UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
COLEGIADO DE CIÊNCIA DA COMPUTAÇÃO

Relatório de trabalho de busca – Inteligência Artificial

Título: Análise comparativa da busca em largura e simulated annealing na resolução de problemas de labirinto

Discentes: André H. dos S. da Silva, Gabriel Y. L. Higuchi, Nathan L. S. Oliboni e Rafael P. Czerniej

Docente: Prof. Dr. Adriana Postal

Data: 09 /12/2024

---

1. DEFINIÇÃO DO PROBLEMA

O problema escolhido é o de encontrar o menor caminho em um labirinto, ou seja, sair de um ponto inicial e encontrar a saída. O labirinto é representado por uma matriz bidimensional, onde cada valor em cada posição da matriz representa a sua função no labirinto.

0: Representa um espaço vazio, por onde é permitido passar.

1: Indica uma parede ou obstáculo, bloqueando o movimento.

2: Marca a posição da saída, que é o destino final.

O labirinto é carregado a partir de um arquivo de texto contendo o tamanho da matriz, sendo eles 10x10 (fácil), 15x15 (médio), 20x20 (difícil). Na interface gráfica são mostradas as cores que representam cada valor, onde 1 = preto, 0 = branco, 2 = azul e o jogador é representado pela cor vermelha.

O usuário pode escolher entre percorrer o labirinto de forma manual ou automática utilizando o algoritmo de Simulated Annealing. O objetivo é chegar ao final em menor tempo ou percorrer uma quantidade menor de nós, ou seja, fazer o caminho até o final pelo menor caminho possível.

2. ALGORITMOS UTILIZADOS

➔ **Simulated Annealing**:

Inspirado no processo físico de recozimento na metalurgia, o Simulated Annealing é uma técnica probabilística usada para resolver problemas de otimização combinatória e contínua(**Geeks for Geeks**). Seu principal objetivo é implementar uma solução em um problema com um grande espaço de busca, visando atingir o melhor resultado possível

dentro da complicação apresentada.

Seu funcionamento consiste em implementar uma aleatoriedade controlada para em caso de ser achada uma solução considerada boa, ele consiga escapar da mesma, caso haja uma outra solução com um resultado válido e melhor, de forma simples, para escapar de máximos locais e buscar um máximo global.

- **Implementação:**

A implementação do algoritmo para o problema proposto por nosso grupo foi realizada da seguinte forma:

Foram definidas as seguintes funções

- **custo:** retorna o comprimento do caminho atual, utilizado para avaliar a qualidade de uma solução.
- **definir_caminho:** responsável por encontrar um caminho no labirinto.
- **get_vizinho:** gera variações no caminho atual, ele seleciona um ponto intermediário dentro do caminho e tenta encontrar um desvio válido para recalcular outro trajeto.
- **simulated annealing:** o caminho inicial é encontrado a partir da função **definir_caminho,** depois disso o algoritmo começa a tentar encontrar o melhor caminho atual, a cada iteração ele encontra um novo vizinho com a função **get_vizinho** e avalia seu custo, se o custo for menor que o atual ele é aceito caso contrário o vizinho pode ser aceito com base em uma probabilidade com base na diferença de custo e da temperatura atual. A temperatura inicial e a taxa de resfriamento controlam a exploração dentro do espaço de soluções. No início a temperatura é alta, o que facilita a aceitação de soluções piores, com o passar das iterações a temperatura vai diminuindo deixando o algoritmo cada vez mais seletivo. Esse processo continua até atingir o número máximo de iterações ou até encontrar a saída, então o melhor caminho é retornado. Os parâmetros utilizados são: **maze** (matriz do labirinto), **inicio** (ponto inicial do labirinto), **objetivo** (ponto final), **tempo inicial** (temperatura inicial), **taxa resfriamento** (taxa de redução da temperatura), **max iter** (número máximo de iterações)

```python
def simulated_annealing(maze, inicio, objetivo, temp_inicial=1000, taxa_resfriamento=0.99):
    # Iniciar monitoramento em paralelo
    total_memory_used = [0]  # Usar lista para compartilhamento entre threads
    def monitor_task():
        total_memory_used[0] = monitor_total_memory()
    monitor_thread = Thread(target=monitor_task, daemon=True) #inicia thread de monitoramento de RAM
    monitor_thread.start()
    max_iter = len(maze) * len(maze[0])  # Ajusta max_iter de acordo com o tamanho do labirinto
    atual = definir_caminho(maze, inicio)   # atual deve ser uma tupla (x, y)
    temperatura = temp_inicial
    melhor = atual
    melhor_custo = custo(atual, objetivo)
    caminho = atual  # Para desenhar o caminho
    iteracoes = 0
    start_time = time.perf_counter()
    while iteracoes < max_iter and temperatura > 1:
        vizinho = get_vizinho(maze, atual)
       # next_pos = random.choice(vizinhos)
        if not isinstance(vizinho, list):
            raise ValueError(f"vizinho deve ser do tipo lista, mas é do tipo {type(vizinho)}")
        # Calcular o custo do próximo estado
        next_custo = custo(vizinho, objetivo)
        # Calcular a diferença de custo
        delta_custo = next_custo - melhor_custo
        # Decidir se move para o próximo estado
        if delta_custo < 0 or random.uniform(0, 1) < math.exp(-delta_custo / temperatura):
            atual = vizinho
            # Atualizar melhor solução
            if next_custo < melhor_custo:
                melhor = vizinho
                melhor_custo = next_custo
        # Resfriar a temperatura
        temperatura *= taxa_resfriamento
        iteracoes += 1
        #print(f"Iteração: {iteracoes}, Temperatura: {temperatura}, Custo: {melhor_custo}")
        if atual == objetivo:
            break
    global stop_monitoring
    stop_monitoring = True
    monitor_thread.join()
    print(f"Memória total acumulada durante a execução: {total_memory_used[0]} Bytes")
    end_time = time.perf_counter()
    exec_ms = (end_time - start_time) * 1000
    return melhor, iteracoes, exec_ms
```

