Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação
SCC0503 - Algoritmos e Estruturas de Dados II

Relatório Exercício 06

Alunos: João Otávio da Silva, Leonardo Gonçalves Chahud
Professor: Leonardo Tórtoro Pereira

Julho
2022

Conteúdo

1 Introdução

A proposta principal do exercício é criar um nível de jogo através do algoritmo de geração aleatória de Dungeons. Esse gerador, é responsável por: criar criar salas em pontos aleatórias, criar caminhos entre elas, designar salas de interesse e distribuir chaves e fechaduras através da masmorra.

Com o nível gerado, são realizadas três tipos de travessias (BFS, DFS e A*) e são feitas comparações entre elas.

2 Desenvolvimento

2.1 Grafo

A representação utilizada foi a implementação de grafo com lista de adjacência, a mesma vista em em aula, `GraphList`.

- **Vértice:** Elemento fundamental do grafo, possui um dado, no caso desse exercício, uma sala.
- **Aresta:** Conexão entre dois vértices, possui a referência para o vértice de destino e um peso.
- **Lista de Vértices:** Uma lista que contém as referências para todos os vértices pertencentes ao grafo.
- **Lista de Adjacência:** Uma lista onde para cada vértice há outra lista, contendo referências para todos as arestas que saem do mesmo.

2.2 Geração de Masmorra

A geração das masmorras aleatórias se dá da seguinte forma:

- Primeiro inicializamos o gerador, passando o número de salas e a semente para a geração de números aleatórios.
- Depois, pegamos o dungeon gerado, que é um grafo. Nesse grafo, os vértices são do tipo "Room", os quais possuem retângulos (coordenadas e dimensões gerados aleatoriamente, tomando cuidado para não ocorrer sobreposição) e representam as salas físicas do jogo.
- Agora, realizamos a triangulação de Delaunay sobre o grafo, o qual irá acrescentar novas arestas no mesmo.
- Após isso, é feita uma "limpeza"nas arestas do calabouço gerado, através do algoritmo de árvore geradora mínima de Prim.
- Agora, são marcadas algumas salas de interesse. A sala mais central é marcada como um checkpoint, a mais periférica como o começo e a mais distante da mais periférica como saída. Essas salas são achadas com base nas excentricidades dos vértices. As excentricidades, necessitam das menores distâncias entre os vértices, que são calculadas pelo algoritmo de Floyd-Warshall.
- Por fim, são sorteadas algumas salas para conterem chaves e alguns caminhos para conterem fechaduras, tomando cuidado para não colocar uma passagem bloqueada antes de uma chave, o que tornaria a conclusão da masmorra impossível.

2.3 A*

O A* é um algoritmo para buscar um caminho no grafo. Ele escolhe o próximo vértice a ser visitado com base na seguinte lógica: para cada vértice possível de se ir, escolha o que possuí a menor distância estimada, sendo essa a distância real do vértice inicial até o candidato, mais a heurística do candidato até o destino. A heurística, é uma estimativa da distância entre dois vértices, fácil de se calcular e que pode variar a depender do contexto.

```
// Distância real do vértice inicial até o vértice i + heurística do vértice i até o destino
float[] distancesToDestination = new float[getGraph().getNumberOfVertices()];
Arrays.fill(distancesToDestination, Float.POSITIVE_INFINITY);
int sourceIndex = getGraph().getVertices().indexOf(source);
distancesToDestination[sourceIndex] = euclideanDistance((Room) source, (Room) destination);
setDistanceToVertex(sourceIndex,  distance: 0f);
List<Vertex> verticesToVisit = new ArrayList<>();
verticesToVisit.add(source);
while (!verticesToVisit.isEmpty()) {
    Vertex currentVisitedVertex = getNextVertex(verticesToVisit, distancesToDestination);
    int currentVisitedVertexIndex = getGraph().getVertices().indexOf(currentVisitedVertex);
    addToPath(currentVisitedVertex);
    if (currentVisitedVertex == destination) {
        break;
    }
    verticesToVisit.remove(currentVisitedVertex);
    Vertex adjacentVertex = getGraph().getFirstConnectedVertex(currentVisitedVertex);
    while (adjacentVertex != null) {
        int adjacentVertexIndex = getGraph().getVertices().indexOf(adjacentVertex);
        float realDistance = getDistanceToVertex(currentVisitedVertexIndex) +
                manhattanDistance((Room) currentVisitedVertex, (Room) adjacentVertex);
        if (realDistance < getDistanceToVertex(adjacentVertexIndex)) {
            setPredecessorVertexIndex(adjacentVertexIndex, currentVisitedVertexIndex);
            setDistanceToVertex(adjacentVertexIndex, realDistance);
            distancesToDestination[adjacentVertexIndex] = realDistance +
                    euclideanDistance((Room) adjacentVertex, (Room) destination);
            if (!verticesToVisit.contains(adjacentVertex)) {
                verticesToVisit.add(adjacentVertex);
            }
        }
        adjacentVertex = getGraph().getNextConnectedVertex(currentVisitedVertex, adjacentVertex);
    }
}
```

Figura 1: Algoritmo A*

Observação: A princípio, a distância euclidiana estava sendo usada como peso da aresta. Porém, ao utilizarmos o A*, passamos a utilizar a distância euclidiana como heurística e a distância de manhattan como distância real, isso até faz mais sentido no contexto do exercício, dificilmente o caminho entre uma sala e outra seria uma linha reta (distância euclidiana).

