ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Geração de Trajetórias com Robôs e Algoritmos Genéticos

Relatório Sprint C

Licenciatura em Engenharia Informática

2023-2024

André Barros	1211299
Carlos Lopes	1211277
Ricardo Moreira	1211285
Tomás Lopes	1211289
Tomás Russo	1211288

Unidade Curricular

Algoritmia Avançada

2 de janeiro de 2024

Conteúdo

1 Introdução

O objetivo do presente relatório é explorar os resultados obtidos na geração de trajetórias de robôs para a execução de tarefas.

Desta forma, vamos explorar as soluções que obtivemos ao aplicar um algoritmo genético que tem como objetivo obter uma solução eficiente para a geração de uma sequência de tarefas a serem realizadas por um robô.

Para analisar as soluções, o presente relatório é complementado com uma análise do estado da arte no domínio da geração de trajetórias de robôs para a execução de tarefas, de forma a ser possível analisar as principais diferenças entre a solução utilizada e as mais avançadas do momento.

2 Algoritmo Genético

2.1 Base do Conhecimento

O algoritmo genético (AG) é utilizado para encontrar soluções em problemas complexos de otimização onde é inviável encontrar a melhor solução de forma exaustiva. É um método inspirado na teoria da evolução biológica, que começa com uma população de soluções aleatórias, cada solução é avaliada em relação a um critério de desempenho, e os indivíduos mais promissores são selecionados para reprodução. Durante esse processo, ocorre recombinação e mutação, introduzindo diversidade na população. Os descendentes resultantes, juntamente com parte da população original, formam a próxima geração. Este ciclo de seleção, reprodução e substituição é repetido ao longo de várias gerações até que um critério de paragem seja alcançado, por exemplo, através da convergência para uma solução ótima.

Para a realização do trabalho referente ao *Sprint* C, da Unidade Curricular de Algoritmia Avançada, é crucial ter uma definição clara dos conceitos de tarefas de um robô.

A empresa "RobDroneGO, S.A"opera num Campus de edifícios, cada edifício é constituído por um ou mais pisos e um elevador. Os diferentes edifícios do Campus estão ligados através de passagens. As tarefas, neste contexto, podem ser simplificadas na forma de um ponto inicial e um ponto final, correspondentes ao local em que o robô deve começar a tarefa e onde deve acabar. Assim, para fazer a representação de uma tarefa, é necessário recorrer ao facto, **t/3**:

t(id_tarefa, inicio, fim).

id_tarefa - Corresponde ao identificador da tarefa.
inicio - Corresponde à célula do inicio da tarefa.
fim - Corresponde à célula do fim da tarefa.

*Nota: Tanto o **inicio** quanto o **fim** são factos do tipo **cel(código piso, x, y)***

2.2 Geração de todas as sequências de tarefas através de permutações

Uma das maneiras de obter uma solução para o problema da obtenção da sequência ótima de tarefas é através da geração de todas as permutações possíveis entre um determinado conjunto de tarefas e posterior escolha da melhor solução. Esta abordagem, mesmo sendo a mais eficaz, garantindo que obtemos sempre a melhor solução possível, é extremamente ineficiente, dado que o algoritmo de geração das permutações de um conjunto de n tarefas tem uma complexidade temporal de ordem fatorial - aproximadamente $O(n!)$.

Mesmo tendo conhecimento da ineficiência deste algoritmo, iremos implementá-lo de maneira a compreender os limites temporais do mesmo, tendo como resultado um estudo de complexidade que analisa o número máximo de tarefas que este algoritmo consegue processar em tempo útil.

O algoritmo, implementado através do predicado **gera_best_bruteforce**, começa por encontrar todas as permutações do conjunto de tarefas definidas através do facto **t/3**. Para tal, utiliza o predicado nativo do Prolog **permutation/2**, que nos dá uma lista contendo todas as permutações possíveis do conjunto de tarefas existentes.

