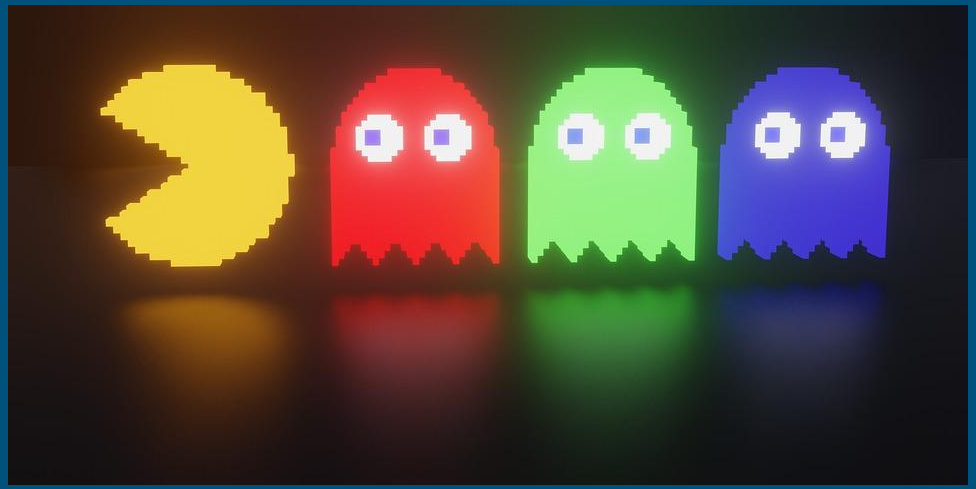
Projeto 1 - Pacman

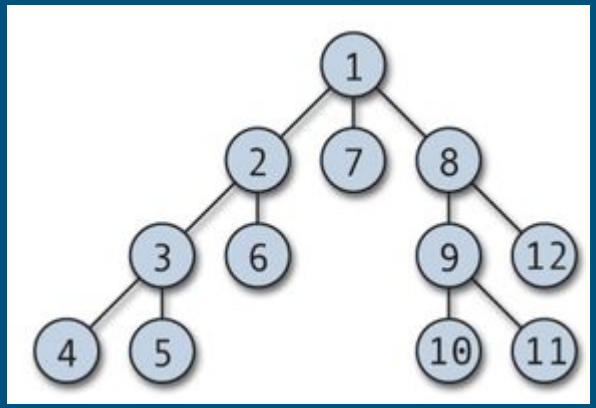
Implementação das funções de busca em Profundidade Largura e A*, e da função de variação de custo

Introdução

Neste projeto, o agente Pacman tem como objetivo encontrar caminhos através de labirintos para chegar a um determinado local. Foram construídos algoritmos de pesquisa, de profundidade de largura, variação de custo e A*.



1. Busca em profundidade (Depth First Search)



https://pt.wikipedia.org/wiki/Busca_em_profundidade

Depth Fisrt Search (DFS): DFS seleciona o nó não expandido mais profundo na árvore de busca para expansão. Ele guarda um caminho único entre a raíz e uma folha, juntamente com os irmãos não expandidos restantes para cada nó no caminho.

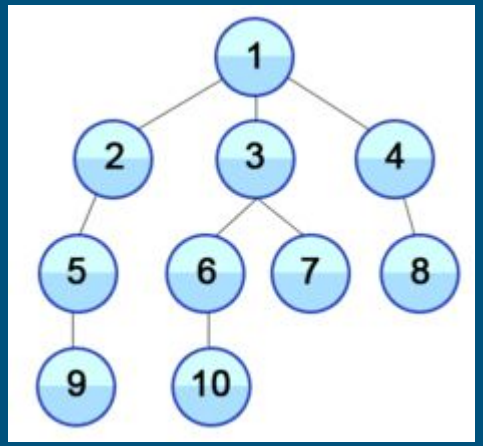
Classificador	Labirinto	Nós Expandidos	Score
DFS	tiny	15	500
	medium	146	380
	big	390	300