A fim de facilitar a implementação e dar enfoque na construção do algoritmo, não modificamos a forma como as arestas são construídas, mas sim colocamos a nova distância real (manhattam) *hard coded* na implementação do A*, ao invés de pegar a distância entre dois vértices adjacentes através da função `getDistance`, disponibilizada pelo grafo, calculamos ela na hora, através da função `manhattanDistance`.

Em uma implementação mais geral do algoritmo, faria mais sentido ele receber, além dos vértices de origem e destino, as funções (específicas para cada contexto) que calculam a distância real e a heurística.

3 Resultados

A seguir, serão apresentados os resultados obtidos para três casos de teste, para cada um deles, as informações serão mostradas da seguinte forma:

1. Parâmetros de aleatoriedade.
2. Representação gráfica da masmorra.
3. Representação gráfica do caminho mais curto na masmorra.
4. Ordem de visita dos algoritmos de travessia na seguinte ordem: BFS, DFS e A*;

As representações gráficas seguem as seguintes convenções:

- Retângulo azul: sala comum.
- Retângulo roxo: sala com chave.
- Retângulo preenchido de verde: sala inicial.
- Retângulo preenchido de amarelo: sala de checkpoint.
- Retângulo preenchido de vermelho: sala final.
- Aresta pink: caminho comum.
- Aresta ciano: arestas percorridas no caminho mais curto.
- Aresta laranja: caminho com tranca.

Observação: Optamos por mostrar duas imagens da dungeon, uma normal e uma com o caminho mais curto, a fim de evitar confusões entre arestas que contém uma tranca e arestas que fazem parte do caminho mais curto.