Depois de obtidas todas as permutações, podemos afirmar que temos uma população de $n!$ indivíduos, sendo n o número de tarefas existentes. Sabendo isso, para chegar à solução final basta obter o indivíduo com melhor *fitness* da população. Para isso, utilizamos primeiramente o predicado **avalia_populacao/2**, que calculará os valores de *fitness* para cada indivíduo da nossa população - de sublinhar que o *fitness* de cada indivíduo, no contexto do nosso problema, corresponde à distância total que um robô tem que percorrer utilizando a sequência descrita por esse individuo. Depois de obtidos os valores do *fitness*, podemos ordenar de forma crescente a população, tendo como critério esses mesmos valores, através do predicado **ordena_populacao/2**. Por fim, basta obter o primeiro elemento dessa lista, de maneira a finalmente encontrar a combinação de tarefas com menor distância percorrida.

2.3 Aleatoriedade no cruzamento

O cruzamento entre indivíduos da população é efetuado com recurso ao operador de cruzamento, *Order Crossover*, visto que este operador é útil quando os indivíduos são representados por sequências de elementos, ou seja, as diversas a ordem em que o robô deve executar a sequência de tarefas.

No algoritmo genético fornecido, o processo de cruzamento é executado sobre os indivíduos da população. Este procedimento consiste em selecionar dois ”pais”da população e aplicar o operador de cruzamento para criar descendentes. Os descendentes gerados, por sua vez, contribuem para a diversidade genética da população, influenciando a busca por soluções mais eficientes.

Além do cruzamento, a etapa de mutação desempenha um papel complementar na introdução de variação genética. A mutação é aplicada a indivíduos selecionados aleatoriamente, altera alguns elementos das sequências de tarefas.

2.4 Seleção da nova geração

O algoritmo genético fornecido inicialmente substituía a população atual pelos seus descendentes, não existindo assim qualquer garantia de que os melhores indivíduos da geração anterior carregavam os ”genes”para as gerações seguintes. Desta forma, foi sugerido que as duas melhores soluções passassem automaticamente para a próxima geração, e assim sucessivamente.

Para solucionar este problema obtemos, inicialmente, são obtidos os dois melhores indivíduos da geração anterior. De seguida, fazemos o cruzamento e a mutação dessa mesma população, e, após esta ser avaliada e ordenada garantimos que na nova lista não existem elementos repetidos recorrendo ao facto: **remove_repetidos/2**.

remove_repetidos(Pop, NovaPop).

Pop - Corresponde à lista em que vamos retirar os elementos repetidos.
NovaPop - Corresponde à lista sem os elementos repetidos.

Após esta operação, a lista pode exceder o tamanho máximo de indivíduos na população. Desta forma, usamos o predicado **remove_ultimos_n/3**. Assim sendo, os piores indivíduos ”extra”da geração são removidos, mantendo a dimensão da população no tamanho especificado no facto **populacao/1**.

remove_ultimos_n(Pop, N, NovaPop).

Pop - Corresponde à lista em que vamos retirar os últimos N elementos.
N - Corresponde ao número de elementos a retirar.
NovaPop - Corresponde à nova lista gerada.

2.5 Condições de paragem

De forma a explorar algumas implementações diferentes do AG, foi sugerida a implementação de várias condições de paragem. Desta forma, além da condição que foi fornecida inicialmente (número de gerações), foram implementadas também as condições de paragem que têm em conta: limite de tempo e estabilização da população entre gerações.

O predicado que valida se o tempo limite para a execução já foi atingido é o **gera_lim_time**. Neste predicado é efetuada, a cada iteração, a verificação do tempo total de execução, comparando-o a um valor de tempo definido arbitrariamente no facto **lim_time**. Caso este limite seja alcançado, o algoritmo termina, garantindo assim que este não ultrapasse limites ”aceitáveis”de execução.

Esta alternativa de paragem é bastante interessante quando não queremos depender de fatores do algoritmo para determinar o tempo de resposta do AG. Na presente solução, é esta condição de paragem que é utilizada na integração com o restante projeto de forma a garantir que não excedemos **2 segundos** de espera quando pretendemos obter uma sequência de tarefas.

