
      resultado = 2;
      // gera array do estado
      estado[0] = m_pEstado[1];
      estado[1] = m_pEstado[0];
      estado[2] = m_pEstado[2];
      estado[3] = m_pEstado[3];
      estado[4] = m_pEstado[4];
      estado[5] = m_pEstado[5];
      estado[6] = m_pEstado[6];
      estado[7] = m_pEstado[7];
      estado[8] = m_pEstado[8];
      // armazena os sucessores
      m_pListaSucessores[0] = new Nodo(estado);
      // gera array do estado
      estado[0] = m_pEstado[3];
      estado[1] = m_pEstado[1];
      estado[2] = m_pEstado[2];
      estado[3] = m_pEstado[0];
      estado[4] = m_pEstado[4];
      estado[5] = m_pEstado[5];
      estado[6] = m_pEstado[6];
      estado[7] = m_pEstado[7];
      estado[8] = m_pEstado[8];
      // armazena os sucessores
      m_pListaSucessores[1] = new Nodo(estado);
}
```

Quadro 12 - Mecanismo de geração de sucessores de um nodo

Da forma como foi implementado, o processo de repetição da busca passa a ser controlado pela ***thread***, de forma que cada execução desta resulta na chamada do método do algoritmo, que consequentemente causa um passo completo na sua execução, ou seja, no caso do algoritmo A*, a cada chamada de execução feita pela ***thread***, um novo elemento é retirado da lista de abertos, comparado ao nodo final, são gerados os seus sucessores, que são então adicionados à lista de nodos abertos ou fechados, dependendo dos seus estados e custos. Este processo é repetido até que o nodo removido da lista de abertos seja o nodo meta, ou até que não haja mais nodos na lista de abertos. A estrutura de repetição da ***thread*** é controlada pelo retorno da função do algoritmo, como mostra o código-fone do Quadro 13.

```
while (ai_bContinuaBusca && !TestDestroy()) {
      // próximo passo da busca
      iRetorno = ai_oBusca->mu_iProximoPasso();
      // testa o retorno da busca
      if (iRetorno > BUSCA_CONTINUA) {
            ai_bContinuaBusca = FALSE;
      }
}
```

Quadro 13 - Estrutura de repetição da ***thread*** de busca

Considerando que o algoritmo A* utiliza sempre o nodo com menor custo para continuar o processo de busca, utilizando-se vetores normais para as listas de nodos abertos e fechados, pode-se ter uma demora significativa no processo de obtenção do nodo com menor custo, pela necessidade de se fazer uma varredura linear completa no vetor.

A utilização de listas ordenadas reduz essa demora, por reorganizar a seqüência dos elementos a cada vez que um deles é removido ou adicionado. Mas mesmo assim, a utilização de um mecanismo de ordenação inadequado para um processo que tende a ser executado várias vezes por segundo, pode acabar por comprometê-lo.

Sendo assim, utilizou-se para o armazenamento dos nodos abertos, uma lista binária ordenada, que reorganiza seus elementos cada vez que um deles é adicionado ou removido, mas sem a necessidade de testar todos os elementos da lista.

A idéia das listas binárias ordenadas consiste em, ao adicionar um novo elemento, posicioná-lo ao final da lista e em seguida compará-lo com o elemento que encontra-se na posição referente a metade do índice atual, por exemplo: um novo elemento inserido ao final da lista, com índice 20, será comparado com o elemento na posição 10 da lista. Se o elemento ao final da lista tiver um valor menor, ele troca de lugar com o elemento ao qual foi comparado. Em seguida, é comparado novamente com o elemento no índice correspondente a metade do atual, ou seja, 5. Se for menor, os elementos são novamente trocados. Este processo se repete até que o elemento alcance a primeira posição da lista ou até que seja maior do que o elemento com o qual está sendo comparado.

Seguindo este princípio, o elemento de menor custo estará sempre na primeira posição da lista, e quando for removido, ocorre o processo inverso ao da adição. O último elemento da lista é colocado na primeira posição e é comparado com o elemento com o dobro do seu índice, e com o elemento com o dobro do seu índice mais um. Caso seja maior que um dos elementos ao qual foi comparado, é trocado com aquele que for menor, e assim sucessivamente. A implementação da função de adição de um novo elemento utilizando este processo de ordenação é apresentada em detalhes no Quadro 14.

```
// repete enquanto i não for menor que 2 e enquanto o custo da
// posição da metade do indice atual for inferior ao custo atual
while ((iTemp > 1) && (ai_aoHeapNodos[iTemp - 1]->mu_iGetCustoF() <
ai_aoHeapNodos[(iTemp / 2) - 1]->mu_iGetCustoF())) {
      //troca as posições dos nodos
      auxN = ai_aoHeapNodos[(iTemp / 2) - 1];
      ai_aoHeapNodos[(iTemp / 2) - 1] = ai_aoHeapNodos[iTemp - 1];
      ai_aoHeapNodos[iTemp - 1] = auxN;
      // testa se está usando indice hash
      if (ai_bUsarIndiceHash) {
            // atualiza o índice no nodo
            ai_aoHeapNodos[(iTemp / 2) - 1]->mu_vSetPosicaoLista((iTemp /
2) - 1);
            ai_aoHeapNodos[iTemp - 1]->mu_vSetPosicaoLista(iTemp - 1);
      }
      // controle do índice
      iTemp = (iTemp / 2);
}
```

Quadro 14 - Adicionando um elemento em uma lista binária ordenada

Implementou-se também juntamente com as listas binárias ordenadas, um mecanismo opcional, ativado a critério do usuário, que armazena todos os nodos da lista em uma tabela *hash*, utilizando para isso a concatenação do seu estado como índice. No objeto do nodo é armazenada a sua posição na lista, e desta forma, sempre que for necessário consultar a existência de um nodo com determinado estado em uma lista, basta fazê-lo através da tabela *hash*. Se não houver elemento com índice igual à concatenação do estado procurado, é porque o objeto não está naquela lista. Porém, caso exista um elemento com aquele índice, consultando o objeto é possível saber exatamente qual a sua posição na lista. Desta forma, utiliza-se mais memória, mas em contra partida obtém-se um aumento considerável no desempenho da aplicação.

Ao término da execução do processo de busca, caso uma solução tenha sido encontrada, há a necessidade de repassar esta informação para o aplicativo cliente. Normalmente, seriam transmitidos os estados de cada nodo, desde o estado inicial até o estado final do problema, de forma ordenada. Porém, considerando que só interessa ao usuário a solução do problema como um todo, e que este dificilmente irá analisar apenas uma parte da solução, optou-se por implementar um mecanismo que transmita ao aplicativo cliente apenas os passos para a solução do problema, ou seja, a movimentação do espaço em branco, desde o estado inicial até o estado final.

Transmitindo apenas a movimentação do espaço em branco, tem-se que uma mensagem que antes transmitiria, por exemplo, dez estados da solução, pode então transmitir,

com a mesma quantidade de informação, noventa estados distintos. A implementação da função que armazena os passos necessários entre o estado inicial e final do problema é apresentada pelo Quadro 15, onde o último nodo processado, equivalente ao estado inicial do problema, é selecionado e posteriormente obtém-se dele a posição do espaço em branco. Em seguida, através do atributo *pai* do objeto da classe *Nodo*, pode-se chegar ao nodo antecessor àquele, do qual também é obtida a posição do espaço em branco, a qual é concatenada à frente da posição do nodo anterior. Este processo se repete até que o nodo com o estado inicial seja alcançado. Em seguida, a solução é transmitida para o aplicativo cliente e a busca é encerrada naquele servidor.

```
int Busca::mi_iConstroiResposta()
{
	// limpa o atributo de resposta
	ai_sSolucao = wxT("");
	// captura o último nodo processado na busca
	// que é o estado final
	Nodo *oNodoAtual = ai_oBuscaAEstrela->getUltimoNodo();
	// repete enquando o pai do nodo não for nulo
	while (oNodoAtual->getPai() != NULL) {
		// concatena a posição do espaço em branco
		// a interpretação da ordem de mudança do
		// espaço gerará o caminho para a resposta
		ai_sSolucao = wxString::Format(wxT("%d"), oNodoAtual-
>getEspaco()) + ai_sSolucao;
		// pega o pai do nodo como próximo nodo a ser testado
		oNodoAtual = oNodoAtual->getPai();
	}

	// o retorno é a quantidade de passos
	// entre o estado final e inicial
	return ai_sSolucao.size();
}
```

Quadro 15 - Função de concatenação dos passos da solução

Já no aplicativo cliente, para que o usuário possa visualizar a solução do problema, a mensagem enviada pelo aplicativo servidor é interpretada de forma que o primeiro índice da mensagem corresponda ao primeiro passo da solução, e considerando que a aplicação conhece o estado inicial, basta movimentar o espaço em branco de acordo com os índices fornecidos na mensagem, que ter-se-á a seqüência completa de passos entre o estado inicial e final do problema.

O controle dos passos é feito por funções que permitem ao usuário avançar ou retroceder na solução. Estas funções simplesmente trocam as posições do espaço em branco, internamente representado pelo número zero e visualmente como um espaço em branco, de

acordo com o passo atual que o usuário está visualizando. A implementação da função que gera o próximo passo de uma determinada solução é apresentada no Quadro 16. Um terceiro método, *selecionaSolucao()*, obtém o estado gerado e o apresenta visualmente ao usuário.

