Uma imagem com Tipo de letra, Gráficos, design gráfico, logótipo

Descrição gerada automaticamente

**Sprint 3**

**Turma 3NA-Grupo 76**

1200583 – João Carrinho

1200586 – Mário Borja

1200618 – Jorge Cunha

1210815 – João Barros

**Professor:**

Francisco Samuel Paiva da Silva, FPS

**Unidade Curricular:**

Algoritmia Avançada (ALGAV)

**Data: 02/01/2024**

**Explicação dos Métodos em Prolog relativo ao Sprint 2**

Este documento apresenta uma explicação detalhada dos métodos utilizados em um programa Prolog para a gestão de edifícios e navegação espacial.

**Declarações Dinâmicas**

**:-dynamic m/5**

Esta declaração permite que fatos do tipo m(Building, Floor, Column, Line, Value) sejam alterados em tempo de execução. É usado para representar uma matriz de valores em um edifício e andar específicos.

**:-dynamic pisos/2**

Esta declaração não é utilizada no código fornecido, mas geralmente seria usada para permitir a modificação dinâmica de fatos com dois argumentos relacionados a pisos.

**:-dynamic ligacel/3**

Permite a modificação dinâmica de fatos do tipo ligacel(Cell1, Cell2, Cost), representando a ligação e o custo entre duas células (ou espaços) em um edifício.

**:-dynamic elevador/2**

Permite a modificação dinâmica de fatos do tipo elevador(Building, Floors), representando a existência de um elevador em um edifício e os andares que ele atende.

**Criação de Matriz**

O método cria\_matriz é utilizado para criar uma matriz com todos os valores inicializados a 0. Ele começa por pedir ao usuário o número de colunas e linhas, e depois utiliza o método cria\_matriz\_0 para preencher a matriz.

**Criação de Grafos**

O método graphs e suas funções auxiliares (graphs\_aux, create\_passageway\_connections, create\_elevator\_connections) são usados para criar ligações entre as células da matriz, representando passagens e conexões de elevador entre diferentes andares e edifícios.

**Busca em Profundidade (DFS)**

O método dfs e suas variantes (all\_dfs, semiBetter\_dfs, better\_dfs1) implementam o algoritmo de busca em profundidade para encontrar caminhos entre dois pontos na matriz.

**Busca em Largura (BFS)**

O método bfs implementa o algoritmo de busca em largura para encontrar o caminho mais curto entre dois pontos.

**A\* Search**

O método aStar implementa o algoritmo A\* para encontrar o caminho mais eficiente entre dois pontos, considerando tanto o custo quanto a estimativa de distância restante.

**Explicação dos Métodos em Prolog relativo ao Sprint 3**

**Arquivo criacao\_tarefas.pl**

**Declaração de Predicados Dinâmicos**

**:- dynamic tarefa\_existentes/2.**

**:- dynamic origem\_tarefa/2.**

**:- dynamic destino\_tarefa/2.**

**:- dynamic lista\_tarefas/1.**

Essas linhas declaram os predicados tarefa\_existentes, origem\_tarefa, destino\_tarefa, e lista\_tarefas como dinâmicos. Isso significa que cláusulas podem ser adicionadas ou removidas dinamicamente durante a execução do programa.

**Buscar Todas as Tarefas**

**buscar\_todas\_tarefas(ListaTarefas) :-**

**findall(Nome, tarefa\_existentes(Nome, \_), ListaTarefas),**

**assertz(lista\_tarefas(ListaTarefas)).**

Este predicado utiliza findall para coletar todos os nomes de tarefas existentes e armazená-los em ListaTarefas. Em seguida, usa assertz para adicionar essa lista à base de dados dinâmica.

**Adicionar Tarefa Completa**

**adicionar\_tarefa\_completa(Nome, Origem, Destino) :-**

**aStar(Origem,Destino,\_,Custo), adicionar\_tarefa(Nome, Custo),**

**origem\_tarefa\_criada(Nome, Origem), destino\_tarefa\_criada(Nome, Destino).**

Este predicado calcula o custo de uma tarefa do ponto de Origem para Destino usando o algoritmo aStar. Ele então arredonda esse custo, adiciona a tarefa com seu custo à base de dados dinâmica e armazena as informações de origem e destino da tarefa.