Também foi implementada a condição de paragem que valida a estabilização da população. O predicado que valida se a população estabilizou é o **gera_estab**. Este predicado vai verificar se as gerações mais recentes não apresentam mudanças na sua população. Assim, se a população estiver estabilizada, o algoritmo termina.

2.6 Adaptação do AG para o Planeamento da Trajetória do Robot dentro de edifícios conectados considerando várias tarefas

Para adaptar o algoritmo genético fornecido, tendo em conta tarefas que envolvam troca de pisos, foi necessário utilizar o predicado desenvolvido no *Sprint* anterior **caminho_celulas_edificios/4**. Desta forma, conseguimos calcular a distância entre duas tarefas mesmo estando em edifícios diferentes.

Para além disso, foi necessário acrescentar dois factos, **distancias_robot_tarefa/2** e **distancias_tarefa_robot/2.** visto o robô tem um ponto inicial que representa o local onde inicia a execução das tarefas. Deste modo, o **distancias_robot_tarefa/2** irá guardar a distância do robô ao ponto inicial da tarefa e o **distancias_tarefa_robot/2.** a distância do ponto final da tarefa ao robô. Assim sendo, podemos calcular o *fitness* dos individuos da forma correta tendo em conta a deslocação inicial que o robô deve ter para iniciar a sequência e a final para voltar ao seu ponto inicial. Apenas desta forma é possível calcular corretamente o *fitness* de um indivíduo da população.

O predicado que carrega as tarefas para memória, **load_tasks**, irá, , tendo em conta o tipo de tarefa (Pick Delivery, Surveillance), calcular o custo entre as tarefas, o custo robô-tarefa e tarefa-robô e guardá-los em factos adequados. Desta forma, ao executar o processo de avaliação do *fitness* dos indivíduos não é necessário voltar a calcular este custo (já que é uma tarefa pesada de computar), assim sendo, o AG necessita apenas de consultar os factos criados neste passo.

Posteriormente, ao avaliar a população **avalia_populacao/2**, na sua primeira iteração, é usado o facto **distancias_robot_tarefa/2** adequado para calcular o custo, e, acrescentar assim à avaliação do indivíduo esse resultado. Já para o último elemento da lista, irá ser utilizado o **distancias_tarefa_robot/2** para calcular o custo correto da última tarefa, acrescentando também o resultado à avaliação do indivíduo.

2.7 Análise de complexidade

Esta secção tem como objetivo fazer um estudo da complexidade do algoritmo desenvolvido para determinar a sequência de tarefas com menor distância a percorrer, obtida através da geração de todas as permutações e posterior escolha da melhor.

Como referido anteriormente, o algoritmo desenvolvimento tem uma complexidade temporal de ordem fatorial - aproximadamente $O(n!)$ - dada a necessidade de gerar todas as permutações de um conjunto de n tarefas.

De maneira a fazer um estudo empírico da complexidade do algoritmo, e entender o limite de tarefas que consegue processar em tempo útil, definimos um conjunto de dez tarefas de teste, através do predicado **t/3**.

Para que possamos observar e registar o tempo de processamento do algoritmo, foi utilizado o predicado **get_time/1** para obter os tempos de inicio e finalização do algoritmo.