Função: simulated_annealing

➔ **Busca em Largura**:

O algoritmo de **Busca em Largura(Breadth-First Search)** explora primeiro todos os caminhos de menor profundidade, garantindo que o primeiro caminho encontrado será o mais curto.

Deque(Fila): Armazena os caminhos parciais durante a busca.

Visited: Armazena as posições já visitadas para evitar ciclos.

Direções: As direções válidas são definidas como combinações de deslocamento(cima, baixo, esquerda, direita)

Etapas:

O algoritmo explora todos os caminhos de forma sistemática, verificando em cada iteração se o objetivo foi alcançado. Verifica a cada iteração os vizinhos disponíveis para cada direção válida, se o vizinho está dentro dos limites da matriz do labirinto, se não é uma parede e se ainda não foi visitado, se todos esses requisitos forem cumpridos esse vizinho é adicionado ao caminho, que é adicionado à fila de caminhos. A busca termina quando o algoritmo encontra o objetivo, e retorna o caminho percorrido.

```python
def busca_em_largura(labirinto, inicio, objetivo):
    nos_gerados=0
    nos_visitados=0

    # Iniciar monitoramento em paralelo
    total_memory_used = [0]  # Usar lista para compartilhamento entre threads
    def monitor_task():···
    monitor_thread = Thread(target=monitor_task, daemon=True) #inicia thread de monitoramento de RAM
    monitor_thread.start()

    atual = inicio
    iteracoes = 0
    queue = deque([[inicio]])
    visited = set([inicio])
    r, c = len(labirinto), len(labirinto[0])
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    start_time = time.perf_counter()
    while queue:
        iteracoes = iteracoes + 1
        path = queue.popleft()
        nos_visitados = nos_visitados + 1
        atual = path[-1]
        for dx, dy in directions:
            if labirinto[atual[1]][atual[0]] == 2:
                end_time = time.perf_counter()
                time.sleep(1) #para ter tempo de calcular o uso de RAM
                global stop_monitoring
                stop_monitoring = True
                monitor_thread.join()
                print(f"Memória total acumulada durante a execução: {total_memory_used[0]} Bytes")
                exec_ms = (end_time - start_time) * 1000
                print("nos gerados durante o processo: ",nos_gerados)
                print("nos viistados durante o processo: ", nos_visitados)
                return path, iteracoes, exec_ms
            vizinho = (atual[0] + dx, atual[1] + dy)
            if (0 <= vizinho[0] < c and 0 <= vizinho[1] < r and labirinto[vizinho[1]][vizinho[0]] != 1 and vizinho not in visited):
                visited.add(vizinho)
                novas_posicoes = list(path)
                nos_gerados = nos_gerados + len(novas_posicoes)
                novas_posicoes.append(vizinho)
                queue.append(novas_posicoes)
    return None
```