```
void Solucao::mu_bPassoProximo()
{
      // incrementa o índice
      ai_iIndiceAtual++;

      // obtem a posição onde o espaço
      // em branco deve ficar
      int iPosicao = atoi(ai_sSolucao.substr(ai_iIndiceAtual, 1));

      // coloca a ficha da posição incrementada
      // no lugar do espaço em branco
      ai_sEstadoAtual = ai_sEstadoAtual.substr(0, ai_iPosicaoEspaco) +
ai_sEstadoAtual.substr(iPosicao, 1) +
ai_sEstadoAtual.substr(ai_iPosicaoEspaco + 1);
      // coloca o espaço em branco na posição anterior
    ai_sEstadoAtual = ai_sEstadoAtual.substr(0, iPosicao) + wxT("0") +
ai_sEstadoAtual.substr(iPosicao + 1);

      // salva a nova posição do espaço em branco
      ai_iPosicaoEspaco = iPosicao;
}
```

Quadro 16 - Geração dos passos de uma solução para o usuário

Outro detalhe importante diz respeito ao controle do término da execução das *threads* de busca, como é o caso da estrutura de repetição apresentada anteriormente no Quadro 13, onde, ao término da execução do algoritmo de busca, o processo de repetição é interrompido e a *thread* seria encerrada. Porém, o mesmo objeto que constitui a *thread* é o responsável, no caso do aplicativo servidor, pela montagem e envio da solução, caso seja encontrada. Desta forma, é necessário que o processo de repetição da *thread* seja encerrado, mas sem que o objeto seja destruído. Sendo assim, implementou-se após o código apresentado no Quadro 13, a estrutura representada pelo Quadro 17.

```
      // impede o término da execução da thread caso
      // apenas a busca tenha encerrado
      while (!TestDestroy()) {
            wxThread::Sleep(100);
      }
```

Quadro 17 - Controle do término de uma *thread*

A função interna *TestDestroy()* somente retornará verdadeiro quando o objeto da *thread* estiver sendo destruído ou quando a aplicação estiver sendo encerrada. Com o uso

desta função, o objeto continua executando os procedimentos necessários, e somente quando concluído, é destruído.

3.3.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Este tópico aborda o teste de operacionalidade da implementação, apresentado através de um estudo de caso que compreende a resolução dos seguintes estados do problema *8-Puzzle*, onde a seqüência dos elementos segue da esquerda para a direita, de cima para baixo, e o numeral zero representa o espaço vazio:

a) estado inicial: 1, 3, 4, 8, 0, 2, 7, 6, 5; e estado final: 1, 2, 3, 8, 0, 4, 7, 6, 5. Sendo que a solução ótima conhecida para este problema possui quatro passos;
b) estado inicial: 2, 8, 1, 4, 6, 3, 0, 7, 5; e estado final: 1, 2, 3, 8, 0, 4, 7, 6, 5. Sendo que a solução ótima conhecida para este problema possui doze passos;
c) estado inicial: 2, 1, 6, 4, 0, 8, 7, 5, 3; e estado final: 1, 2, 3, 8, 0, 4, 7, 6, 5. Sendo que a solução ótima conhecida para este problema possui dezoito passos;
d) estado inicial:5, 6, 7, 4, 0, 8, 3, 2, 1; e estado final: 1, 2, 3, 8, 0, 4, 7, 6, 5. Sendo que a solução ótima conhecida para este problema possui trinta passos.

Inicialmente, para que possa existir a comunicação entre os aplicativos cliente e servidor, é necessário que o servidor, ou *daemon*, do mecanismo de comunicação em grupo *Spread* esteja devidamente configurado e esteja em execução, como mostra a Figura 18.

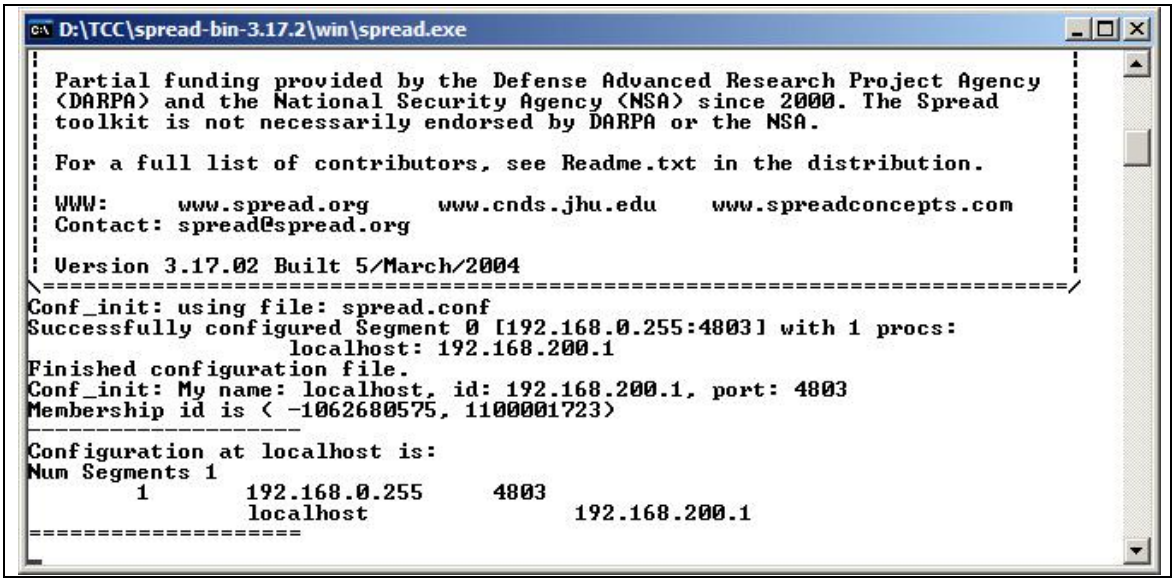


Figura 18 - Interface do servidor *Spread* na plataforma *Windows*

A Figura 18 apresenta a interface do servidor *Spread* em execução na plataforma *Windows*, onde são exibidos os dados referentes à configuração atual do servidor.

Quanto ao aplicativo cliente, no momento em que é iniciado, este contém apenas a guia *Monitor*, que exibe todas as informações relevantes sobre os procedimentos e eventos que estão ocorrendo. A interface inicial do aplicativo cliente executando na plataforma *Linux* é apresentada pela Figura 19, onde estão sendo gerenciados 26 servidores.

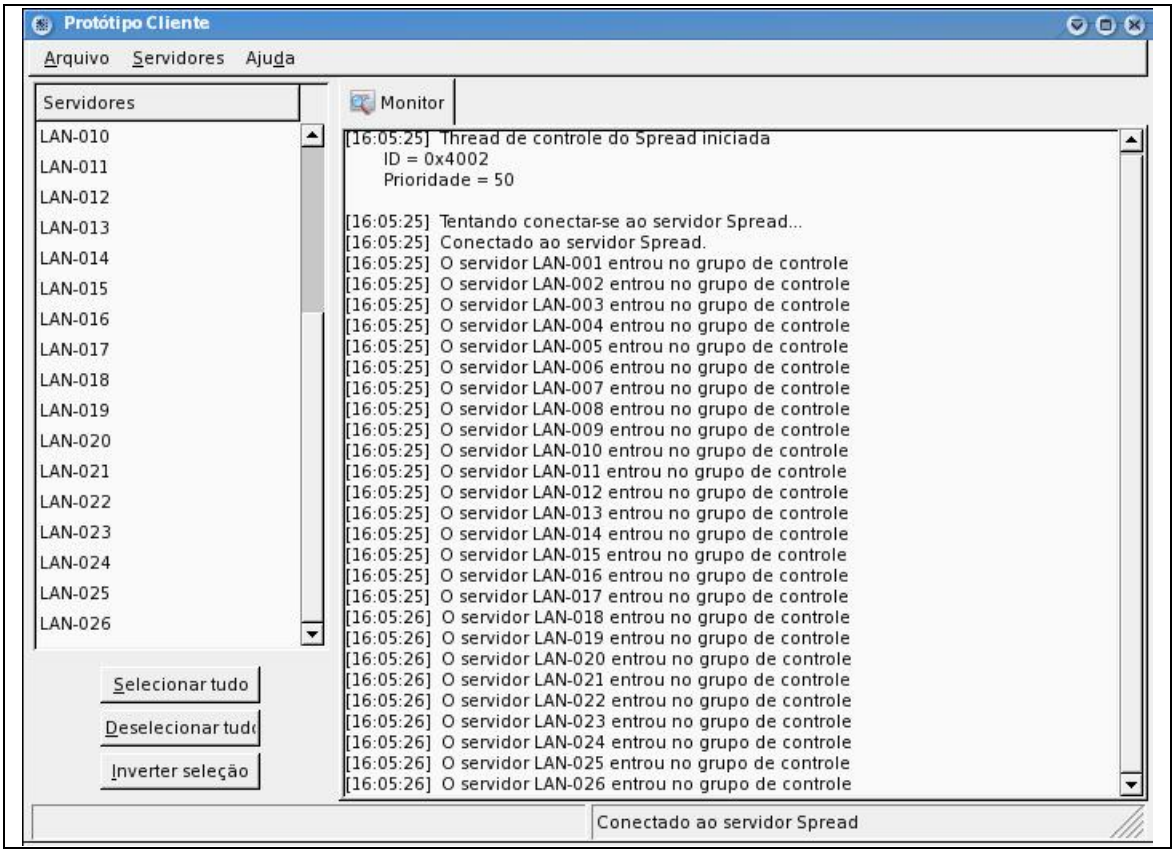


Figura 19 - Interface do aplicativo cliente na plataforma *Linux*

As informações quanto aos procedimentos e eventos que estão ocorrendo também estão disponíveis no aplicativo servidor, porém, na forma de um controle que ocupa toda a área do aplicativo. A interface inicial do aplicativo servidor executando na plataforma *Linux* é ilustrada pela Figura 20.