**Adicionar Tarefa, Origem e Destino**

**adicionar\_tarefa(Nome, Custo) :-**

**assertz(tarefa\_existentes(Nome, Custo)).**

**origem\_tarefa\_criada(Nome,PontoOrigem):-**

**assertz(origem\_tarefa(Nome,PontoOrigem)).**

**destino\_tarefa\_criada(Nome,PontoOrigem):-**

**assertz(destino\_tarefa(Nome,PontoOrigem)).**

Estes predicados são usados para adicionar individualmente informações sobre a tarefa (nome e custo), origem e destino à base de dados dinâmica.

**Arquivo criacao\_tarefas.pl**

**Cálculo do Custo de uma Única Tarefa**

**calcular\_custo\_unica\_tarefa(Tarefa, CustoTarefaTotal)**

Calcula o custo total de uma única tarefa, incluindo o custo de deslocamento entre sua origem e destino.

**Cálculo Recursivo de Custo de Caminho**

**calculo\_custo\_caminho\_tarefas([], Acumulado, Acumulado).**

Calcula o custo acumulado de um caminho de tarefas. Há casos para listas vazias, uma única tarefa, e listas com duas ou mais tarefas.

**Cálculo do Menor Custo entre Permutações**

**calcular\_minimo\_custo\_permutacoes([], MinSequence, MinCost, MinSequence, MinCost).**

Encontra a sequência de tarefas com o menor custo total entre todas as permutações.

**Início do Processo de Cálculo**

**calcular\_custo\_lista\_tarefa\_inicial([], \_, \_).**

Inicia o processo de cálculo do custo para a primeira sequência de tarefas e, em seguida, encontra o menor custo entre todas as sequências.

**Encontrar e Calcular o Menor Custo**

**gerar\_e\_calcular\_minimo\_custo(ListaTarefas, Lista, MenorCusto)**

Gera todas as permutações de uma lista de tarefas e encontra a permutação com o menor custo total.

**Big O**

Ao analisar a complexidade computacional (Big O) dos métodos fornecidos, considerando que a complexidade do algoritmo aStar é

O(E), onde E representa o número de arestas no grafo utilizado pelo algoritmo, observamos as seguintes complexidades:

**calcular\_custo\_lista\_tarefa(ListaTarefas, CustoTotal):**

Este método envolve uma chamada recursiva de calculo\_custo\_caminho\_tarefas.

A sua complexidade depende do número de tarefas na lista (n) e da complexidade de cálculo do custo entre tarefas.

Complexidade: O(n×E)

Justificação: Cada chamada recursiva processa uma tarefa e calcula o custo entre tarefas, o que é O(E).

Este processo é repetido para cada tarefa na lista.

**rearranjar\_lista(Lista, Permutação)**

Este método gera permutações de uma lista. A geração de permutações tem uma complexidade conhecida.

Complexidade: O(n!)

Justificação: Para uma lista de tamanho n, existem n! (fatorial de n) permutações possíveis.

**gerar\_permutacoes(ListaTarefas, ListaDeListasModificadas)**

Este método utiliza rearranjar\_lista para gerar todas as permutações de uma lista de tarefas e armazená-las.

Complexidade: O(n!)

Justificação: A complexidade é dominada pela geração de permutações, que é O(n!).

**calcular\_minimo\_custo\_permutacoes(Sequências, MinSequence, MinCost)**

Este método percorre todas as permutações para encontrar a de menor custo.

Complexidade: O(n!×n×E)

Justificação: Para cada permutação (e há n! delas), o custo é calculado percorrendo

n tarefas com cada cálculo de custo sendo O(E).

**Conclusão**

Ao analisar os métodos no sistema de gestão de tarefas, considerando que a complexidade do algoritmo aStar é O(E), observamos variações significativas na complexidade computacional dos diferentes métodos.