1. Busca em profundidade (Depth First Search)

1.O nó raiz vai previamente para a lista de visitados.
2. Em seguida irá verificar os próximos, e ir empilhando.
3. Depois de empilhados é pego do topo da pilha e testado se são objetivo e inseridos na lista de visitados.
4. É repetido o passo 2 e 3 até que a pilha esteja vazia.

```python
def depthFirstSearch(problem: SearchProblem):

    #pega o estado inicial do pacman
    root = Node(problem.getStartState())
    #lista de nós visitados
    visited = []
    #pilha pra controlar o backtracking
    stack = Stack()

    #enquanto o nó não é o objetivo
    while not problem.isGoalState(root.state):
        #insere o nó como visitado  1
        visited.append(root.state)
        #getSuccessors sucessor, ação para chegar lá, e custo getSuccessors
        for state,action,cost in problem.getSuccessors(root.state):
            #cria o nó, com a ação e o custo
            child = Node(state,action,cost,root)
            #se ainda não foi visitado vai colocar na pilha
            if not child.state in visited:
                #insere o nó no topo da pilha  2
                stack.push(child)
        #vão sendo removidos e testados como objetivo ou não
        root = stack.pop()  3

    #se root state é o objetivo ele vai colocar na solução o caminho
    solution = []
    while root.parent:
        solution.insert(0,root.action)
        root = root.parent

    return solution
```

2. Busca em largura (Breadth First Search)



https://pt.wikipedia.org/wiki/Busca_em_largura

Breadth First Search (BFS): BFS seleciona o nó mais superficial que não foi expandido ainda na árvore de busca para expansão. É um algoritmo completo e ótimo quando o custo de cada passo é igual..

Classificador	Labirinto	Nós Expandidos	Score
BFS	tiny	15	502
	medium	269	442
	big	210	300

2. Busca em largura (Breadth First Search)

1. Inicia colocando um dos vértices do gráfico no final da fila.

2. Em seguida é pego o nó da frente da fila e colocado na lista de visitados.

3. Criado uma lista dos nós adjacentes desse vértice, que recebe os que ainda não foram visitados.

4. Continue as etapas dois e três até que a fila esteja vazia.

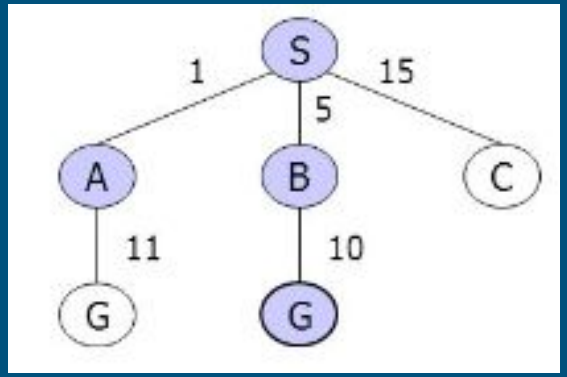
```python
def breadthFirstSearch(problem: SearchProblem):
    #pega o estado inicial do pacman, cria a fila exploração, cria a fila fronteira de nós filhos
    root = Node(problem.getStartState())
    explored = Queue()
    fronteir = Queue()
    #se não é o objetivo vai inserir na 'fronteira'
    if not problem.isGoalState(root.state):
        fronteir.push(root)
    goal = None
    #enquanto tem nó na fronteira vai explorar
    while not (fronteir.isEmpty()):
        #remove da pilha fronteira
        node = fronteir.pop()
        #coloca na pilha de explorado
        explored.push(node.state)
        #se o novo for o objetivo para
        if problem.isGoalState(node.state):
            goal = node
            break
        #retorna pra fronteirStates os dados
        fronteirStates = map(lambda x: x.state, fronteir.list)
        #pega os filhos do estado e olha
        for state,action,cost in problem.getSuccessors(node.state):
            #cria o nó, com a ação e o custo
            child = Node(state,action,cost,node)
            #se eles não foram explorados ou visitados coloca na fronteira
            if not (child.state in explored.list or child.state in fronteirStates):
                fronteir.push(child)
    solution = []
    while goal and goal.parent:
        solution.insert(0,goal.action)
        goal = goal.parent
    return solution
```

1

2

3

3. Variação da função de custo (Varying the Cost Function)



Uniform Cost Search (UCS): O algoritmo de busca de custo uniforme (Uniform Cost Search ou ucs) pode ser considerado o mesmo algoritmo de bfs se os passos para cada nó fossem os mesmos. Porém esse algoritmo expande o próximo nó com base no menor caminho existente entre eles.