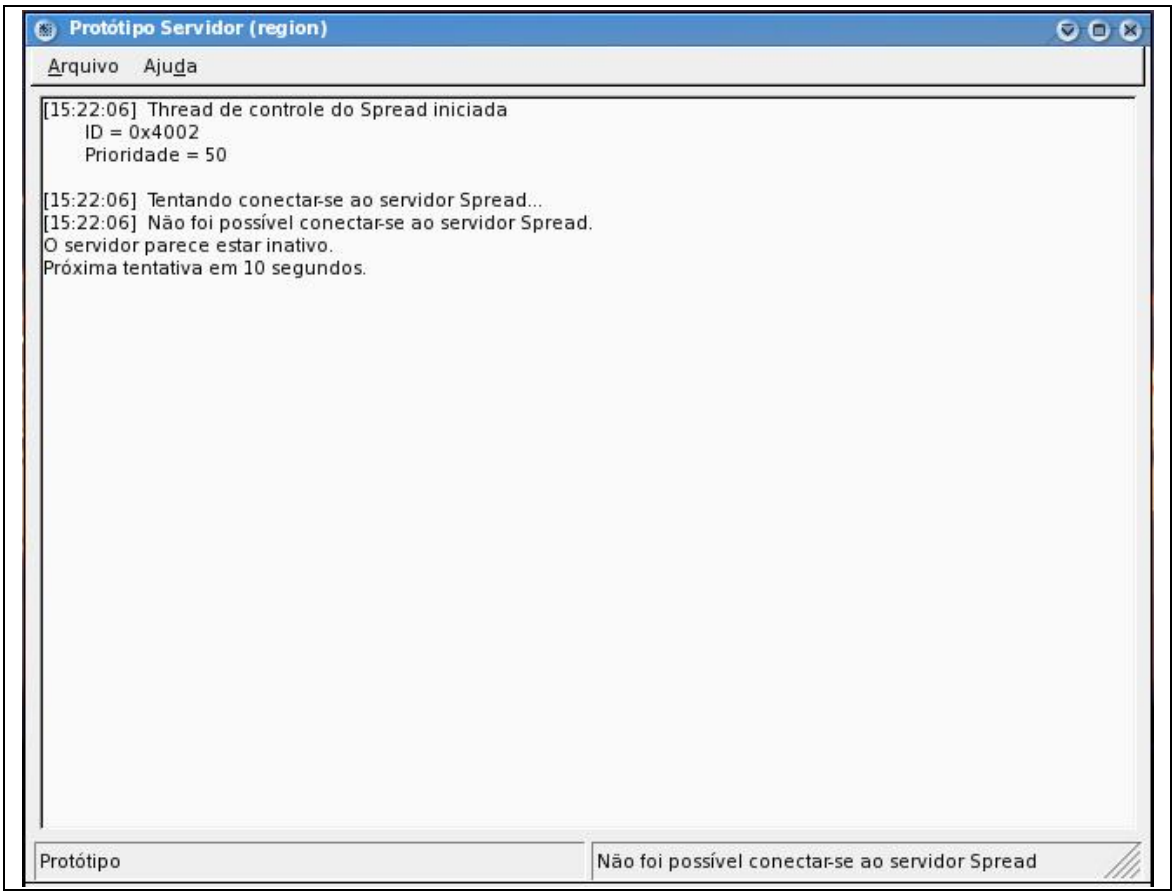


Figura 20 - Interface do aplicativo servidor na plataforma *Linux*

A operação do sistema por parte do usuário se dá totalmente através do aplicat}ivo cliente, o qual irá controlar a busca. Para criar-se um novo processo de busca, deve-se ativar a opção *Nova busca* no menu *Arquivo* do aplicativo cliente. Abrir-se-á uma nova guia contendo propriedades do novo processo de busca, como ilustrado pela Figura 21, onde o aplicativo cliente está sendo executado na plataforma *Windows*.

Na guia do novo processo de busca, apresentam-se inicialmente os estados inicial e final do problema que deseja-se solucionar. Para modificar a seqüência dos elementos de um dos estados, basta um duplo clique sobre o elemento que se quer mover, e em seguida um clique simples sobre o elemento que ocupa a posição de destino.

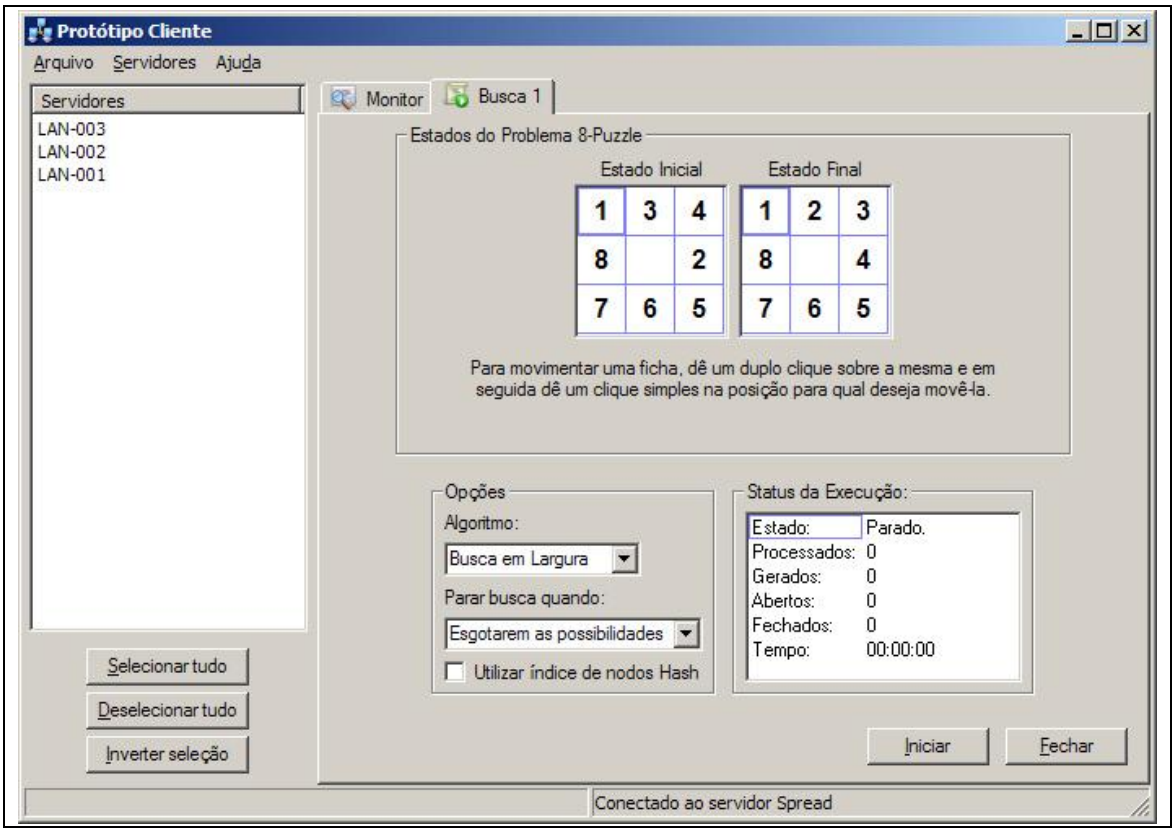


Figura 21 - Guia de um novo processo de busca na plataforma *Windows*

Supondo, por exemplo, que queira-se mover o elemento 3 do estado inicial para o centro do tabuleiro, no lugar do espaço vazio, dá-se um duplo clique sobre o elemento 3 e em seguida um clique simples sobre o espaço vazio. Os elementos irão trocar de lugar, como ilustrado pela Figura 22.

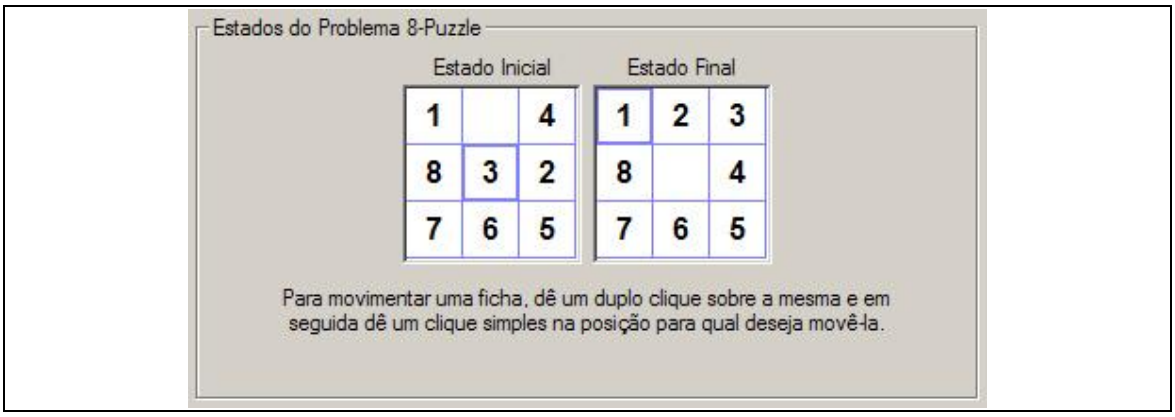


Figura 22 - Configuração dos elementos de um estado

As outras opções disponíveis tratam quanto ao algoritmo de busca que se quer utilizar, *Busca em Largura* ou *Busca Heurística A**, e quanto ao método de parada da busca, onde a

opção *Esgotarem as possibilidades* faz com que o processo de busca seja encerrado somente quando todos os servidores retornarem alguma resposta, seja esta a solução ou não, e a opção *Encontrar a primeira solução*, que fará com que o processo de busca seja encerrado assim que a primeira solução for encontrada. A última opção, *Utilizar índice de nodos Hash*, quando habilitada, utiliza no processo de busca uma tabela *hash* como índice para os nodos nas listas de nodos abertos e fechados, o que aumenta o consumo de memória, mas acelera consideravelmente o processo de busca.

Inicialmente, para os estados descritos no item *a* do estudo de caso, será utilizado o algoritmo de busca em largura, com a opção de parada quando esgotar as possibilidades, e com o índice *hash* desativado, conforme a Figura 23. Na lista à esquerda, encontram-se os servidores disponíveis. É através da seleção dos itens desta lista que define-se os servidores que farão parte de cada processo de busca. Inicialmente será utilizado apenas um servidor, no caso, *LAN-001*.