3.1 Caso 1

Semente: 20
Quantidade de Salas = 15
Chance de criar chave/fechadura: 20%

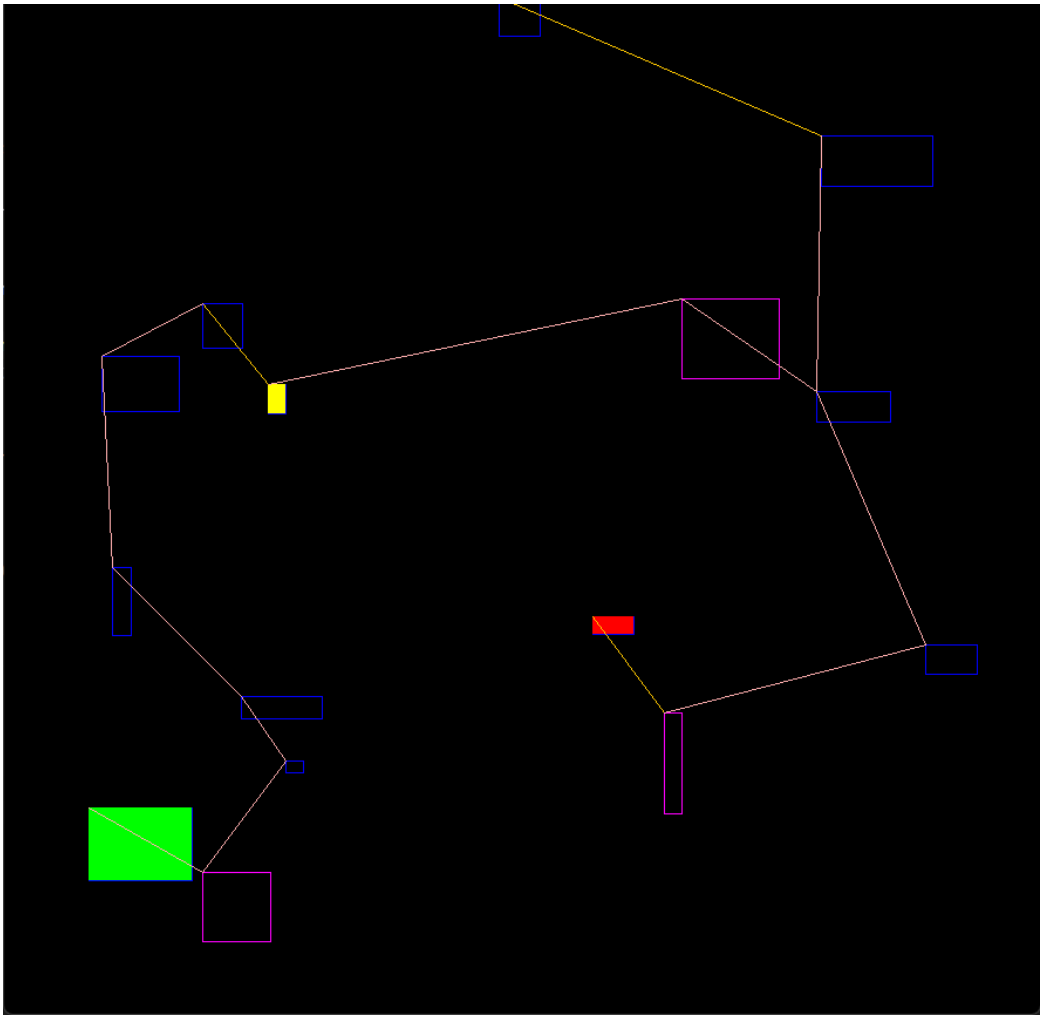

Figura 2: Dungeon Caso 1

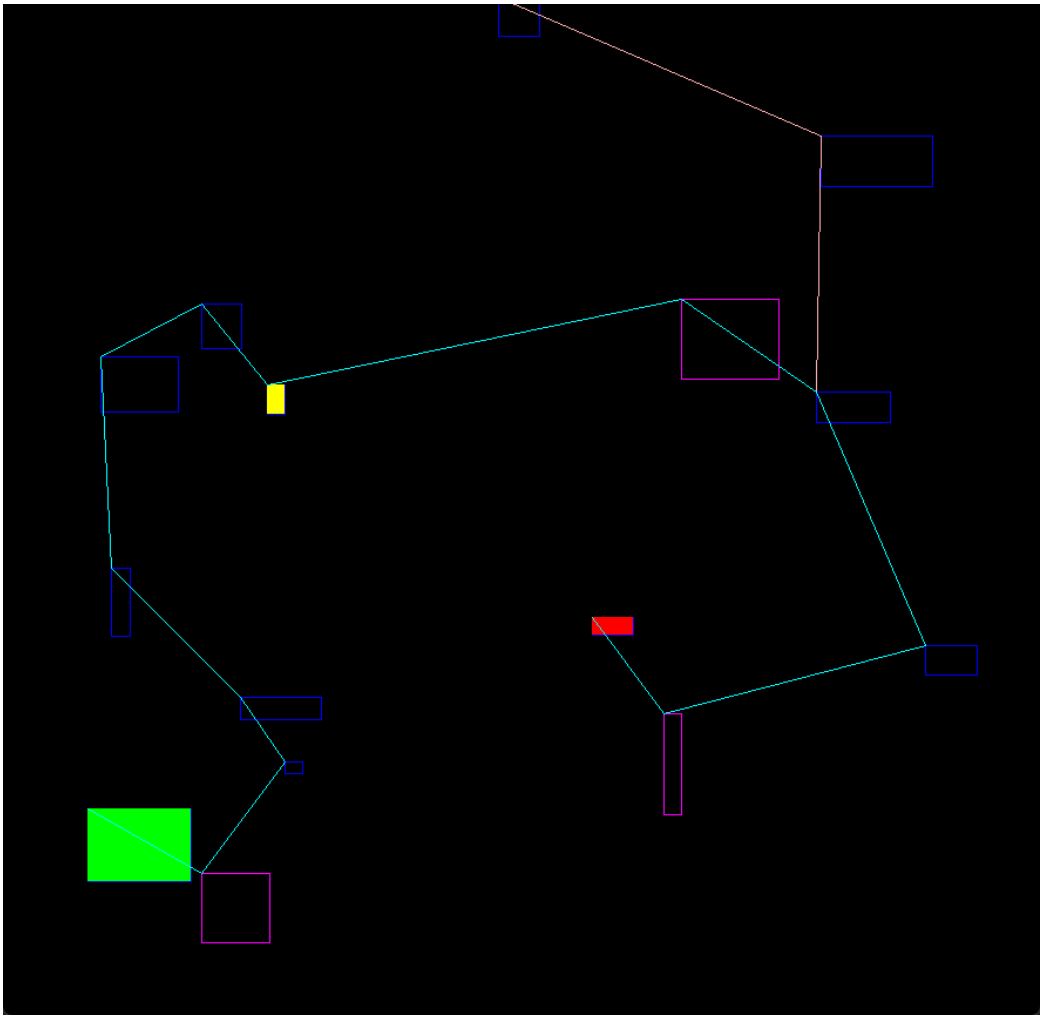

Figura 3: Caminho Caso 1

```
Room{(X, Y)= (80.0,716.0
Room{(X, Y)= (177.0,771.0)}
Room{(X, Y)= (248.0,676.0)}
Room{(X, Y)= (210.0,621.0)}
Room{(X, Y)= (100.0,511.0)}
Room{(X, Y)= (91.0,331.0)}
Room{(X, Y)= (177.0,286.0)}
Room{(X, Y)= (233.0,355.0)}
Room{(X, Y)= (586.0,282.0)}
Room{(X, Y)= (701.0,361.0)}
Room{(X, Y)= (705.0,143.0)}
Room{(X, Y)= (794.0,577.0)}
Room{(X, Y)= (430.0,25.0)}
Room{(X, Y)= (571.0,635.0)}
Room{(X, Y)= (510.0,553.0)}
```

Figura 4: BFS Caso 1

```
Room{(X, Y)= (80.0,716.0)}
Room{(X, Y)= (177.0,771.0)}
Room{(X, Y)= (248.0,676.0)}
Room{(X, Y)= (210.0,621.0)}
Room{(X, Y)= (100.0,511.0)}
Room{(X, Y)= (91.0,331.0)}
Room{(X, Y)= (177.0,286.0)}
Room{(X, Y)= (233.0,355.0)}
Room{(X, Y)= (586.0,282.0)}
Room{(X, Y)= (701.0,361.0)}
Room{(X, Y)= (705.0,143.0)}
Room{(X, Y)= (430.0,25.0)}
Room{(X, Y)= (794.0,577.0)}
Room{(X, Y)= (571.0,635.0)}
Room{(X, Y)= (510.0,553.0)}
```