Função busca_em_largura:

3. COMPARAÇÃO

Para a comparação foram realizados 20 testes para cada um dos algoritmos, tanto Simulated annealing quanto Busca em Largura, para o jogo foram criados três labirintos constituídos por matrizes de 0, 1 e 2. Cada labirinto tendo uma dificuldade, como citado no início deste relatório, para se obter uma métrica mais aprofundada, as 20 execuções dos algoritmos foram realizadas no labirinto cuja complexidade foi definida como difícil e esses foram os resultados médios obtidos:

Caracteristica avaliada	Busca em largura	Simulated annealing
Número de nós gerados	3539	320
Número de nós visitados	115	307,55
Tempo de execução(ms)	0,7109	129,15
Memória utilizada no processo(Bytes)	52.069.171,2	5.989.908,48

4. RESULTADOS E DISCUSSÃO

Analisando os resultados, observa-se que a Busca em Largura gera um número significativamente maior de nós, pois explora todas as possibilidades possíveis para encontrar a solução. Por outro lado, o Simulated Annealing é mais econômico, focando em uma exploração mais direcionada, o que resulta em um consumo menor de recursos computacionais e na possibilidade de encontrar soluções aceitáveis em um tempo reduzido.

Em termos de memória, a Busca em Largura apresenta um custo muito mais elevado, pois precisa armazenar todos os nós gerados para garantir que cada caminho seja avaliado de forma exaustiva. Já o Simulated Annealing é mais eficiente nesse aspecto, mantendo apenas os estados necessários durante sua execução, o que o torna mais viável em sistemas com recursos limitados.

No entanto, a economia do Simulated Annealing traz consigo algumas implicações. Como é baseado em uma abordagem probabilística e exploratória, ele não garante encontrar a solução ótima, especialmente em labirintos grandes ou com estruturas complexas. A Busca em Largura, em contrapartida, por ser exaustiva, sempre encontra a solução ótima (se uma solução existir), o que pode ser uma vantagem em situações onde a qualidade da solução é mais importante que o tempo ou os recursos utilizados.

5. CONCLUSÕES

A escolha dos dois algoritmos depende do contexto de utilização e da importância do problema a ser resolvido. Se o objetivo for o desempenho e a solução não for a ideal, o **simulated annealing** pode ser a opção mais adequada. Por outro lado, para problemas importantes onde a melhor solução deve ser encontrada, a **busca em largura** é o método mais confiável se houver memória e tempo suficientes para suportar sua execução.

Também é possível investigar soluções híbridas que combinam bem os dois métodos. Por exemplo, usar a heurística para orientar a busca em largura pode reduzir o número de nós criados, enquanto o ajuste da temperatura e dos parâmetros introduzidos no **simulated annealing** pode aumentar o número de nós para a melhor solução.

Por fim, vale ressaltar que o foco do algoritmo também é afetado pelo tamanho da busca, labirintos grandes podem destacar a eficácia dos dois métodos e tornar interessante experimentar em várias situações para verificar os resultados, ou seja, realizar mais testes com labirintos maiores e mais complexos.

BIBLIOGRAFIA

GEEKSFORGEEKS. **Rat in a Maze Problem when movement in all possible directions is allowed**. Disponível em: https://www-geeksforgeeks-org.translate.goog/rat-in-a-maze-problem-when-movement-in-all-possible-directions-is-allowed/?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em: 08 dez. 2024.

GEEKSFORGEEKS. **Shortest path in a Binary Maze**. Disponível em: https://www-geeksforgeeks-org.translate.goog/shortest-path-in-a-binary-maze/?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em: 08 dez. 2024.

GEEKSFORGEEKS. **Implement Simulated Annealing in Python**. Disponível em: https://www.geeksforgeeks.org/implement-simulated-annealing-in-python/?ref=gcse_outind. Acesso em: 08 dez. 2024.

documentação do módulo pygame. Disponível em: https://www.pygame.org/docs/. Acesso em: 10 dez. 2024.