Sistemas Inteligentes
Prof. Elder Rizzon Santos
Universidade Federal de Santa Catarina
Sistemas de Informação
Alunos: Camila dos Reis (15200909)
Renato Matta Machado Pereira da Silva (18100871)

Relatório do Trabalho sobre Métodos de busca (2019-1)

Respostas das propostas pedidas no relatório

1. Qual a representação (estrutura de dados) do estado;

Nosso estados (nodos) foram representados através de uma lista (ArrayList) de inteiros.
Cada nodo, tem também um um valor de custo armazenado em uma variável inteira, e uma lista (ArraList) de nodos filhos, que pode, ou não, ser uma lista vazia.

2. Qual a estrutura de dados para a fronteira e nodos fechados;

A nossa fronteira consiste um uma lista (ArrayList) de caminhos (classe Caminho). O caminho, por sua vez, é uma lista de Nodos.

3. Descrição da implementação (ideia geral e métodos relacionados) das heurísticas;

Implementamos 3 tipos de busca:

1. Custo uniforme: Nesse modo, os custos foram calculados simplesmente adicionando +1 nos custos dos nodos filhos, em relação ao nodo pai, ou seja, a cada nível, o custo aumenta em 1.

2. A* simples: Nesse modo, implementamos uma heurística simples, onde baseado nas posições dos quadrados do nodo que estamos calculando o peso em relação à posição dos quadrados do nosso objetivo (nodo com o estado final), adicionamos +1 para cada quadrado que não esteja na sua posição final.

3. A* melhorado: Conseguimos fazer um A* um pouco melhor, incrementando como calculamos a custo para cada quadrado que não está na sua posição final. Ao invés de simplesmente adicionar +1 para cada quadrado, nós calculamos a distância do quadrado, da posição

em que ele esta, até a posição onde deveria estar. Conseguimos verificar que com essa heurística melhorada o resultado é achado mais rápido (menos nodos abertos)

4. Como foi gerenciada a fronteira, verificações, quais etapas foram feitas ao adicionar um estado na fronteira (explicação das estratégias, respectivos métodos e possibilidades além do que foi implementado);
Cada vês que expandimos um nodo (criado seus filhos), pegamos o caminho até esse nodo, e para cada filho gerado, criamos um novo caminho incluindo todos os nodos já presentes nesse caminho, mais o filho.

Adicionamos os caminhos expandidos criados na nossa fronteira, e excluímos o caminho original (não expandido). Depois disso, ordenamos o fronteira novamente.

Para evitar que criássemos caminhos circulares, cada vez que expandimos um nodo passamos o caminho ao qual ele pertence no método que cria os filhos, filtrando os filhos criados, removendo todos que já pertençam ao caminho que ele está.

5. O papel de cada classe e os métodos principais;
Main:
Classe principal do nosso programa, utilizada para ler as entradas do usuário, iniciar a busca e imprimir o resultado.
(A explicação de cada método está detalhada com comentários no código fonte)

Algoritmo:
Esse arquivo não é uma Classe, mas um “enum”, que criamos para facilitar gerenciar os tipos de algoritmos utilizados (Custo uniforme, A* simples e A* melhorado)

Caminho:
Classe utilizada para armazenar a lista de nodos que representam um caminho que estamos percorrendo na nossa fronteira.
Métodos:

- getCustoTotal() : soma os custos de todos os nodos que tem e retorna o custo total.

- compareTo(): Implementamos o método compareTo para facilitar a ordenação dos caminhos.
- equals(): Implementamos o método equals para facilitar a comparação de caminhos.
- toString(): Implementamos o método toString para podermos imprimir um caminho facilmente.

Nodo:
Classe utilizada para armazenar n osso estados que foram representados através de uma lista (ArrayList) de inteiros.
Cada nodo, tem também um um valor de custo armazenado em uma variável inteira, e uma lista (ArraList) de nodos filhos, que pode, ou não, ser uma lista vazia.
métodos:

- getCusto() : Retorna o custo do nodo, de acrodo com qual Algoritmo foi escolhido pelo usuário.
- getPosicaoVazia() : retorna qual é a posição vazia (zero) do Nodo.
- expandeNodo(): gera os filhos do nodo.

ResultadoBusca:
Classe utilizada apenas para conter todas as informações que precisamos retornar no final do jogo, é bem simples e simplesmente tem um método imprimeTela() para imprimir todos os seus atributos
equals(): implementamos o equals para facilitar a comparação de nodos removendo a complexidade de métodos que precisem fazer isso.

Tabuleiro:
Classe mais complexa que fizemos, onde armazena toda a lógica do jogo.
Ela possui os seguintes atributos:

- Nodo estadoInicial: estado inicial do nosso jogo;
- Nodo estadoFinal: estado final definido do jogo;
- Nodo estadoAtual: estado atual de algum momento do jogo;
- ArrayList<Nodo> estadosVizitados: lista com todos os estados que já foram percorridos em algum momento do jogo;
- ArrayList<Caminho> fronteira: armazena a fronteira do nosso jogo, em algum momento do jogo;
- int maiorTamanhoFrontera: variável utilizada para armazenar a maior fronteira que do jogo, em algum momento do jogo.

métodos:

- acharCaminho(): método chamado pelo Main, inicia a busca do nodo final.
- buscaObjetivo(): método com o loop de busca, enquanto não acha um nodo igual ao nodo final definido, continua fazendo a busca. Esse é o método mais complexo, e tem comentários no código explicando cada passo.
- adicionaEstadosVizitados(): recebe um nodo, e caso o mesmo já não esteja na lista de estados visitados, adiciona ele na lista.

6. Caso algum dos objetivos não tenha sido alcançado explique o que você faria VS o que foi feito e exatamente qual o(s) problema(s) encontrado(s), bem como limitações da implementação;

Acreditamos ter alcançado os objetivos do trabalho, apesar de que achamos que a nossa heurística poderia ser ainda mais melhorada, possivelmente.
Uma limitação que temos, é não verificar se um estado inicial pode realmente levar até um estado final, existem casos onde o programa fica procurando indefinidamente, até que eventualmente trave.

7. Referente ao algoritmo MINMAX, apresente uma função de utilidade e uma função heurística para o jogo Ligue4. Apresente os critérios de análise para ambas funções e exatamente como os critérios são traduzidos para valores.

<u>Função heurística</u>

A função de utilidade no limite de profundidade definido. ela é utilizada para sabermos o quão próximo de uma vitória, ou derrota estamos.

pensamos na seguinte função:

f = 1*s + 10*d + 100*t + 1000*q

sendo:

s - o número de peças com sequências de uma peça (uma única peça);

d - o número de peças com sequências de duas peças;

t - o número de peças com sequências de três peças;

q - o número de peças com sequências de quatro peças;

Dessa forma, sequências grandes terão um peso muito maior do que sequências menores.

Função de utilidade

A função de utilidade verifica se um nodo é um nodo terminal (fim do jogo). A função deve ser bem simples, verificando se existe uma sequência de tamanho 4 de peças da mesma cor (na horizontal, vertical ou diagonal).

Podendo retornar, por exemplo, 1, no caso de vitória do computador, -1 para vitória do adversário, ou 0 para empate (para saber se houve empate deve ser verificado se ainda existem casas em branco, se não houverem mais, então é um empate, caso ainda não existe uma sequência de tamanho 4).