Figura 5: DFS Caso 1

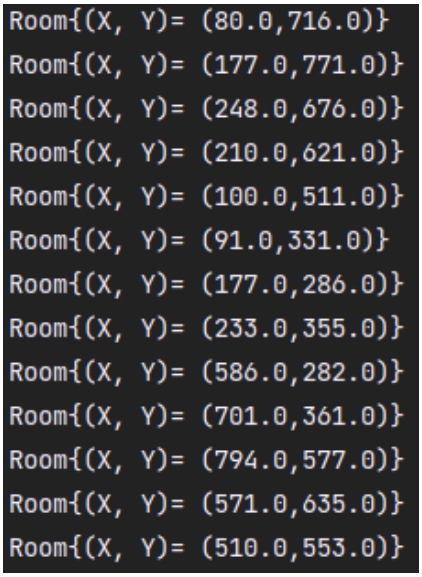

Figura 6: A* Caso 1

Nesse caso, como a quantidade de salas é pequena, a masmorra é bem linear, só é perceptível a diferença entre as travessias a partir da décima visita.

3.2 Caso 2

Semente: 42
Quantidade de Salas = 20
Chance de criar chave/fechadura: 30%

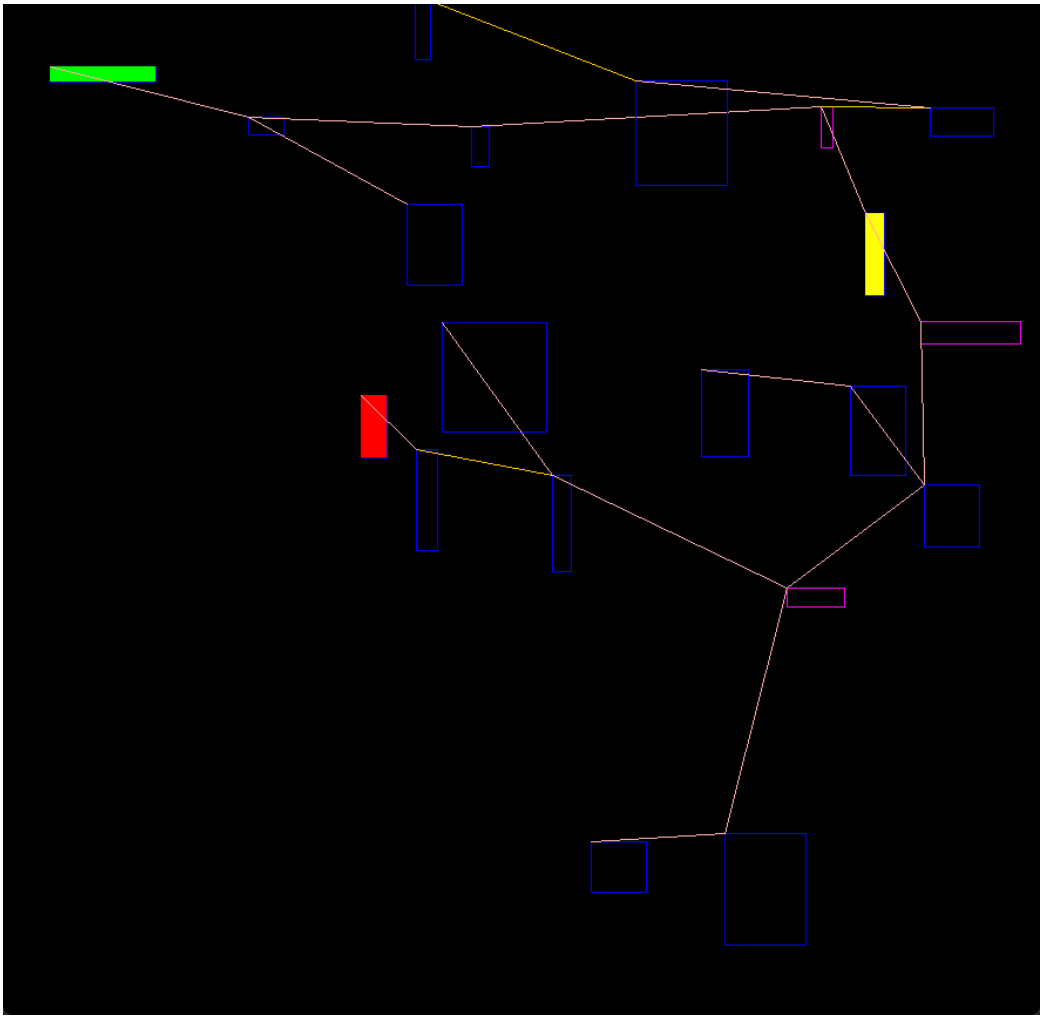

Figura 7: Dungeon Caso 2

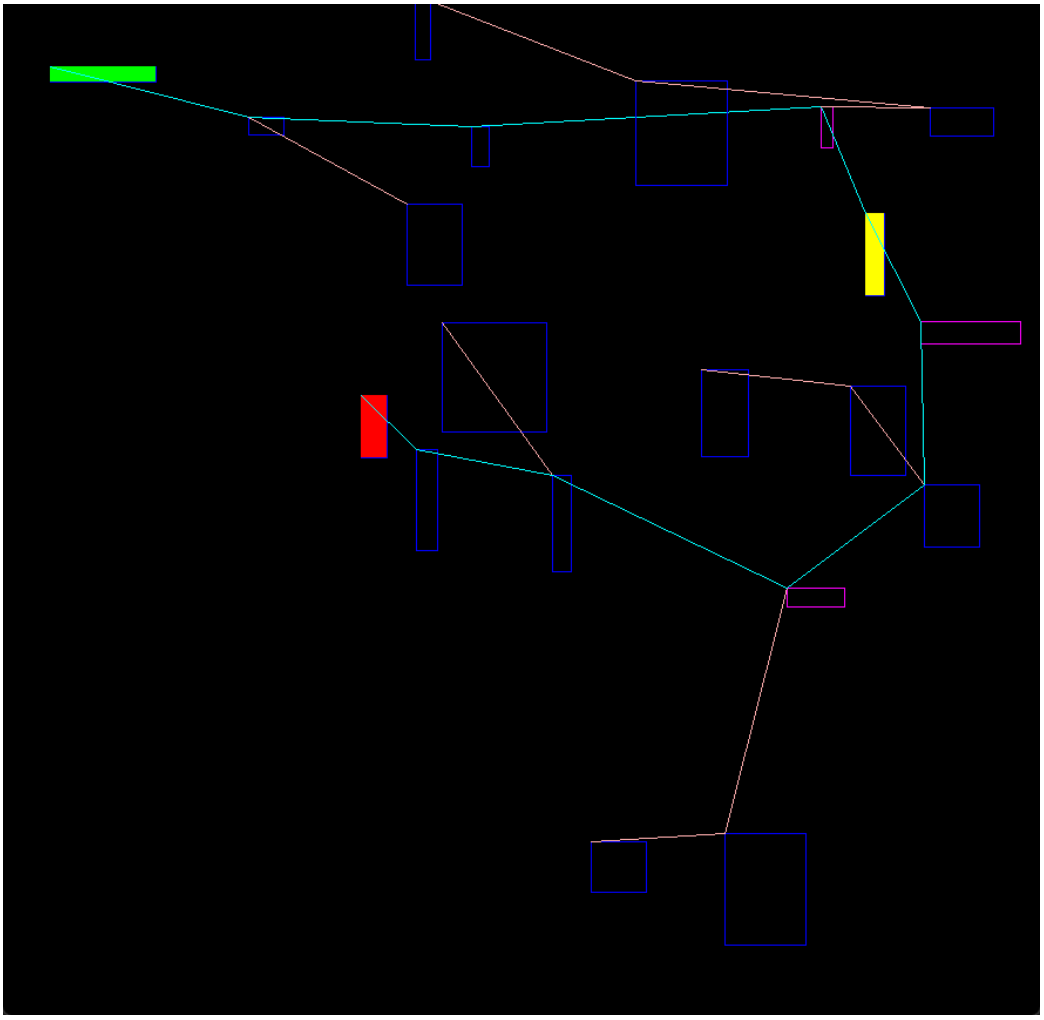

Figura 8: Caminho Caso 2

```
Room{(X, Y)= (48.0,84.0)}
Room{(X, Y)= (217.0,127.0)}
Room{(X, Y)= (407.0,135.0)}
Room{(X, Y)= (352.0,201.0)}
Room{(X, Y)= (705.0,118.0)}
Room{(X, Y)= (743.0,209.0)}
Room{(X, Y)= (798.0,119.0)}
Room{(X, Y)= (790.0,301.0)}
Room{(X, Y)= (547.0,96.0)}
Room{(X, Y)= (793.0,440.0)}
Room{(X, Y)= (359.0,23.0)}
Room{(X, Y)= (730.0,356.0)}
Room{(X, Y)= (676.0,528.0)}
Room{(X, Y)= (603.0,342.0)}
Room{(X, Y)= (476.0,432.0)}
Room{(X, Y)= (623.0,737.0)}
Room{(X, Y)= (382.0,302.0)}
Room{(X, Y)= (360.0,410.0)}
Room{(X, Y)= (509.0,744.0)}
Room{(X, Y)= (313.0,364.0)}
```