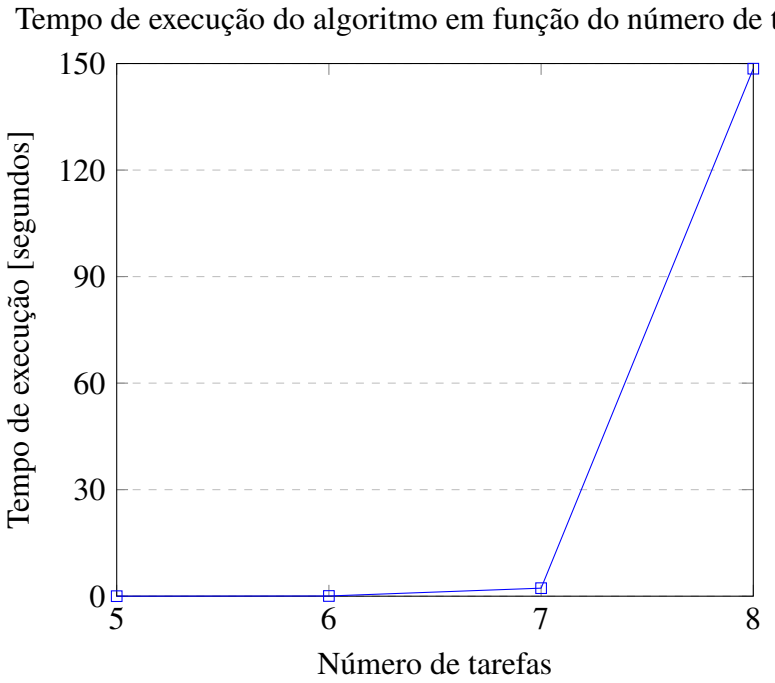
Iniciamos o estudo começando por executar o predicado **gera_best_bruteforce** tendo declaradas as cinco primeiras tarefas do conjunto das dez tarefas definidas; depois, anotamos o tempo de processamento do algoritmo para as cinco tarefas; repetimos a execução, ainda com as cinco tarefas, por mais duas vezes, anotando os tempos de processamento; avançamos para o próximo conjunto de tarefas, adicionando mais uma tarefa às declaradas anteriormente; e repetimos o processo até que o algoritmo ultrapasse o tempo de processamento de **duas horas**.

Os resultados obtidos foram os seguintes:

Nº de Tarefas	1º Execução (s)	2º Execução (s)	3º Execução (s)	Duração Média (s)
5	0.0061	0.0071	0.0054	0.0062
6	0.0641	0.0559	0.0587	0.0597
7	2.1925	2.3056	2.2858	2.2614
8	153.1170	142.3263	150.2079	148.5505
9	>7200	-	-	-

Tabela 1: Tempos de execução do algoritmo

Colocando os dados anteriores em gráfico, ficamos com o seguinte:



Observamos que o tempo de processamento aumenta de maneira exponencial a partir de $n = 8$ tarefas, pelo que com apenas $n = 9$ tarefas o tempo ultrapassa já as duas horas de execução.

Desta forma, concluímos que a utilização deste algoritmo numa situação real não é viável, sendo necessária a implementação de um algoritmo de geração de soluções recorrendo a outros métodos de processamento mais eficientes, como, por exemplo, um algoritmo genético.

3 Estado da Arte

Nos dias atuais, a robótica tem passado por uma notável evolução, com o crescente desenvolvimento de robôs capazes de realizar tarefas de forma autónoma. A capacidade de se deslocarem de forma eficiente em ambientes com obstáculos e a geração de trajetórias tornaram-se conceitos fundamentais para a aplicação prática destes robôs, tanto em contextos industriais e domésticos, quanto em contextos académicos e científicos [1].

Uma das áreas que impulsionou significativamente o avanço na robótica foi a Inteligência Artificial (IA) [2]. A incorporação de IA permite que os robôs aprendam com o ambiente ao seu redor, adaptando-se a diferentes situações e otimizando as trajetórias de maneira a que se consiga utilizar o mínimo de tempo e energia [3]. Esta capacidade de aprendizagem contínua e adaptação em tempo real representa um salto significativo na eficiência operacional dos robôs autónomos.

Alguns algoritmos populares de IA utilizados no planeamento de trajetórias e no movimento de robôs incluem:

- ***Graph Search Algorithms***: Estes incluem algoritmos como A*, pesquisa em profundidade, pesquisa em largura e pesquisa do melhor primeiro [4];
- ***Sampling-Based Algorithms***: Exemplos destes algoritmos incluem *rapidly-exploring random tree* (RRT) e outras técnicas semelhantes [5];
- ***ML Based Algorithms***: Algoritmos clássicos de Machine Learning (ML) tais como *Support Vector Machine* (SVM), *Long Short-Term Memory* (LSTM), *Convolutional Neural Networks* (CNN) e *Reinforcement Learning* [1];
- ***Artificial Potential Field Method***: Este é um algoritmo clássico que é utilizado no planeamento de percursos de robôs [6].