Classificador	Labirinto	Nós Expandidos	Score
UCS	tiny	16	502
	medium	269	442
	big	620	300

3. Variação da função de custo (Varying the Cost Function)

1. Inicia colocando um dos vértices do gráfico no final da fila.

2. Em seguida é pego o nó da frente da fila e colocado na lista de visitados.

3. Verifica a fila de prioridade e vai avaliando os nós.

4. Se não encontrou realinha a fila de prioridades.

```python
def uniformCostSearch(problem: SearchProblem):
    #criando o nó, explorados e fronteira
    root = Node(problem.getStartState())
    explored = Queue()
    fronteir = PriorityQueue()
    # verifica se o começo ja é objetivo e insere na fronteira
    if not problem.isGoalState(root.state):
        fronteir.push(root,0)                                    1
    goal = None
    #se ha nós na fronteira remove de fronteira e insere em explorado
    while not (fronteir.isEmpty()):
        node = fronteir.pop()
        explored.push(node.state)                                2
        #se é o objetivo para
        if problem.isGoalState(node.state):
            goal = node
            break
        #retorna pra fronteirStates os dados
        fronteirStates = map(lambda x: x[2].state, fronteir.heap)
        #olhando a fronteira
        for state,action,cost in problem.getSuccessors(node.state):
            child = Node(state,action,cost,node)                                3
            #se o nó não está na fronteira nem explorado
            if not (child.state in explored.list or child.state in fronteirStates ):
                #coloca na fronteira
                fronteir.push(child,child.path_cost)
            #senão
            elif child.state in fronteirStates:
                idx = fronteirStates.index(child.state)
                if fronteir.heap[idx][0] > child.path_cost:                              4
                    fronteir.push(child,child.path_cost)
    solution = []
    while goal and goal.parent:
        solution.insert(0,goal.action)
        goal = goal.parent
    return solution
```

4. Busca A* (A star)

A Star (A*): O A* (A estrela) é um dos algoritmos path-finding mais utilizados em jogos. É útil devido às suas características, como por exemplo, pode-se alterar seu comportamento apenas modificando ou alterando heurísticas. A heurística utilizada para o cálculo da distância foi a Manhattan.

Classificador	Labirinto	Nós Expandidos	Score
A*	tiny	14	502
	medium	222	442
	big	549	300

4. Busca A* (A star)

1. Em primeiro lugar, coloque o nó inicial em OPEN e encontre seu valor f (n).

2. Em seguida é pego o nó da frente da fila e coloca nos visitados.

3. Se não é o objetivo, reorganize os seguintes e coloca na fronteira.

4. Repete os passos 3 e 4 até o objetivo.

```python
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    #pega e inicia o Nó, e cria explorados e fronteira
    root = Node(problem.getStartState())
    explored = Queue()
    fronteir = PriorityQueue()
    #se não é o objetivo pega a heurística e coloca na fronteira
    if not problem.isGoalState(root.state):
        h = heuristic(root.state, problem)
        fronteir.push(root,h)
    goal = None
    #enquanto tem nó fronteira tira da lista de nó e coloca nos explorados
    while not (fronteir.isEmpty()):
        node = fronteir.pop()
        explored.push(node.state)
        #se achou objetivo para
        if problem.isGoalState(node.state):
            goal = node
            break

        fronteirStates = map(lambda x: x[2].state, fronteir.heap)
        #percorrendo os seguintes
        for state,action,cost in problem.getSuccessors(node.state):
            child = Node(state,action,cost,node)
            h = heuristic(child.state, problem)
            #se não explorou ainda nem inseriu
            if not (child.state in explored.list or child.state in fronteirStates ):
                #insere na fronteira
                fronteir.push(child,child.path_cost+h)
            #senão
            elif child.state in fronteirStates:
                idx = fronteirStates.index(child.state)
                if fronteir.heap[idx][0] > child.path_cost+h:
                    fronteir.push(child,child.path_cost+h)
    #inserir nas soluções
    solution = []
    while goal and goal.parent:
        solution.insert(0,goal.action)
        goal = goal.parent
    return solution
```