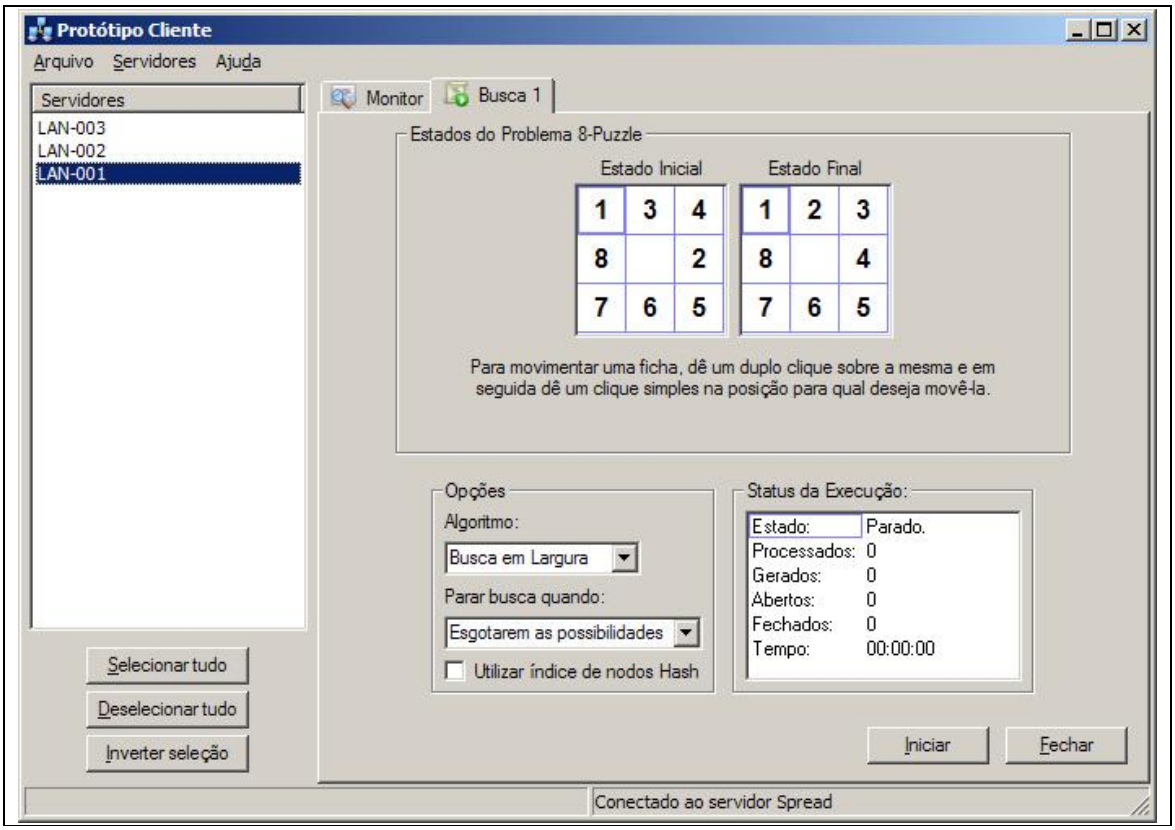


Figura 23 - Configuração do processo de busca

Em seguida, pressiona-se o botão *Iniciar*, o que fará com que o processo de busca inicie. O aplicativo servidor no computador *LAN-001* irá receber os dados e começará um novo processo de busca, conforme a Figura 24.

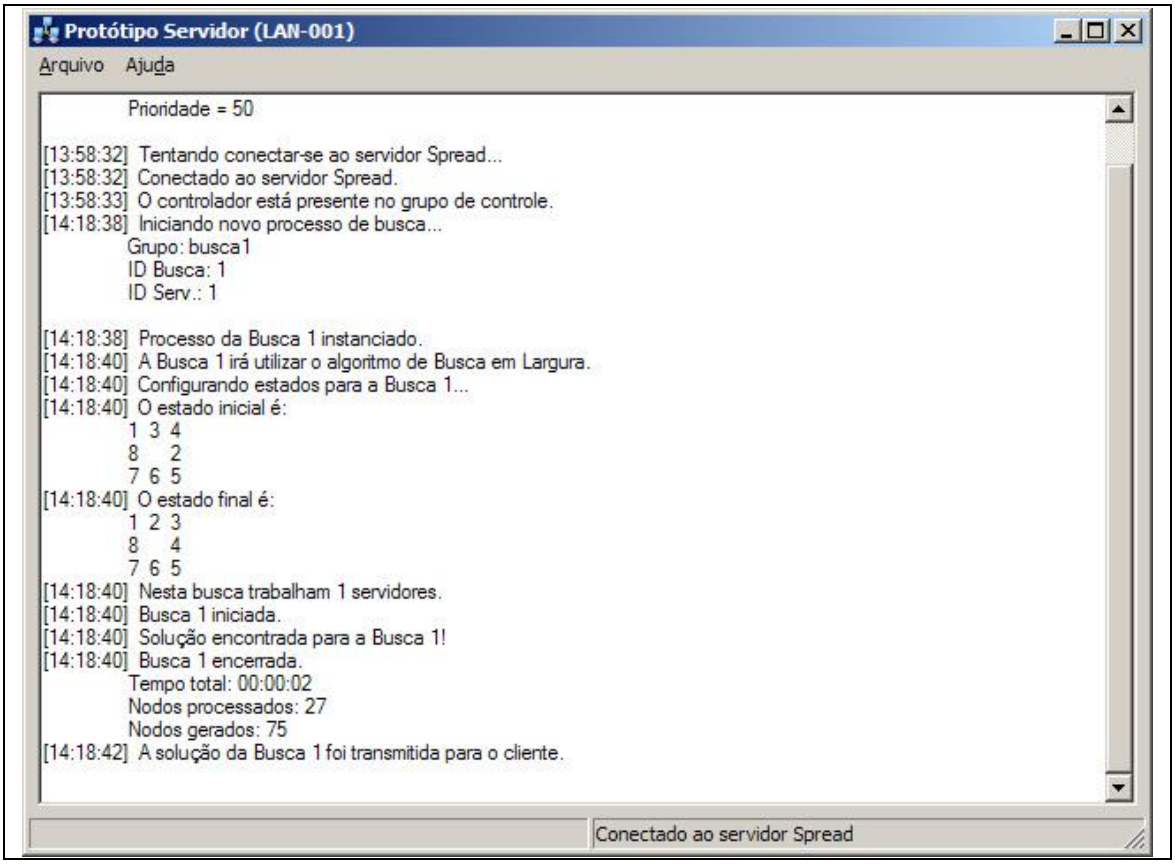


Figura 24 - Aplicativo servidor processando uma nova busca

Considerando que os estados do item *a* do estudo de caso geram uma solução de quatro passos, a solução será rapidamente encontrada e transmitida pelo aplicativo servidor, e apresentada no aplicativo cliente, como mostra a Figura 25.

Assim que uma solução é encontrada, a tela do aplicativo cliente modifica-se. O quadro com o estado inicial dá lugar à lista de soluções encontradas, e os botões *Anterior* e *Próximo* aparecem, os quais permitem ao usuário avançar ou retornar os passos de uma solução.

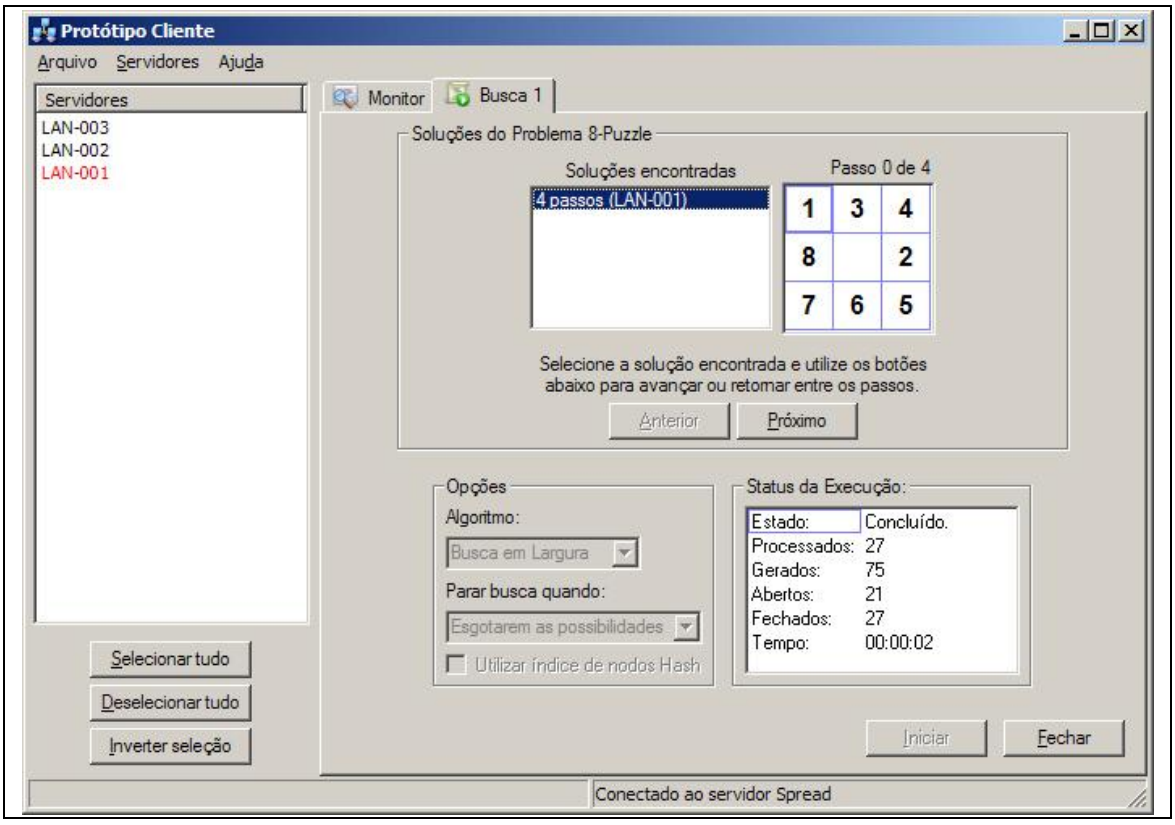


Figura 25 - Apresentação da solução no aplicativo cliente

Inicialmente, o passo apresentado é o zero, que corresponde ao passo inicial da busca. Pressionando o botão *Próximo*, será apresentado o passo seguinte, ou seja, passo 1, com a primeira mudança na posição dos elementos. Desta forma, o usuário pode verificar cada um dos passos, desde o inicial, até o final, conforme a Figura 26.