Estes algoritmos são utilizados para permitir que os robôs planeiem as suas trajetórias de forma eficiente, evitem obstáculos e naveguem por ambientes complexos, tendo em consideração várias restrições e otimizando o percurso com base em diferentes métricas de desempenho.

Cada algoritmo possui vantagens e limitações, e a escolha do algoritmo depende dos requisitos específicos da tarefa em questão. Por exemplo, algoritmos tradicionais como os algoritmos de pesquisa em grafos são eficientes para encontrar o caminho de menor custo, enquanto algoritmos

Sampling-Based como o RRT são úteis para o planeamento de trajetórias complexas de robôs. Algoritmos *ML Based*, como SVM, LSTM e CNN, conseguem aprender a partir de dados armazenados em histórico de forma a prever o caminho ótimo para um robô.

Apesar dos avanços significativos na área da robótica nos últimos anos, ainda persistem diversos desafios relacionados com o mapeamento robótico e a navegação autónoma, que requerem atenção e inovação. Alguns exemplos que ilustram esses desafios incluem [7]:

- **Localização e Mapeamento Simultâneos (SLAM)**: O problema da localização e mapeamento simultâneos continua a ser um desafio importante na robótica, especialmente em ambientes dinâmicos ou desconhecidos. A capacidade dos robôs construírem e atualizarem mapas do ambiente enquanto se localizam é essencial para a navegação autónoma;
- **Planeamento de Trajetória e Exploração**: O planeamento de trajetória e a exploração de ambientes desconhecidos são desafios críticos para a navegação autónoma. Os robôs precisam ser capazes de planear trajetórias eficientes, evitando obstáculos e otimizando a navegação em tempo real;
- **Mapeamento Autónomo em Ambientes Dinâmico**s: A capacidade dos robôs realizarem o mapeamento autónomo em ambientes dinâmicos, onde as condições e os obstáculos podem mudar, é um desafio contínuo que exige inovação em algoritmos e sensores.

4 Conclusão

Após o desenvolvimento das funcionalidades requisitadas no âmbito da Unidade Curricular de Algoritmia Avançada e da análise do Estado da Arte sobre a geração de trajetórias com robôs podemos concluir que esta é uma área com constantes desafios que requerem a adoção de métodos inovadores de forma a continuar a melhorar a eficiência e eficácia dos seus resultados.

O desenvolvimento e posterior estudo de uma solução recorrendo a uma abordagem *bruteforce*, ao gerar todas as combinações possíveis, permitiu-nos perceber que é vital a utilização de algoritmos mais eficientes, que recorrem, por exemplo, a heurísticas, de forma a encontrar soluções válidas em pouco tempo.

Foi através da implementação do Algoritmo Genético que conseguimos compreender melhor como podemos abordar problemas complexos usando técnicas de combinação e mutação, de forma a conseguir obter uma solução satisfatória em tempo útil.

Ao efetuar o estudo do Estado da Arte nesta área foi-nos também possível obter uma compreensão mais abrangente sobre o estado atual da área, da importância deste tipo de algoritmos para a resolução de problemas complexos, e da aplicação da inteligência artificial numa área bastante relevante.