1

2

3

Conclusão

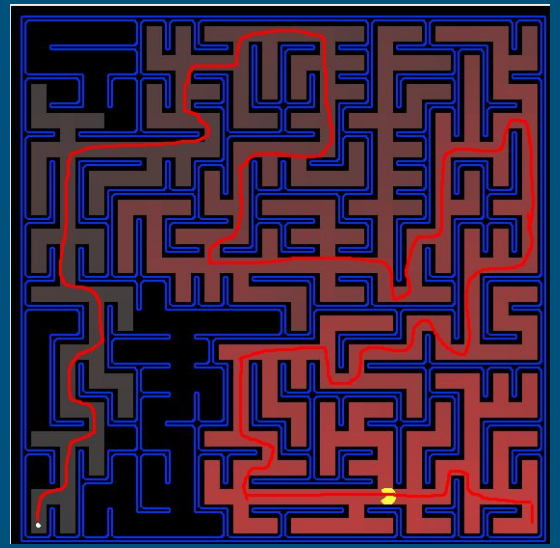
Classificador	Tamanho	Nós Expandidos	Score
DFS	tiny	15	500
	medium	146	380
	big	390	300
UCS	tiny	16	502
	medium	269	442
	big	620	300
BFS	tiny	15	502
	medium	269	442
	big	210	300
A*	tiny	14	502
	medium	222	442
	big	549	300

Em relação aos nós expandidos é possível observar que para o labirinto Tiny a diferença é irrelevante, enquanto que para os labirintos Medium e Big observou-se que o DFS obteve uma quantidade menor de nós expandidos.

Conclusão

Classificador	Labirinto	Nós Expandidos	Score
DFS	tiny	15	500
	medium	146	380
	big	390	300
UCS	tiny	16	502
	medium	269	442
	big	620	300
BFS	tiny	15	502
	medium	269	442
	big	210	300
A*	tiny	14	502
	medium	222	442
	big	549	300

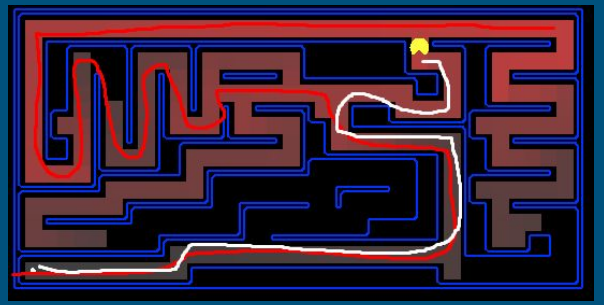
Em relação ao Score. Para o labirinto big, não houve diferença entre todos os algoritmos,muito provavelmente devido a sua configuração, pois observou-se que este labirinto havia apenas um caminho correto que levava o pacman ao destino final.



Conclusão

Classificador	Labirinto	Nós Expandidos	Score
DFS	tiny	15	500
	medium	146	380
	big	390	300
UCS	tiny	16	502
	medium	269	442
	big	620	300
BFS	tiny	15	502
	medium	269	442
	big	210	300
A*	tiny	14	502
	medium	222	442
	big	549	300

Porém para o tiny e o medium, apenas o algoritmo DFS, obteve uma desvantagem de score em relação aos outros três, devido ao fato dele preferir seguir o caminho mais longo para chegar na bolinha. Estes dois labirintos possuíam duas opções de caminhos que chegavam à bolinha.



Conclusão

Classificador	Labirinto	Nós Expandidos	Score
DFS	tiny	15	500
	medium	146	380
	big	390	300
UCS	tiny	16	502
	medium	269	442
	big	620	300
BFS	tiny	15	502
	medium	269	442
	big	210	300
A*	tiny	14	502
	medium	222	442
	big	549	300

Pode-se observar que o algoritmo menos eficiente foi o DFS. No entanto, ele foi o que menos expandiu nós para alcançar o objetivo, ou seja, ocupa menos memória.

Os outros três algoritmos obtiveram resultados semelhantes entre si.

Referências

1. https://inst.eecs.berkeley.edu/~cs188/sp22/project1/
2. https://inst.eecs.berkeley.edu/~cs188/sp22/project0/#autograding
3. https://inst.eecs.berkeley.edu/~cs188/sp22/