Figura 9: BFS Caso 2

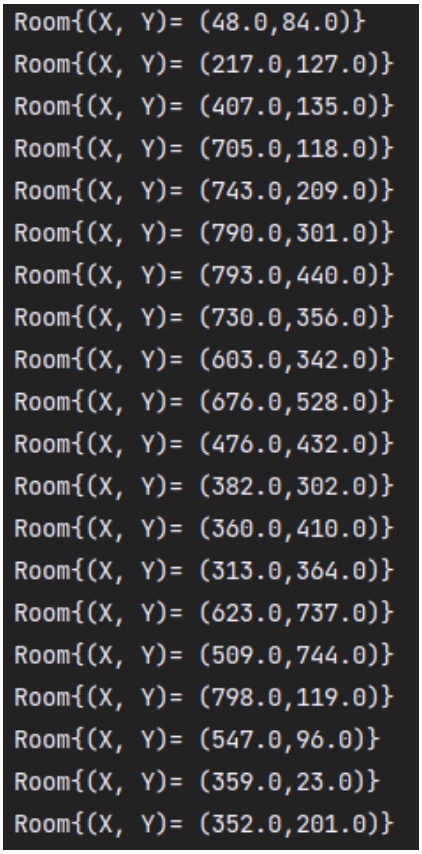

Figura 10: DFS Caso 2

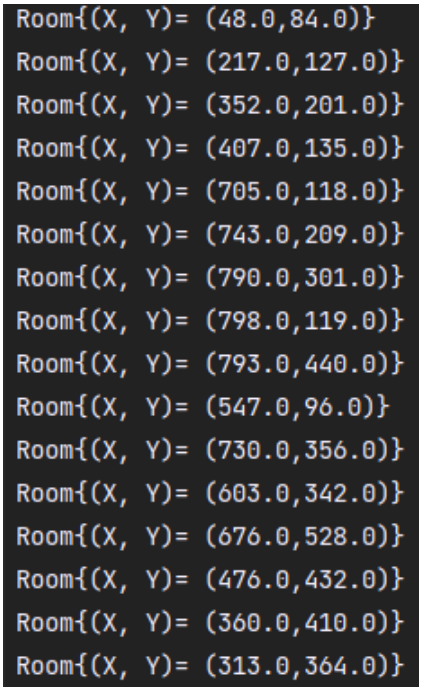

Figura 11: A* Caso 2

Nesse caso é interessante notar o comportamento do A* ao visitar a terceira sala, como ela está a uma distância euclidiana pequena do destino, o algoritmo supõe que é um bom caminho, porém isso não é verdade, pois esse caminho leva a uma sala sem saída.