O método calcular\_custo\_lista\_tarefa apresenta uma complexidade de O(n×E), refletindo a combinação do número de tarefas com a complexidade de calcular o custo entre elas. Em contrapartida, os métodos relacionados à geração e avaliação de permutações, como rearranjar\_lista, gerar\_permutacoes, e calcular\_minimo\_custo\_permutacoes, exibem complexidades significativamente maiores, alcançando O(n!) para a geração de permutações e O(n!×n×E) para a avaliação do custo mínimo entre todas as permutações. Estas análises indicam um aumento exponencial na complexidade com o acréscimo do número de tarefas, destacando desafios potenciais em termos de escalabilidade e eficiência para listas extensas de tarefas.

**Tabela de análise da complexidade**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nº de Tarefas | | Nº lig. | Caminho | Custo | Tempo |
| 1 | 152 | | [tarefaG] | 10 | 0 ms |
| 2 | 152 | | [tarefaT,tarefaG] | 22 | 1 ms |
| 3 | 152 | | [tarefaT,tarefaZ,tarefaG] | 31 | 4 ms |
| 4 | 152 | | [tarefaT,tarefaZ,tarefaX,tarefaG] | 35.2 | 20 ms |
| 5 | 152 | | [tarefaT,tarefaL,tarefaZ,tarefaX,tarefaG] | 41 | 157 ms |
| 6 | 152 | | [tarefaG,tarefaP,tarefaT,tarefaL,tarefaX,tarefaZ] | 48 | 1215 ms |
| 7 | 152 | | [tarefaG,tarefaP,tarefaT,tarefaL,tarefaX,tarefaZ,tarefaR] | 57.4 | 10183 ms |
| 8 | 152 | | Error | error | error |

**Observações:**

Aumento Exponencial do Tempo: Observa-se um aumento exponencial no tempo de execução à medida que o número de tarefas aumenta. Este é um comportamento típico de algoritmos de força bruta, especialmente aqueles que envolvem permutações, como neste caso. Por exemplo, o tempo aumenta de 1 ms para 2 tarefas para 10.183 ms para 7 tarefas.

Limitações Práticas: Com apenas 8 tarefas, o algoritmo já enfrenta um erro ("Error"). Isto sugere que o algoritmo não é escalável para um maior número de tarefas, possivelmente devido a limitações de memória ou de tempo de processamento.

Número de Ligações Constante: O número de ligações mantém-se constante (152) independentemente do número de tarefas, indicando que o número de ligações não influencia diretamente a complexidade do problema neste contexto específico.

Custo Crescente: O custo total das tarefas aumenta à medida que mais tarefas são adicionadas, o que é esperado, mas o aumento não é tão acentuado quanto o do tempo de execução.

**Conclusão:**

A análise da tabela revela que, embora o algoritmo de força bruta possa encontrar a solução ótima para um pequeno número de tarefas, a sua aplicabilidade prática é severamente limitada pela sua ineficiência e falta de escalabilidade. O crescimento exponencial do tempo de execução com o aumento do número de tarefas torna o algoritmo impraticável para cenários com mais do que algumas tarefas. Este é um exemplo emblemático das limitações dos algoritmos de força bruta, em que a garantia de encontrar a solução ótima é sobreposta pelo custo computacional proibitivo para problemas de tamanho realista. Em contextos práticos, torna-se essencial considerar algoritmos mais eficientes, como aqueles baseados em heurísticas ou métodos aproximados, que podem fornecer soluções suficientemente boas num tempo significativamente reduzido.

**Desenvolvimento:**

**1. Parametrização da Condição de Término:**

O algoritmo genético em análise utiliza principalmente o número de gerações como critério de término.

A função gerações (NG) define o número total de gerações após o qual o algoritmo cessará a execução, como também utiliza a taxa de mutação entre a população, o cruzamento e o número de tarefas escolhido.

**2. Adaptação para o Planeamento da Trajetória de Robôs:**

O algoritmo incorpora o uso do algoritmo A\* (Astar) para calcular distâncias e efetuar a interligação entre edifícios. Esta integração é fundamental para a adaptação do algoritmo genético ao contexto específico de planeamento de trajetórias de robôs em ambientes complexos. A inclusão do A\* permite calcular rotas eficientes levando em conta a localização das tarefas e a configuração dos edifícios, elementos cruciais para a navegação robótica.

