Régua-Puzzle(Jogo das Fichas)
Grupo: Marcos Mateus Oliveira dos Santos

1 - Problema
O jogo das fichas consiste em uma régua alinhada com um tamanho de 2N+1 de posições, onde 2N é o número de fichas alinhadas. No caso em que estamos trabalhando vamos considerar os fichas pretas e brancas, representadas por 1(pretas), -1(brancas) e 0 o espaço vazio.

2 - Implementação:

- **Linguagem utilizada:** C + +.
- **Formato de entrada:** Via teclado, informando o tamanho do tabuleiro, o estado inicial e o estado final e o resultado final de execução dos algoritmos será via output.txt.

- **Execução:**
    1) g++ -o exec -O3 *.cpp.
    2) ./exec.
    3) Escolha o tamanho do tabuleiro.
    4) Defina o estado inicial.
    5) Defina o estado final.

- **Estrutura:** O código é estruturado da seguinte forma. Possui um arquivo .cpp e .h para cada um dos algoritmos implementados.

(A) Arquivos:

- **Actions** (.cpp e .h): O está a implementação dos movimentos possíveis.

R1:

1	0	1	-1	-1

R2:

0	1	1	-1	-1

R3:

1	1	-1	0	-1

R4:

1	1	-1	-1	0

- **Backtracking** (.h e .cpp): Implementação do algoritmo backtracking.
- **BreadthFirstSearch** (.h e .cpp): Implementação do algoritmo de Busca em largura.
- **DepthFirstSearch** (.h e cpp): Implementação do algoritmo de Busca em Profundidade.
- **OrdinateSearch** (.h e .cpp): Implementação do algoritmo de Busca Ordenada.
- **GreedySearch** (.h e .cpp): Implementação do algoritmo de Busca Gulosa.
- **AStarSearch** (.h e .cpp): Implementação do algoritmo de Busca A*.
- **IDAStar** (.h e .cpp): Implementação do algoritmo de Busca IDA*.
- **No** (.h e .cpp): Nós utilizados em todos os algoritmos para criação do grafo de busca.

(B) Nó:

Os nós das buscas é estruturado da seguinte forma:

```
11  class No
12  {
13
14      private:
15          int cost;
16          int heuristic_cost;
17          vector<int> state;
18          No* father;
19          No* son1;
20          No* son2;
21          No* son3;
22          No* son4;
23
```

- Cost: Um inteiro onde armazena o custo do nó para as as buscas que utilizam custos.
- Heuristic Cost: Inteiro onde é armazenado o custo Heurístico do nó.
- State: É o vetor onde fica armazenado o estado do nó.
- Father: É o pai do nó.
- Son 1,2,3,4: São os possíveis filhos daquele nó

(C) Estruturas e funções:
Em todos os algoritmos implementados possuem uma estruturação em comum, o exemplo que será utilizado será o do Backtracking.

```
class BackTracking{


    private:
        No *root;
        int tam;
        vector<int> start;
        vector<int> goal;

    public:
        BackTracking(int tam,vector<int> start,vector<int> final_state);
        void init(std::ofstream& myfile);
        bool is_goal(vector<int> arr);
        void set_goal(vector<int> arr);
        void set_start(vector<int> arr);
        void path(No *no,std::ofstream& myfile);
};
```

- root: é a raiz do grafo.
- tam: tamanho do Nó
- goal: é o estado final ou objetivo.

- **void init()**:

  É a função principal de todos os algoritmos, é essa função que está estruturada todo a lógica dos algoritmos de busca, essa lógica possui variações de acordo com o tipo de algoritmo implementado

  O algoritmo de Busca em Largura, Busca Ordenada,Gulosa ,A*, possui duas filas de abertos e fechados.

```
queue<No*> openers;
queue<No*> closed;
```

Busca em profundidade, possui duas pilhas de abertos e fechados.

```
stack<No*> openers;
stack<No*> closed;
```

- **is_goal():**

  É a função que verifica se o estados passado por parâmetro é o estado final buscado.

- **set_goal():**

  É a função que irá estabelecer o objetivo.

- **Algoritmos que utilizam custos**:

  Os algoritmos que utilizam custos em seus nós para chegar ao objetivo final, possui uma função cujo objetivo é retornar o menor custo na lista de abertos.

```
int OrdinateSearch::get_lower_cost(int lower, vector<No*> openers){


    for(int i = 0; i < openers.size(); i++)
    {

        if(openers[i]->get_cost() < lower || lower == -1){

            return openers[i]->get_cost();
        }

    }

    return -1;
}
```

O custo é determinado pela sequência do movimento, o R1 possui custo 1, o R2 possui custo 2 e assim respectivamente.