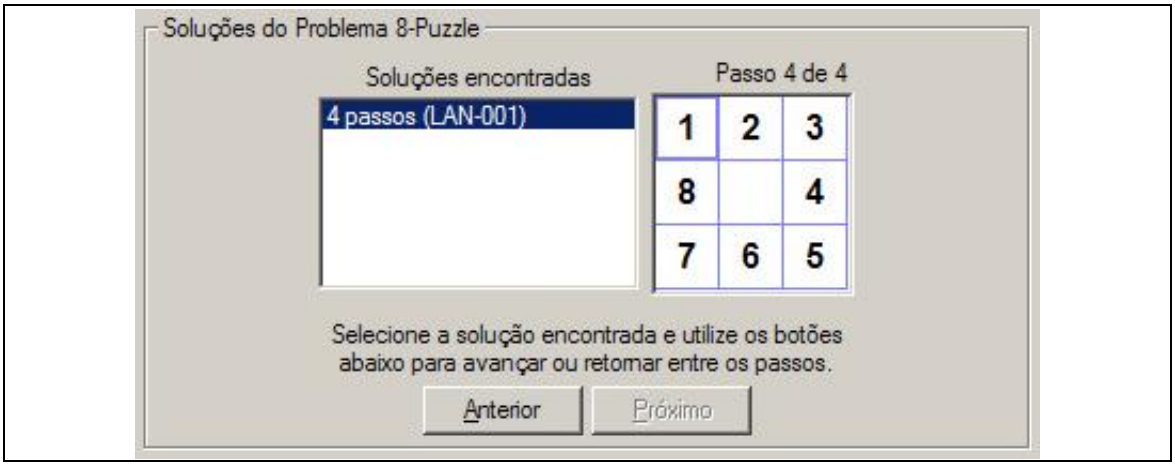


Figura 26 - Passos da solução de um processo de busca

Os mesmos estados indicados pelo item *a* do estudo de caso foram utilizados em uma nova busca com as mesmas configurações citadas anteriormente, porém, com três servidores, conforme a Figura 27, onde então, cada servidor encontrou uma solução, das quais uma é a solução ótima, com quatro passos, e as duas soluções restantes possuem seis passos cada.

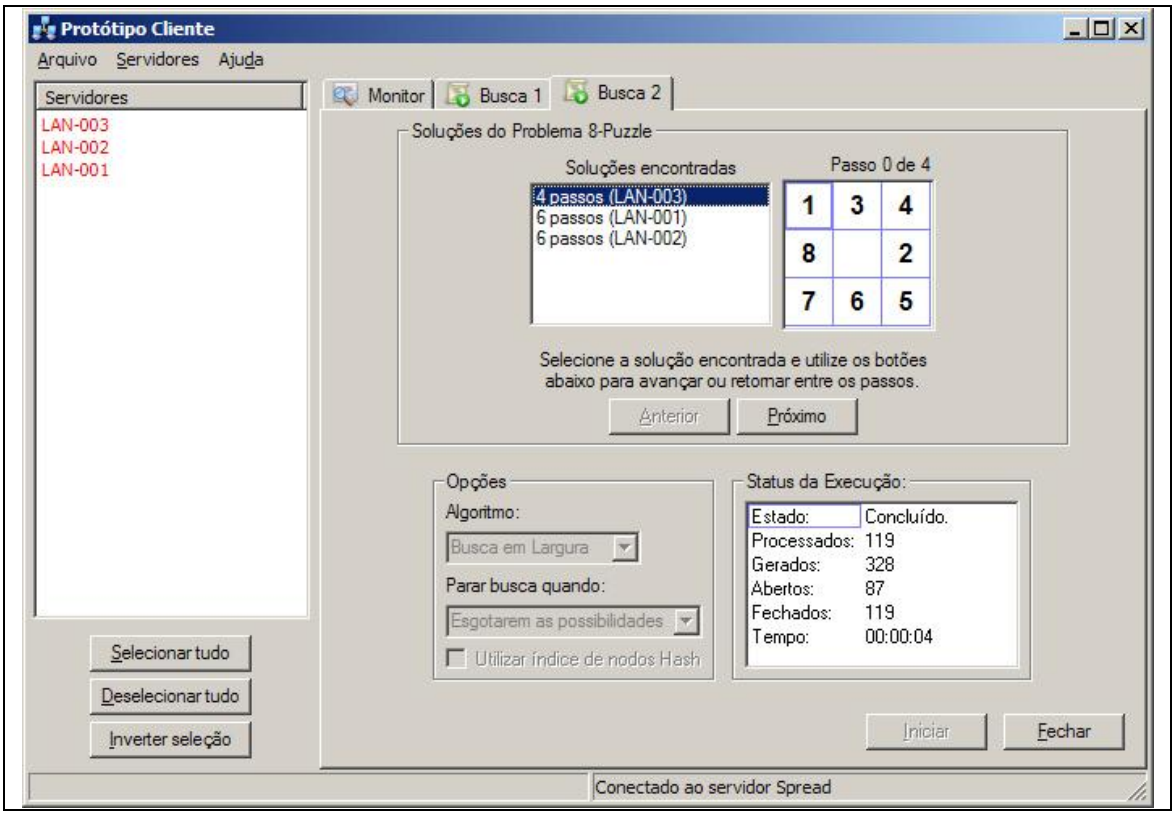


Figura 27 - Soluções de um processo de busca

Quanto aos estados descritos no item *b* do estudo de caso, foram estes configurados em um novo processo de busca utilizando a opção *Encontrar a primeira solução* como método de parada da busca, e fazendo-se uso de apenas um servidor. A solução é apresentada pela Figura 28.

Um segundo processo de busca com as mesmas opções da busca anterior, porém, utilizando três servidores, tem sua solução apresentada pela Figura 29. Os dois processos de busca, tanto o apresentado pela Figura 28 quanto pela Figura 29, encontraram a solução ótima, com doze passos.

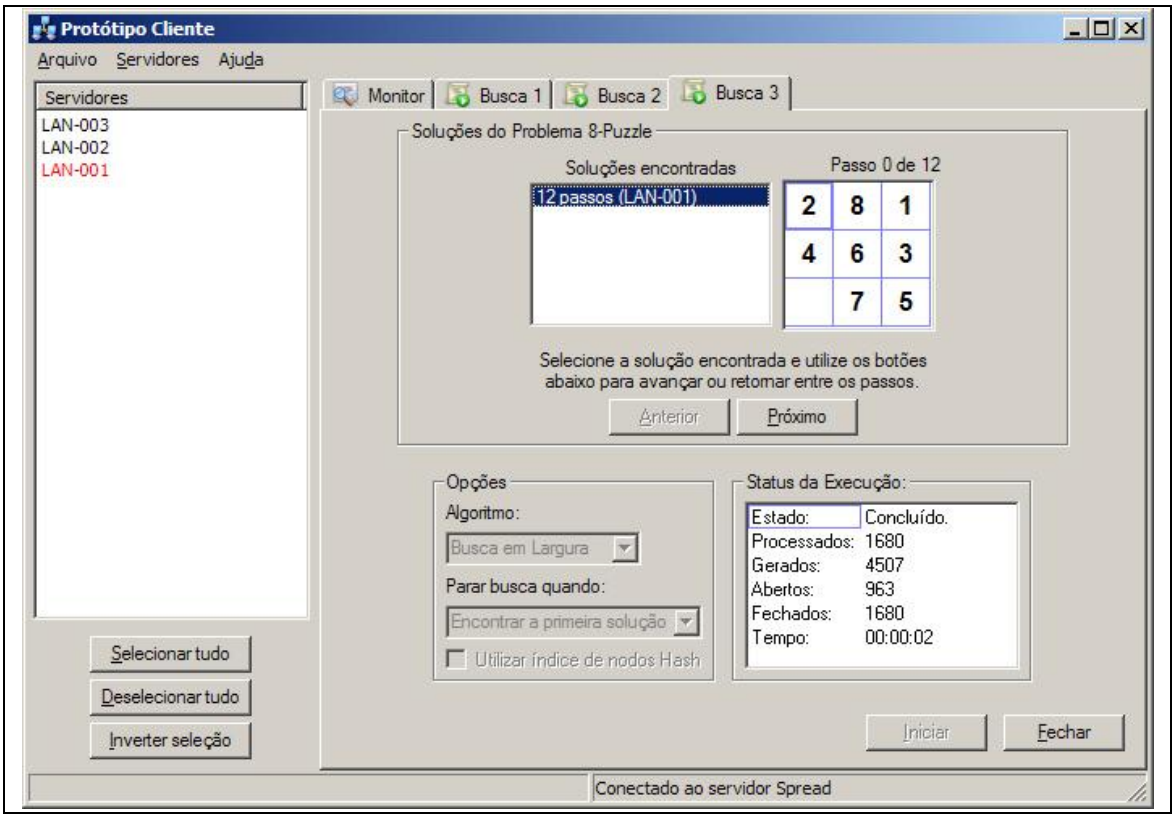


Figura 28 - Solução do processo de busca utilizando-se um servidor

Para os estados propostos no item *c*, foram efetuados oito processos de busca, alternando as opções de algoritmos de busca e índice de nodos *hash*, inicialmente com um servidor e em seguida com três servidores. Em todos os processos de busca a solução com dezoito passos foi encontrada.