**3. Aleatoriedade no Cruzamento entre Indivíduos:**

O algoritmo implementa aleatoriedade no processo de cruzamento de duas formas: através da geração de pontos de cruzamento aleatórios e da decisão de realizar ou não o cruzamento com base numa probabilidade definida. A função gerar\_pontos\_cruzamento(P1,P2) é utilizada para determinar os pontos de cruzamento dentro dos indivíduos de forma aleatória, enquanto a decisão de cruzamento baseia-se numa probabilidade (prob\_cruzamento(Pcruz)), garantindo assim diversidade genética na população, não esquecendo a Seleção por Roleta.

**4. Seleção por Roleta:**

A técnica de seleção por roleta é empregada para escolher indivíduos para a reprodução. Esta técnica assegura que indivíduos com maior aptidão tenham uma maior probabilidade de serem selecionados, embora indivíduos com menor aptidão também possam ser escolhidos. Esta abordagem ajuda a manter um equilíbrio entre a exploração de boas soluções já encontradas e a exploração de novas áreas do espaço de soluções.

**Conclusão:**

O algoritmo genético analisado demonstra ser uma ferramenta robusta para o planeamento da trajetória de robôs em edifícios conectados. A inclusão do algoritmo A\* para cálculo de distâncias e a implementação de técnicas como a seleção por roleta e a introdução de aleatoriedade nos processos de cruzamento contribuem significativamente para a sua eficácia. A adição de condições de término adicionais poderia potencialmente melhorar a eficiência do algoritmo, representando uma área para futuras melhorias.

**Estudo do Estado da Arte de métodos de planeamento de trajetória e movimento de robots que se movimentam numa superfície plana com obstáculos**

O estudo do estado da arte em métodos de planeamento de trajetória e movimento de robôs que se movimentam em superfícies planas com obstáculos revela avanços significativos, especialmente na integração de algoritmos e técnicas de otimização avançada.

Um exemplo notório é o desenvolvimento do algoritmo "Graphs of Convex Sets (GCS) Trajectory Optimization" pela equipa do MIT CSAIL. Este sistema inovador combina a pesquisa de gráficos com a otimização convexa para criar um método de planeamento de movimento escalável e sem colisões. O algoritmo é capaz de navegar eficientemente em ambientes complexos, otimizando a trajetória do robô e mostrando-se superior em relação a planejadores comparáveis. O sucesso das demonstrações reais e simulações reforça o potencial deste algoritmo em aplicações como manufatura e organização doméstica, onde a coordenação de múltiplos robôs é crucial​​.

Outra abordagem é o uso do método de planeamento de trajetória baseado em campo potencial artificial (APF) para um robô móvel omnidirecional (MWOMR). Este método atualiza iterativamente a trajetória de referência considerando a posição do robô, dos obstáculos e do alvo, sendo eficaz mesmo com obstáculos dinâmicos. Além disso, foi implementado um algoritmo de controle DITSM (Discrete Integral Terminal Sliding Mode) que assegura a precisão no seguimento da trajetória planeada, mantendo robustez contra incertezas e perturbações do sistema​​.

**Conclusão:**

Estes avanços mostram um crescente enfoque na otimização e no controle inteligente para lidar com ambientes complexos e dinâmicos na área de planeamento de trajetória de robôs, evidenciando um campo em rápida evolução e de grande relevância prática.

**Referencias:**

MIT Computer Science and Artificial Intelligence Laboratory (CSAIL). "A new optimization framework for robot motion planning," MIT CSAIL, 2023. [Online]. Disponível em: https://www.csail.mit.edu/news/new-optimization-framework-robot-motion-planning. [Acessado em: 11-12-2023].

Y. Zhang, Z. Yi, L. Xu, e Y. Zhao, "Obstacle-avoidance trajectory planning and sliding mode-based tracking control of an omnidirectional mobile robot," Frontiers in Computer Science and Technology Engineering, vol. 1, pp. 1-10, 2023. [Online]. Disponível em: https://www.frontiersin.org/articles/10.3389/fcteg.2023.1135258/full. [Acessado em: 11-12-2023].

Grupo 76, "Estudo da Arte," [Local]. Disponível em: Estudo\_da\_Arte .