3.3 Caso 3

Semente: 7896
Quantidade de Salas = 35
Chance de criar chave/fechadura: 40%

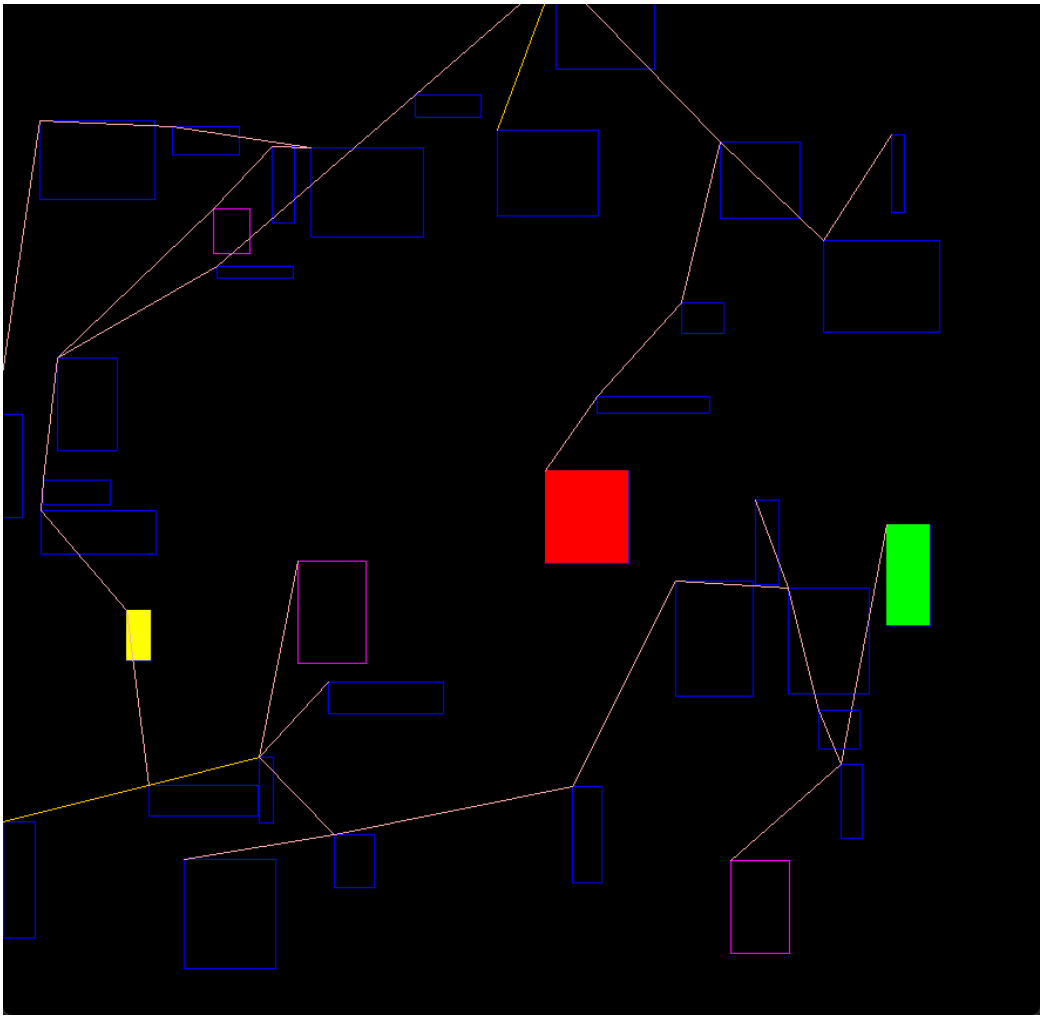

Figura 12: Dungeon Caso 3

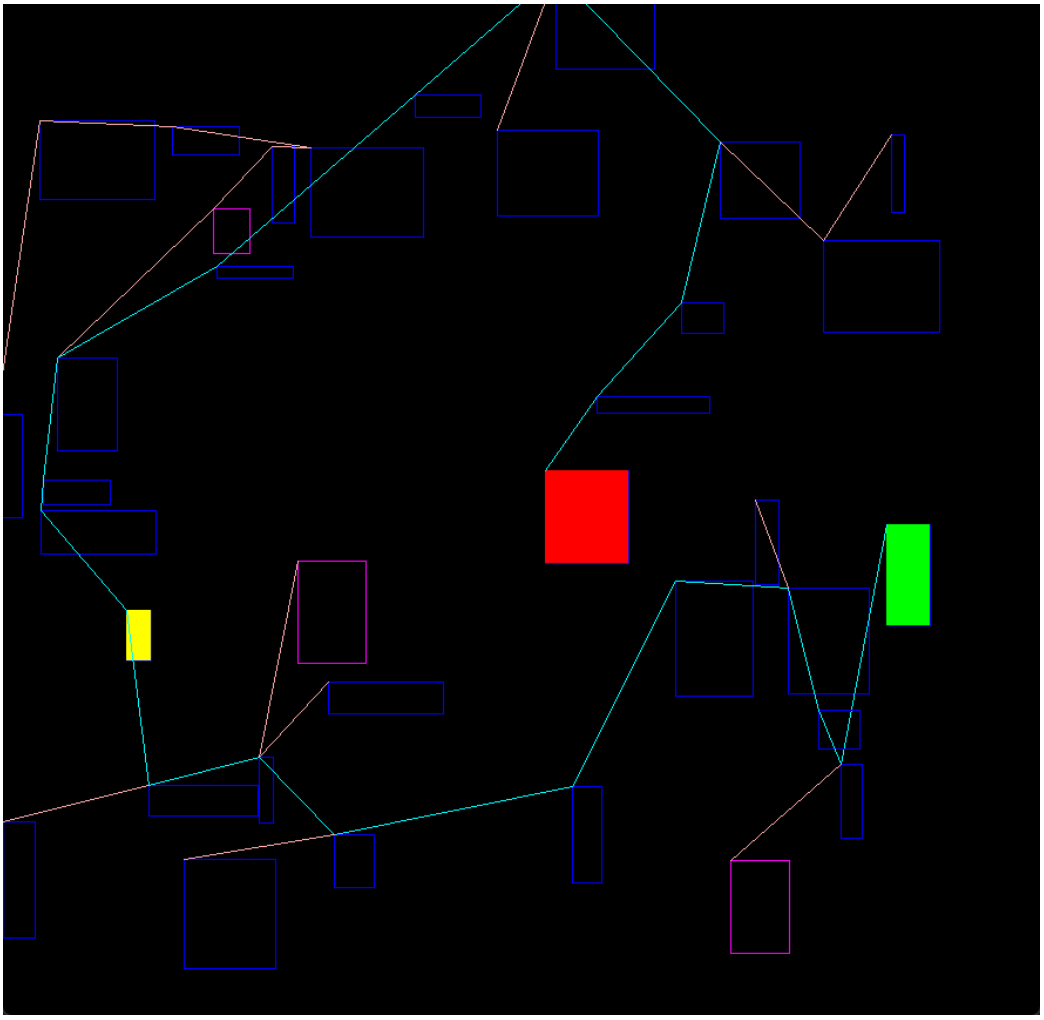

Figura 13: Caminho Caso 3

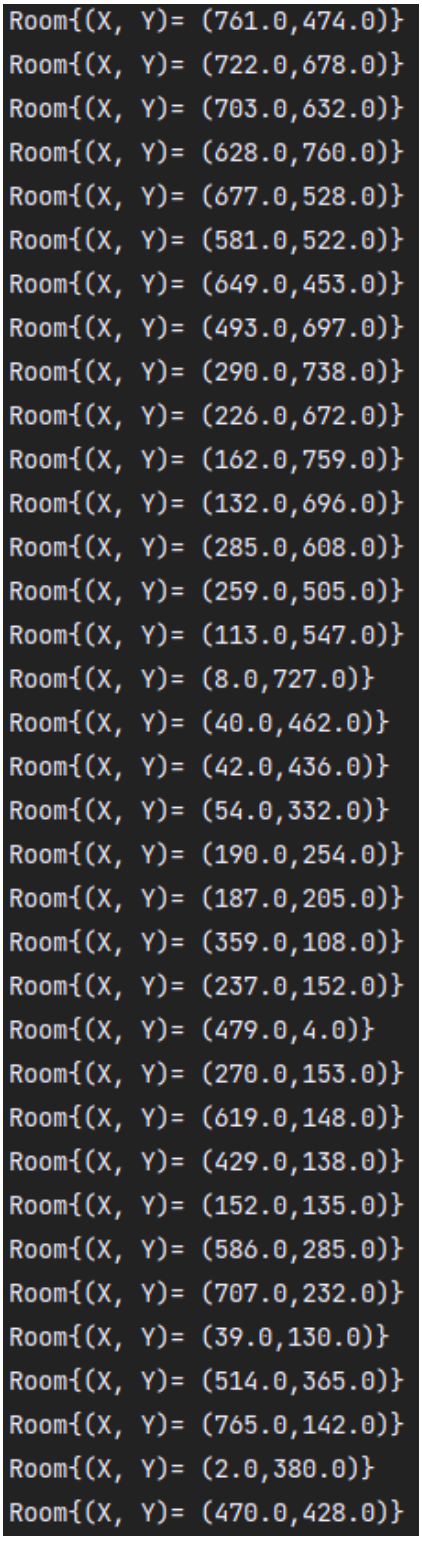

Figura 14: BFS Caso 3

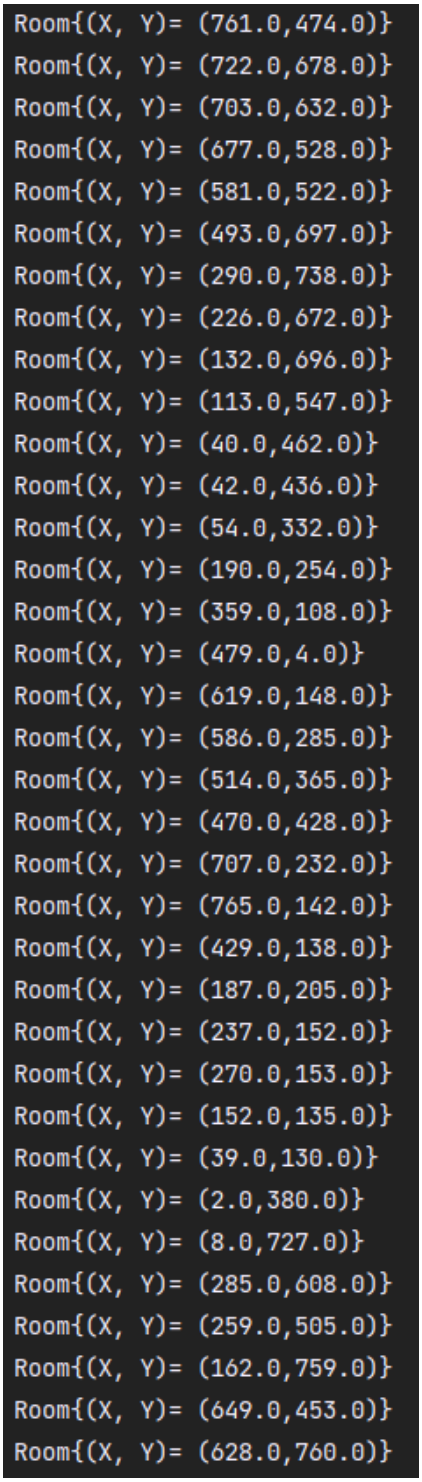

Figura 15: DFS Caso 3

```
Room{(X, Y)= (761.0,474.0)}
Room{(X, Y)= (722.0,678.0)}
Room{(X, Y)= (703.0,632.0)}
Room{(X, Y)= (677.0,528.0)}
Room{(X, Y)= (581.0,522.0)}
Room{(X, Y)= (649.0,453.0)}
Room{(X, Y)= (628.0,760.0)}
Room{(X, Y)= (493.0,697.0)}
Room{(X, Y)= (290.0,738.0)}
Room{(X, Y)= (226.0,672.0)}
Room{(X, Y)= (285.0,608.0)}
Room{(X, Y)= (259.0,505.0)}
Room{(X, Y)= (162.0,759.0)}
Room{(X, Y)= (132.0,696.0)}
Room{(X, Y)= (113.0,547.0)}
Room{(X, Y)= (8.0,727.0)}
Room{(X, Y)= (40.0,462.0)}
Room{(X, Y)= (42.0,436.0)}
Room{(X, Y)= (54.0,332.0)}
Room{(X, Y)= (187.0,205.0)}
Room{(X, Y)= (237.0,152.0)}
Room{(X, Y)= (270.0,153.0)}
Room{(X, Y)= (190.0,254.0)}
Room{(X, Y)= (359.0,108.0)}
Room{(X, Y)= (479.0,4.0)}
Room{(X, Y)= (619.0,148.0)}
Room{(X, Y)= (707.0,232.0)}
Room{(X, Y)= (586.0,285.0)}
Room{(X, Y)= (514.0,365.0)}
Room{(X, Y)= (470.0,428.0)}
```

Figura 16: A* Caso 3

3.4 Conclusão

Por fim, podemos fazer algumas conclusões a respeito dos resultados observados, e traçar um paralelo entre os algoritmos de travessia e o contexto do exercício. Cada travessia pode simular o comportamento de um jogador com um determinado perfil.

- **BFS:** Um jogador que visita todos as salas vizinhas antes de de fato avançar para a próxima, ou seja completa tudo possível na sala que está e vai avançando.
- **DFS:** um jogador que avança o máximo possível e depois vai retrocedendo e completando o que deixou pra trás.
- **A*:** Um jogador que somente busca chegar ao final da masmorra, através do menor caminho possível. Para isso, ele tem um conhecimento a mais (heurística) que serve para ajudá-lo na escolha do caminho, podendo ser por exemplo, uma bússola que aponta para o fim da masmorra.