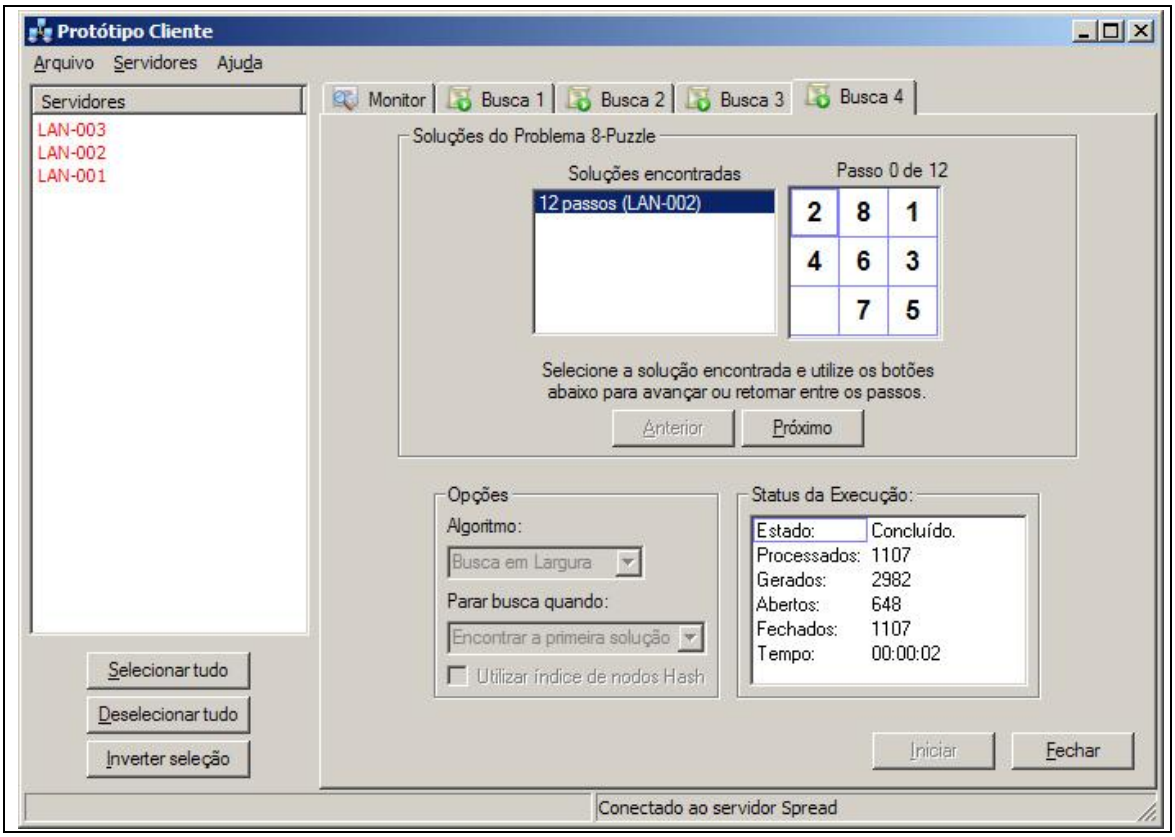


Figura 29 - Solução do processo de busca utilizando-se três servidores

No caso dos estados do item *d*, foram utilizados seis servidores, também variando as opções de algoritmo e índice de nodos *hash*. As buscas utilizando índice *hash* com ambos os algoritmos encontraram a solução, enquanto que a busca utilizando a opção *Busca em Largura* com o índice *hash* desativado foi cancelada quando atingiu o tempo de 12 minutos, não retornando solução.

3.4 RESULTADOS E DISCUSSÃO

De acordo com os testes realizados para o estudo de caso apresentado, obteve-se bons resultados em todos os aspectos do protótipo e até mesmo superiores ao esperado, como no caso da utilização do mecanismo de indexação de nodos *hash* para as listas de nodos abertos e fechados. Tomou-se os resultados das buscas pelos estados descritos no item *c* do estudo de caso e com estes elaborou-se um comparativo detalhado, apresentado na Tabela 4.

Tabela 4 - Comparativo de resultados

Busca	Servidores	Algoritmo	Índice *hash*	Nodos		Concluído	Tempo (m:ss)
				Gerados	Processados		
1	1	Largura	Não	64712	23642	Sim	3:22
2	1	Largura	Sim	64712	23642	Sim	0:19
3	1	A*	Não	49995	18409	Sim	2:15
4	1	A*	Sim	49995	18409	Sim	0:17
5	3	Largura	Não	162483	60843	Sim	1:59
6	3	Largura	Sim	183640	67762	Sim	0:16
7	3	A*	Não	96793	35576	Sim	0:41
8	3	A*	Sim	119065	43863	Sim	0:10

Comparando os resultados das buscas de 1 à 4 com as buscas 5 à 8, percebe-se uma diferença de tempo nos resultados que comprova a eficiência da distribuição do processo de busca. O algoritmo de busca em largura sem a utilização de índice ***hash*** obteve uma diferença de 1 minuto e 23 segundos à menos quando distribuído. A diferença para o algoritmo de busca heurística A* na mesma situação foi de 1 minuto e 34 segundos.

Observando-se os resultados das buscas 1 e 3, nota-se uma diferença de tempo de 1 minuto e 7 segundos que comprova o melhor desempenho do algoritmo de busca heurística A* em relação ao algoritmo de busca em largura. O mesmo pode ser observado nos resultados das buscas 2 e 4, porém com uma diferença de 2 segundos.

A diferença de desempenho entre os algoritmos de busca também pode ser notada na execução do processo de busca proposto pelo item ***d*** do estudo de caso onde, utilizando o índice de nodos ***hash***, o algoritmo de busca heurística encontrou a solução no tempo de 2 minutos e 17 segundos, enquanto que o algoritmo de busca em largura ocupou o tempo de 3 minutos e 6 segundos para encontrar a mesma solução.

A utilização do mecanismo de comunicação em grupo ***Spread*** mostrou-se vantajosa visto que toda a complexidade da comunicação foi abstraída dos aplicativos cliente e servidor. O mecanismo manteve-se estável e com bom desempenho durante todos os testes e em nenhum momento foram relatados problemas na troca de mensagens.

Quanto à utilização da opção de índice de nodos ***hash***, os resultados obtidos foram bons. A Figura 30 apresenta um gráfico comparativo ilustrando a quantidade de nodos processados em relação ao tempo. Os dados foram obtidos do ***log*** de execução do aplicativo

servidor em um dos testes, com amostragens em intervalos de 5 segundos. O *log* encontra-se no Anexo A.

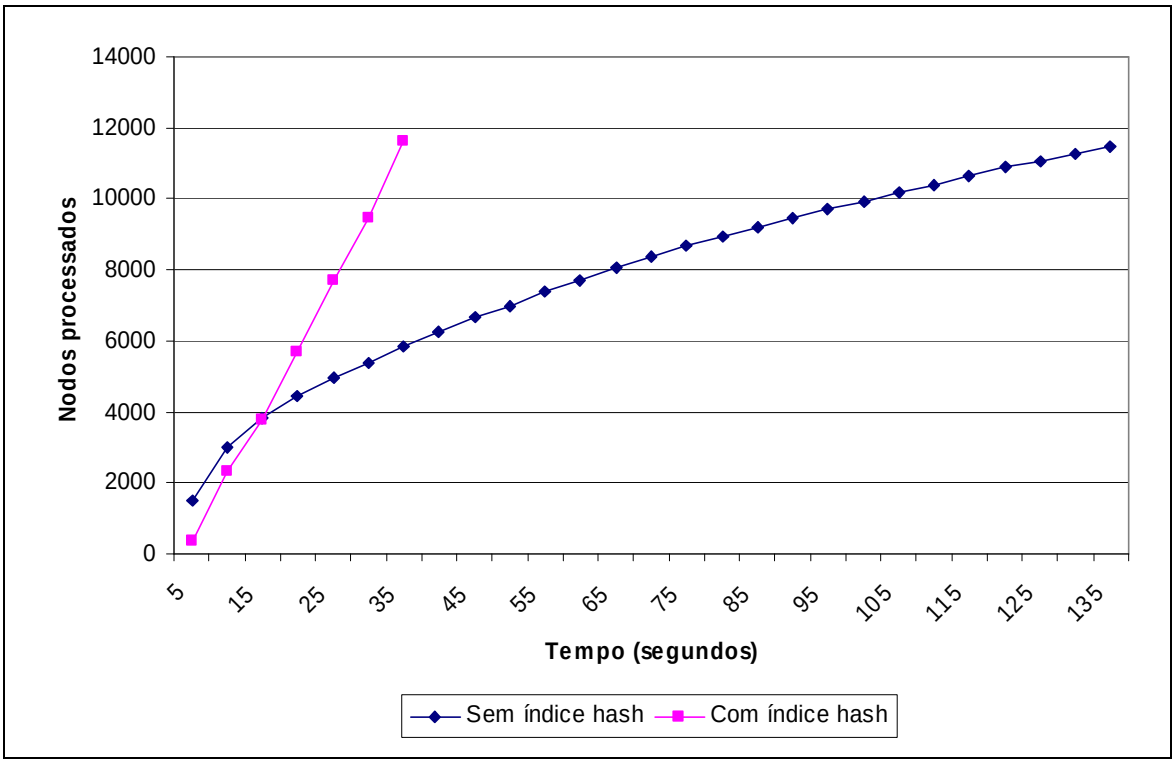


Figura 30 - Gráfico comparativo de nodos processados em relação ao tempo

Através da Figura 30 é possível notar a diferença na utilização do índice de nodos *hash*. O processo de busca com esta opção levou pouco tempo para processar a quantidade de nodos necessária para se encontrar a solução, enquanto que a busca sem esta opção teve uma evolução mais lenta.