Há uma variação dessa função em algoritmos que utilizam heurísticas, que verificam a soma do custo do movimento mais o custo heurístico calculado pela função heurística.

```
int AStarSearch::get_lower_cost(int lower, vector<No*> openers){


    for(int i = 0; i < openers.size(); i++)
    {

        if(openers[i]->get_cost() + openers[i]->get_heuristic_cost() < lower || lower == -1){

            return openers[i]->get_cost();
        }

    }

    return -1;

}
```

- **Heurística:**

Os algoritmos que possuem uma heurística são os Busca Gulosa, A* e IDA*, para esse problema **Régua-Puzzle(Jogo das Fichas),** foi utilizado uma heurística que consiste em calcular quantas peças estão fora de seus lugares em comparação ao objetivo, assim retornando um custo.

Exemplo:

Objetivo:

-1	1	0	1	-1

Estado atual:

1	0	1	-1	-1

O custo heurístico desse estado é de 4.

```cpp
int AStarSearch::heuristic(vector<int> arr){


    int count = 0;
    vector<int> aux = this->goal;

    for (auto it = arr.begin(), it2 = aux.begin();
              it != arr.end() && it2 != aux.end(); it++, it2++) {

            if(*it != *it2)
            {
                count++;
            }


    }

    return count;
}
```

3 - Execuções:

Essa seção tem como objetivo demonstrar algumas execuções e mostrar os dados estatísticos dos algoritmos, irá ser demonstrada dois casos de estados iniciais e objetivos diferentes.

1)

Estado inicial:

1	1	0	-1	-1

Estado final:

1	-1	-1	0	1

Backtracking:
Resultado: Sucesso
Visitados: 18
Profundidade: 15
Expandidos: 18
Fato médio de ramificação: 1.21429
Tempo da Busca Backtracking: 0.018435s
Caminho:
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,0,1,-1] => [1,0,-1,1,-1] => [0,1,-1,1,-1] => [-1,1,0,1,-1] => [-1,0,1,1,-1] => [-1,1,1,0,-1] =>[-1,1,1,-1,0] => [-1,1,0,-1,1] => [-1,0,1,-1,1] => [0,-1,1,-1,1] => [1,-1,0,-1,1]=>[1,0,-1,-1,1] => [1,-1,-1,0,1,]
Custo do caminho: 0

Busca em largura:
Resultado: Sucesso
Visitados: 55
Profundidade: 15
Expandidos: 32
Fato médio de ramificação: 0.563636
Tempo da Busca Backtracking: 0.036867s
Caminho:
[1,1,0,-1,-1,] => [1,0,1,-1,-1,] => [1,-1,1,0,-1,] => [1,-1,0,1,-1,] => [1,-1,-1,1,0,] => [1,-1,-1,0,1]
Custo do caminho: 0

Busca em Profundidade:
Resultado: Sucesso
Visitados: 13
Profundidade: 15
Expandidos: 6
Fato médio de ramificação: 0.384615
Tempo da Busca Backtracking: 0.018569s
Caminho:
[1,1,0,-1,-1] => [1,1,-1,-1,0] => [1,1,-1,0,-1] => [1,0,-1,1,-1] => [1,-1,0,1,-1] => [1,-1,-1,1,0] => [1,-1,-1,0,1]
Custo do caminho: 0

Busca Ordenada:
Resultado: Sucesso
Visitados: 54
Profundidade: 15
Expandidos: 32
Fato médio de ramificação: 0.574074
Tempo da Busca Backtracking: 0.03352s
Caminho:
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,0,1,-1] => [1,-1,-1,1,0] => [1,-1,-1,0,1]
Custo: 33

Busca Gulosa:
Resultado: Sucesso
Visitados: 54
Profundidade: 5
Expandidos: 32
Fato médio de ramificação: 0.574074
Tempo da Busca Backtracking: 0.039244s
Caminho:
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,0,1,-1] => [1,-1,-1,1,0] => [1,-1,-1,0,1]
Custo: 11

Busca com A*:
Resultado: Sucesso
Visitados: 54
Profundidade: 54
Expandidos: 32
Fato médio de ramificação: 0.574074
Tempo da Busca Backtracking: 0.034166s
Caminho:
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,0,1,-1] => [1,-1,-1,1,0] => [1,-1,-1,0,1]
Custo: 44