5 Apêndice

Figura 1: *load_tasks*/2

```
load_tasks([H|T],N):-
	((H.type=="pick_delivery",
		asserta(
		t(
			H.id,
			cel(H.startFloorCode,H.startCoordinateX,H.startCoordinateY),
			cel(H.endFloorCode,H.endCoordinateX,H.endCoordinateY)
		)),
		planning:caminho_celulas_edificios(
			cel(H.device.initialCoordinates.floorCode, H.device.initialCoordinates.width, H.device.initialCoordinates.depth),
			cel(H.startFloorCode, H.startCoordinateX, H.startCoordinateY), _, C),

		% tarefa -> robot
		planning:caminho_celulas_edificios(
			cel(H.endFloorCode, H.endCoordinateX, H.endCoordinateY),
			cel(H.device.initialCoordinates.floorCode, H.device.initialCoordinates.width, H.device.initialCoordinates.depth), _, C1),

		asserta(distancias_robot_tarefa(H.id, C)),
		asserta(distancias_tarefa_robot(H.id, C1))
	);
		asserta(
		t(
			H.id,
			cel(H.floorId,H.startCoordinateX,H.startCoordinateY),
			cel(H.floorId,H.endCoordinateX,H.endCoordinateY)
		)),
		planning:caminho_celulas_edificios(
			cel(H.device.initialCoordinates.floorCode, H.device.initialCoordinates.width, H.device.initialCoordinates.depth),
			cel(H.floorId, H.startCoordinateX, H.startCoordinateY), _, C),

		% tarefa -> robot
		planning:caminho_celulas_edificios(
			cel(H.floorId, H.endCoordinateX, H.endCoordinateY),
			cel(H.device.initialCoordinates.floorCode, H.device.initialCoordinates.width, H.device.initialCoordinates.depth), _, C1),

		asserta(distancias_robot_tarefa(H.id, C)),
		asserta(distancias_tarefa_robot(H.id, C1))
	),
	N1 is N+1,
	load_tasks(T,N1).
```

Figura 2: *avalia*/2 *avalia_populacao*/2

```
avalia_populacao([],[]).
avalia_populacao([H|Resto],[H*V|Resto1]):-
        [Tarefa|_]=H,
        distancias_robot_tarefa(Tarefa, V1),
        avalia(H,V2),
        V is V1+V2,
        avalia_populacao(Resto,Resto1).

avalia([T1,T2|Resto],V):-
        tarefas(T1,T2,V1),
        avalia([T2|Resto],V2),
        V is V1 + V2.

avalia([H],Dist):-distancias_tarefa_robot(H, Dist).
avalia([],0).
```

Figura 4: *cruzar*/5

```
cruzar(Ind1,Ind2,P1,P2,NInd11):-
        sublista(Ind1,P1,P2,Sub1),
        n_tarefas(NumT),
        R is NumT-P2,
        rotate_right(Ind2,R,Ind21),
        elimina(Ind21,Sub1,Sub2),
        P3 is P2 + 1,
        insere(Sub2,Sub1,P3,NInd1),
        eliminah(NInd1,NInd11).
```

Figura 5: *gera_bruteforce*

```
gera_best_bruteforce:-
        debug_mode(D),
        get_time(Ti),
        findall(Tarefa,t(Tarefa,_,_),Tarefas),
        findall(P, permutation(Tarefas,P), Pop),
        avalia_populacao(Pop,PopAv),
        ordena_populacao(PopAv,[Ind|_]),
        get_time(Tf),
        ((D==1,write('Melhor individuo: '), write(Ind), nl);true),
        ((D==1,write('Tempo de execução: '), T is Tf - Ti, write(T), nl, nl);true).
```

Figura 3: *cruzamento*/2

```
cruzamento([],[]).
cruzamento([Ind*_],[Ind]).
cruzamento([Ind1*_,Ind2*_|Resto],[NInd1,NInd2|Resto1]):-
        gerar_pontos_cruzamento(P1,P2),
        prob_cruzamento(Pcruz),random(0.0,1.0,Pc),
        ((Pc =< Pcruz,!,
        cruzar(Ind1,Ind2,P1,P2,NInd1),
          cruzar(Ind2,Ind1,P1,P2,NInd2))
        ;
        (NInd1=Ind1,NInd2=Ind2)),
        cruzamento(Resto,Resto1).
```

Figura 6: *gera_geracao_time*/4

```
gera_geracao_time(T, G, Pop, Ind) :-
        debug_mode(D),
        lim_time(Lim),
        get_time(Ti),
        Tf is Ti - T,
        Tf > Lim,
        ((D==1,write('Geração '), write(G), write(':'), nl, write(Pop), nl);true),
          [M1, M2 | _] = Pop,
        cruzamento(Pop,NPop1),
        mutacao(NPop1,NPop),
        avalia_populacao(NPop,NPopAv),
        ordena_populacao(NPopAv,NPopOrd),
        avalia_populacao(NPop, NPopAv),
   ordena_populacao(NPopAv, NPopOrd),
        [Ind, M4 | _] = NPopOrd,
        ((D == 1, write('Melhor 1 individuo atual: '), write(Ind), nl, nl); true),!.
```