A partir do mesmo *log* de execução, foi obtida a diferença na quantidade de nodos processados em intervalos de 5 segundos. Esta diferença pode ser vista pelo gráfico apresentado na Figura 31. Sem o uso do índice *hash*, a quantidade de nodos processados decai em relação ao tempo, visto que as listas de nodos abertos e fechados aumentam e é necessário mais tempo para fazer a varredura linear nestas, enquanto que a quantidade de nodos processados utilizando-se o índice *hash* mantém-se quase estável, com valores superiores, o que conseqüentemente faz com que a busca seja concluída em menor tempo.

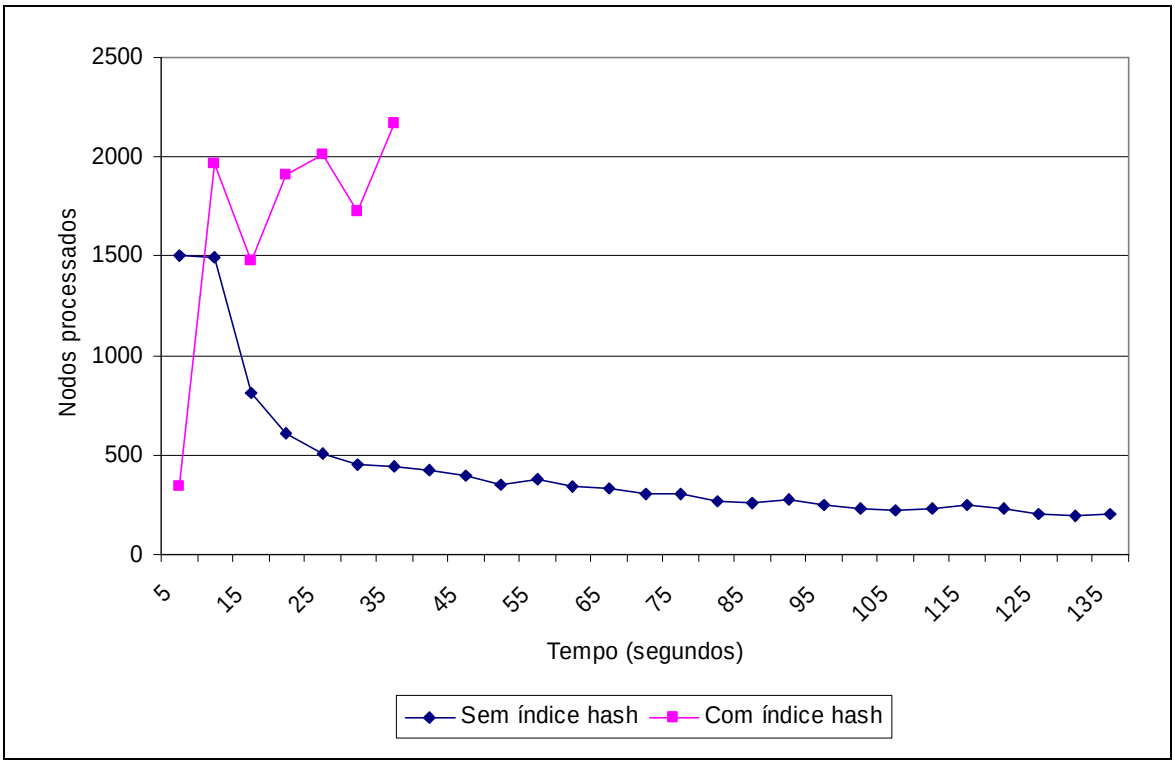


Figura 31 - Quantidade de nodos processados em intervalos de 5 segundos

Em relação ao uso da biblioteca *wxWidgets*, os aplicativos mostraram-se estáveis e com bom desempenho em todos os testes realizados na plataforma *Windows*. Porém, notou-se certa instabilidade na plataforma *Linux*, onde tanto o aplicativo cliente quanto servidor eram abortados apresentando erros na manipulação de janelas relacionadas ao *GIMP Toolkit* (GTK+).

Quanto aos trabalhos correlatos, visto que não foram encontrados quaisquer com propósito similar a este, comparou-se apenas no que diz respeito ao uso do mecanismo de comunicação em grupo *Spread*. E sendo assim, constatou-se que a forma de utilização do *Spread* é a mesma, desde a utilização da API até a forma como as mensagens são tratadas.

Tentou-se comparar o desempenho da comunicação com o mecanismo *Spread* em relação ao tempo, mas a documentação de nenhum dos trabalhos correlatos apresentava de forma específica valores que pudessem ser utilizados para tal finalidade.

Os aplicativos que também fazem uso da biblioteca *wxWidgets* se mostram mais estáveis do que o protótipo desenvolvido quando executados na plataforma *Linux*. Na plataforma *Windows*, o desempenho e estabilidade foram similares.

4 CONCLUSÕES

Ao término do desenvolvimento do trabalho pode-se concluir que os resultados alcançados foram bons em relação aos objetivos previamente formulados, os quais foram atendidos. A distribuição dos processos de busca mostrou-se uma técnica vantajosa e eficiente, tanto para o algoritmo de busca em largura quanto para o algoritmo de busca heurística A*, em um ambiente de rede LAN.

Porém, a maior expectativa e satisfação veio com as idéias incorporadas ao trabalho após a elaboração dos objetivos, como é o caso do desenvolvimento dos aplicativos cliente e servidor para múltiplas plataformas e não apenas *Windows* como planejado inicialmente, e também pela utilização do mecanismo de indexação de nodos através de tabelas *hash*, o qual obteve desempenho superior ao esperado.

As ferramentas utilizadas foram adequadas ao desenvolvimento do trabalho, desde as tecnologias, como o mecanismo *Spread* e a biblioteca *wxWidgets*, até o ambiente de desenvolvimento. Os recursos de depuração oferecidos pelo *Microsoft Visual C++* foram essenciais e de fácil uso, visto que são amigáveis, assim como o restante do ambiente.

De modo geral, este trabalho contribuiu para o desenvolvimento científico, visto que é composto por uma aplicação distribuída para a manipulação de estruturas de dados, fazendo uso de tecnologias emergentes de desenvolvimento multiplataforma, incorporando um mecanismo de comunicação também multiplataforma, e que por fim, utiliza o paradigma da orientação a objeto.

Quanto às limitações do trabalho, a maior é a capacidade de encontrar soluções apenas para os estados do problema *8-Puzzle*, seguida da falta de um mecanismo de tolerância e recuperação de falhas, o que provoca o cancelamento da busca caso um dos processos envolvidos pare de funcionar.

4.1 EXTENSÕES

De acordo com os conhecimentos adquiridos durante o desenvolvimento do trabalho, tornou-se possível a análise de diversas possibilidades que podem ser sugeridas como trabalhos futuros, sendo elas:

a) desenvolvimento de um aplicativo cliente e servidor, com as mesmas características, porém na linguagem *Java*, na forma de *applet*;
b) aperfeiçoar a estrutura de gerenciamento de servidores de forma que seja possível a execução da aplicação em ambientes de rede WAN;
c) desenvolvimento de um mecanismo de tolerância e recuperação de falhas que permita prosseguir com o processo de busca mesmo em caso de particionamento da rede, armazenando a solução, caso seja encontrada, para posterior verificação;
d) desenvolvimento de um mecanismo que permita comparar as soluções encontradas, descartando soluções iguais e apresentando somente as melhores;
e) aprimorar a forma de distribuição da busca entre os servidores;
f) possibilidade de trabalhar com outras estruturas de dados, implementando algoritmos de busca diferentes, estendendo a aplicabilidade para outros problemas do campo da inteligência artificial, e até mesmo de outras áreas, como matemática, química ou biologia, através de algoritmos de análise genética.

REFERÊNCIAS BIBLIOGRÁFICAS

AMIR, Yair; STANTON, Jonathan. **The spread wide area group communication system**, Baltimore, 1998. Disponível em: <http://www.cnds.jhu.edu/publications/ >. Acesso em: 8 set. 2004.

AOL. **Aol communicator**, Dulles, 2003. Disponível em: <http://www.aolepk.com/communicator/index.html>. Acesso em 12 nov. 2004.

AUDACITY. **Audacity**, [s.l.], [2004]. Disponível em: <http://audacity.sourceforge.net/>. Acesso em 12 nov. 2004.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed systems**: concepts and design. Harlow: Addison-Wesley, 2001.

FURTADO, Camille; HEUSELER, Fábio; LYCHOWSKY, Rafael. **Wxwindows**: informações gerais, Rio de Janeiro, [2003]. Disponível em: <http://www.dcc.ufrj.br/~schneide/psi/2003/2/gp09/>. Acesso em: 6 ago. 2004.

GIORDANO, Liliane; JANSCH-PÔRTO, Ingrid E. S. **Roteamento dinâmico para melhorar resultados decorrentes da imprecisão dos detectores de defeitos**, Porto Alegre, [2000]. Disponível em: <http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana2000/Liliane/>. Acesso em: 7 set. 2004.

GOULART, Ademir. **Avaliação de mecanismos de comunicação em grupo para ambientes wan.** 2002. 169 f. Dissertação (Mestrado em Ciências da Computação) - Universidade Federal de Santa Catarina, Florianópolis.

GOULART, Ademir; FRIEDRICH, Luis F. **Gerenciamento de recursos em sistemas distribuídos usando comunicação em grupo**, Itajaí, 2002. Disponível em: <http://www.cbcomp.univali.br/anais/anais.htm>. Acesso em: 7 set. 2004.

GRID. **United devices cancer research project**, [s.l.], 2004. Disponível em: <http://www.grid.org/projects/cancer/>. Acesso em: 29 mar. 2004.

GRISOFT. **Avg anti-virus**, Lidicka, 2004. Disponível em: <http://www.grisoft.com>. Acesso em 12 nov. 2004.

GUITOOLKIT. **The gui toolkit framework page**, [s.l.], 2003. Disponível em: <http://www.atai.org/guitool/>. Acesso em 21 ago. 2004.