Busca com IDA:
Resultado: Sucesso
Visitados: 125
Profundidade: 1
Tempo da Busca Ordenada: 0.077651s
Fato médio de ramificação: 0.992
Patamar máximo: 12
Caminho:
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,0,1,-1] => [1,-1,-1,1,0] => [1,-1,-1,0,1]
Custo: 48

2)

Estado inicial:

1	1	0	-1	-1

Estado final:

-1	-1	0	1	1

Backtracking:
Resultado: Fracasso
Visitados: 19
Profundidade: 15
Expandidos: 19
Fato médio de ramificação: 1.2
Tempo da Busca Backtracking: 0.022219s

Busca em largura:
Resultado: Sucesso
Visitados: 273
Profundidade: 24
Expandidos: 170
Fato médio de ramificação: 0.619048
Tempo da Busca Backtracking: 0.203607s
Caminho:
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,1,-1,0] => [1,-1,0,-1,1] => [0,-1,1,-1,1] => [-1,0,1,-1,1] => [-1,-1,1,0,1] => [-1,-1,0,1,1]

Busca em Profundidade:
Resultado: Sucesso
Visitados: 28
Profundidade: 34
Expandidos: 17
Fato médio de ramificação: 0.571429
Tempo da Busca Backtracking: 0.027215s
Caminho:
[1,1,0,-1,-1] => [1,1,-1,-1,0] => [1,1,-1,0,-1] => [1,0,-1,1,-1] => [1,-1,0,1,-1] => [1,-1,-1,1,0] => [1,-1,-1,0,1] => [1,0,-1,-1,1] => [1,-1,0,-1,1] => [0,-1,1,-1,1] => [-1,0,1,-1,1] => [-1,-1,1,0,1] => [-1,-1,1,1,0] => [-1,-1,0,1,1]

Busca Ordenada:
Resultado: Sucesso
Visitados: 272
Profundidade: 24
Expandidos: 170
Fato médio de ramificação: 0.621324
Tempo da Busca Backtracking: 0.199428s
Caminho
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,1,-1,0,] => [1,-1,0,-1,1,] => [0,-1,1,-1,1] => [-1,0,1,-1,1] => [-1,-1,1,0,1] => [-1,-1,0,1,1]
Custo: 90

Busca Gulosa:
Resultado: Sucesso
Visitados: 273
Profundidade: 278
Expandidos: 171
Fato médio de ramificação: 0.622711
Tempo da Busca Backtracking: 0.170776s
Caminho:
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,1,-1,0] => [1,-1,0,-1,1] => [0,-1,1,-1,1] => [-1,0,1,-1,1] => [-1,-1,1,0,1] => [-1,-1,0,1,1]
Custo: 23

Busca com A*:
Resultado: Sucesso
Visitados: 273
Profundidade: 273
Expandidos: 171
Fato médio de ramificação: 0.622711
Tempo da Busca Backtracking: 0.180509s
Caminho:
[1,1,0,-1,-1] => [1,0,1,-1,-1] => [1,-1,1,0,-1] => [1,-1,1,-1,0] => [1,-1,0,-1,1] => [0,-1,1,-1,1] => [-1,0,1,-1,1] => [-1,-1,1,0,1] => [-1,-1,0,1,1]
Custo: 571234417

Busca com IDA:
Resultado: Sucesso
Visitados: 789
Profundidade: 2
Tempo da Busca Ordenada: 0.651428s
Fato médio de ramificação: 0.998733
Patamar máximo: 19
Caminho:
[1,1,0,-1,-1] => [1,1,-1,0,-1] => [1,0,-1,1,-1] => [0,1,-1,1,-1] => [-1,1,0,1,-1] =>[-1,1,-1,1,0] => [-1,1,-1,0,1] => [-1,0,-1,1,1] => [-1,-1,0,1,1]
Custo: 117

4 - Dificuldades encontradas

A primeira dificuldade encontrada durante o desenvolvimento deste trabalho foi pensar na estruturação do projeto, como iria organizar a execução dos algoritmos e dos movimentos do Régua-Puzzel.
A segunda dificuldade encontrada foi fazer o backtracking, todos os algoritmos utilizam como base o backtracking então a implementação geral dependia dele, uma vez implementado os outros algoritmos foram desenvolvidos com tranquilidade e os problemas encontrados foram resolvidos rapidamente.
Por fim foi encontrar a melhor função heurística para os algoritmos, Guloso, A* e IDA*, que neste caso a melhor função heurística encontrada foi a já explicada nesse documento.