HÜBNER, Jomi Fred. **Busca em espaço de estados**, Blumenau, 2004. Disponível em: <http://www.inf.furb.br/~jomi/ia/slides/busca.pdf>. Acesso em: 16 nov. 2004.

JBONLINE. **FGV:** Brasil terá 22,4 milhões de computadores em 2003, Rio de Janeiro, 2003. Disponível em: <http://jbonline.terra.com.br/jb/online/internet/noticias/2003/03/21/onlintnot20030321001.html>. Acesso em: 29 mar. 2004.

JONES, Justin Heyes. **Justin Heyes Jones a* tutorial**, Cheshire, 2004. Disponível em: <http://www.geocities.com/jheyesjones/astar.html>. Acesso em: 16 nov. 2004.

LESTER, Patrick. **A* pathfinding for beginners**, [s.l.], 2004. Disponível em: <http://www.policyalmanac.org/games/aStarTutorial.htm>. Acesso em: 16 nov. 2004.

MACÊDO, José A. R. **Causal order protocols for group communication**, Salvador, 1995. Disponível em: <http://www.lasid.ufba.br/public/publicacoes.html>. Acesso em: 8 set. 2004.

MODLOG. **Mod_log_spread:** a tool for distributed auditing and monitoring, [s.l.], 2000. Disponível em: <http://www.lethargy.org/mod_log_spread/>. Acesso em: 11 ago. 2004.

RUSSEL, Stuart J.; NORVIG, Peter. **Artificial intelligence**: a modern approach. Nova Jersey: Prentice Hall, 1995.

SANTOS, Gilliard L. **Inteligência artificial em jogos 3d**: uma estratégia de busca de caminhos no espaço dividido em volumes convexos. 2002. 39 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Departamento de Ciência da Computação, Universidade Federal Fluminense, Niterói.

SCHNEEBELI, Hans J. A. **Usando wxwindows para programação windows**, Vitória, [2000]. Disponível em: <http://www2.ele.ufes.br/~hans/wxwin2/>. Acesso em: 6 ago. 2004.

SETI@HOME. **Seti@home:** search for extraterrestrial intelligence at home, [s.l.], 2003. Disponível em: <http://setiathome.ssl.berkeley.edu/>. Acesso em: 29 mar. 2004.

SPREAD. **Spread overview**, Baltimore, 2004. Disponível em: <http://www.spread.org/docs/spreadoverview.html>. Acesso em: 10 ago. 2004.

STANTON, Jonathan R. **A users guide to spread**, [s.l.], 2002. Disponível em: <http://www.spread.org/docs/guide/users_guide.pdf>. Acesso em: 7 set. 2004.

TANENBAUM, Andrew S.; STEEN, Maarten V. **Distributed systems**: principles and paradigms. Nova Jersey: Prentice Hall, 2002.

ZAWODNY, Jeremy. Do it yourself: the spread toolkit. **Linux Magazine**, San Francisco, abr. 2003. Disponível em: <http://www.linux-mag.com/2003-04/diy_01.html>. Acesso em: 10 ago. 2004.

ANEXO A – Log de execução do aplicativo servidor

```
[09:20:01]  Iniciando novo processo de busca...
      Grupo: busca3
      ID Busca: 3
      ID Serv.: 1

[09:20:01]  Processo da Busca 3 instanciado.
[09:20:05]  A Busca 3 irá utilizar o algoritmo de Busca Heurística A*.
[09:20:05]  Configurando estados para a Busca 3...
[09:20:05]  O estado inicial é:
      2  1  6
      4     8
      7  5  3
[09:20:05]  O estado final é:
      1  2  3
      8     4
      7  6  5
[09:20:05]  Nesta busca trabalham 3 servidores.
[09:20:05]  Busca 3 iniciada.
[09:20:06]  Estatística da Busca 3: 1506 nodos processados de 4089 gerados.
Agora: 904 abertos e 1505 fechados.
[09:20:11]  Estatística da Busca 3: 3003 nodos processados de 8150 gerados.
Agora: 1770 abertos e 3002 fechados.
[09:20:16]  Estatística da Busca 3: 3814 nodos processados de 10328
gerados. Agora: 2200 abertos e 3813 fechados.
[09:20:21]  Estatística da Busca 3: 4427 nodos processados de 12034
gerados. Agora: 2570 abertos e 4426 fechados.
[09:20:26]  Estatística da Busca 3: 4937 nodos processados de 13483
gerados. Agora: 2906 abertos e 4936 fechados.
[09:20:31]  Estatística da Busca 3: 5389 nodos processados de 14772
gerados. Agora: 3210 abertos e 5388 fechados.
[09:20:36]  Estatística da Busca 3: 5835 nodos processados de 15959
gerados. Agora: 3427 abertos e 5834 fechados.
[09:20:41]  Estatística da Busca 3: 6258 nodos processados de 17079
gerados. Agora: 3620 abertos e 6257 fechados.
[09:20:46]  Estatística da Busca 3: 6651 nodos processados de 18116
gerados. Agora: 3795 abertos e 6650 fechados.
[09:20:51]  Estatística da Busca 3: 6999 nodos processados de 19083
gerados. Agora: 3994 abertos e 6998 fechados.
[09:20:56]  Estatística da Busca 3: 7378 nodos processados de 20038
gerados. Agora: 4127 abertos e 7377 fechados.
[09:21:01]  Estatística da Busca 3: 7719 nodos processados de 20941
gerados. Agora: 4280 abertos e 7718 fechados.
[09:21:06]  Estatística da Busca 3: 8055 nodos processados de 21797
gerados. Agora: 4406 abertos e 8054 fechados.
[09:21:11]  Estatística da Busca 3: 8362 nodos processados de 22618
gerados. Agora: 4538 abertos e 8361 fechados.
[09:21:16]  Estatística da Busca 3: 8666 nodos processados de 23414
gerados. Agora: 4669 abertos e 8665 fechados.
[09:21:21]  Estatística da Busca 3: 8935 nodos processados de 24173
gerados. Agora: 4829 abertos e 8934 fechados.
[09:21:26]  Estatística da Busca 3: 9192 nodos processados de 24895
gerados. Agora: 4979 abertos e 9191 fechados.
[09:21:31]  Estatística da Busca 3: 9466 nodos processados de 25614
gerados. Agora: 5089 abertos e 9465 fechados.
[09:21:36]  Estatística da Busca 3: 9711 nodos processados de 26305
gerados. Agora: 5231 abertos e 9710 fechados.
[09:21:41]  Estatística da Busca 3: 9942 nodos processados de 26963
gerados. Agora: 5378 abertos e 9941 fechados.
```

```
[09:21:46]  Estatística da Busca 3: 10165 nodos processados de 27606
gerados. Agora: 5522 abertos e 10164 fechados.
[09:21:51]  Estatística da Busca 3: 10397 nodos processados de 28247
gerados. Agora: 5642 abertos e 10396 fechados.
[09:21:56]  Estatística da Busca 3: 10645 nodos processados de 28890
gerados. Agora: 5719 abertos e 10644 fechados.
[09:22:01]  Estatística da Busca 3: 10879 nodos processados de 29509
gerados. Agora: 5806 abertos e 10878 fechados.
[09:22:06]  Estatística da Busca 3: 11080 nodos processados de 30092
gerados. Agora: 5937 abertos e 11079 fechados.
[09:22:11]  Estatística da Busca 3: 11274 nodos processados de 30669
gerados. Agora: 6083 abertos e 11273 fechados.
[09:22:16]  Estatística da Busca 3: 11475 nodos processados de 31243
gerados. Agora: 6200 abertos e 11474 fechados.
[09:22:21]  Solução encontrada para a Busca 3!
[09:22:21]  Busca 3 encerrada.
      Tempo total: 00:02:20
      Nodos processados: 11646
      Nodos gerados: 31734
[09:22:22]  A solução da Busca 3 foi transmitida para o cliente.
[09:24:03]  Iniciando novo processo de busca...
      Grupo: busca4
      ID Busca: 4
      ID Serv.: 1

[09:24:03]  Processo da Busca 4 instanciado.
[09:24:07]  A Busca 4 irá utilizar índice Hash para os nodos.
[09:24:07]  A Busca 4 irá utilizar o algoritmo de Busca Heurística A*.
[09:24:07]  Configurando estados para a Busca 4...
[09:24:07]  O estado inicial é:
      2  1  6
      4     8
      7  5  3
[09:24:07]  O estado final é:
      1  2  3
      8     4
      7  6  5
[09:24:07]  Nesta busca trabalham 3 servidores.
[09:24:07]  Busca 4 iniciada.
[09:24:08]  Estatística da Busca 4: 339 nodos processados de 933 gerados.
Agora: 221 abertos e 338 fechados.
[09:24:13]  Estatística da Busca 4: 2303 nodos processados de 6309 gerados.
Agora: 1427 abertos e 2302 fechados.
[09:24:18]  Estatística da Busca 4: 3783 nodos processados de 10244
gerados. Agora: 2185 abertos e 3782 fechados.
[09:24:23]  Estatística da Busca 4: 5689 nodos processados de 15565
gerados. Agora: 3350 abertos e 5688 fechados.
[09:24:28]  Estatística da Busca 4: 7702 nodos processados de 20896
gerados. Agora: 4272 abertos e 7702 fechados.
[09:24:33]  Estatística da Busca 4: 9431 nodos processados de 25518
gerados. Agora: 5072 abertos e 9430 fechados.
[09:24:38]  Estatística da Busca 4: 11599 nodos processados de 31607
gerados. Agora: 6278 abertos e 11599 fechados.
[09:24:38]  Solução encontrada para a Busca 4!
[09:24:38]  Busca 4 encerrada.
      Tempo total: 00:00:35
      Nodos processados: 11646
      Nodos gerados: 31734
[09:24:43]  A solução da Busca 4 foi transmitida para o cliente.
```