Figura 7: *gera_individuo*/3

```
gera_individuo([G],1,[G]):-!.

gera_individuo(ListaTarefas,NumT,[G|Resto]):-
        NumTemp is NumT+1, % To use with random
        random(1,NumTemp,N),
        retira(N,ListaTarefas,G,NovaLista),
        NumT1 is NumT-1,
        gera_individuo(NovaLista,NumT1,Resto).
```

Figura 8: *gera_lim_ger*/1

```
gera_lim_ger(Melhor):-
        debug_mode(D),
        gera_populacao(Pop),
        ((D==1,write('Pop='),write(Pop),nl);true),
        avalia_populacao(Pop,PopAv),
        ((D==1,write('PopAv='),write(PopAv),nl);true),
        ordena_populacao(PopAv,PopOrd),
        geracoes(NG),
        gera_geracao_ger(0,NG,PopOrd,Melhor),!.
```

Figura 9: *gera_lim_time*/1

```
gera_lim_time(Melhor):-
        debug_mode(D),
        gera_populacao(Pop),
        ((D==1,write('Pop='),write(Pop),nl);true),
        avalia_populacao(Pop,PopAv),
        ((D==1,write('PopAv='),write(PopAv),nl);true),
        ordena_populacao(PopAv,PopOrd),
        get_time(Ti),
        [M1, M2 | _] = PopOrd,
        ((D == 1, write('Melhor 1 individuo passado: '), write(M1), nl, nl); true),
        gera_geracao_time(Ti,0,PopOrd, Melhor), !.
```

Figura 10: *gera_populacao*/1

```
gera_populacao(Pop):-
        populacao(TamPop),
        n_tarefas(NumT),
        findall(Tarefa,t(Tarefa,_,_),ListaTarefas),
        gera_populacao(TamPop,ListaTarefas,NumT,Pop).

gera_populacao(0,_,_,[]):-!.

gera_populacao(TamPop,ListaTarefas,NumT,[Ind|Resto]):-
        TamPop1 is TamPop-1,
        gera_populacao(TamPop1,ListaTarefas,NumT,Resto),
        gera_individuo(ListaTarefas,NumT,Ind),
        \+member(Ind,Resto).

gera_populacao(TamPop,ListaTarefas,NumT,L):-
        gera_populacao(TamPop,ListaTarefas,NumT,L).
```

Figura 11: *gerar_pontos_cruzamento*/2

```
gerar_pontos_cruzamento(P1,P2):-
        gerar_pontos_cruzamento1(P1,P2).

gerar_pontos_cruzamento1(P1,P2):-
        n_tarefas(N),
        NTemp is N+1,
        random(1,NTemp,P11),
        random(1,NTemp,P21),
        P11\==P21,!,
        ((P11<P21,!,P1=P11,P2=P21);(P1=P21,P2=P11)).
gerar_pontos_cruzamento1(P1,P2):-
        gerar_pontos_cruzamento1(P1,P2).
```

Figura 12: *load_tarefa*/2

```
load_tarefas:-
        findall(T,t(T,_,_),L),
        load_tarefas(L).

load_tarefas([]).

load_tarefas([T|Resto]):-
        load_tarefa(T,Resto),
        load_tarefas(Resto).


load_tarefa(_,[]).
load_tarefa(Tarefa,[H|T]):-
        load_tarefa2(Tarefa,H),
        load_tarefa(Tarefa,T).


load_tarefa2(T1,T2):-
        debug_mode(D),
        t(T1,S1,F1), t(T2,S2,F2),
        ((D==1,nl,write('Tarefa '), write(T1), write(' e '), write(T2), write(': '), nl);true),
        ((D==1,write('F1: '), write(F1), write(' S2: '), write(S2));true),
        planning:caminho_celulas_edificios(F1,S2,_,W1),
        ((D==1,write(' W1: '), write(W1), nl);true),
        ((D==1,write('F2: '), write(F2), write(' S1: '), write(S1),nl);true),
        planning:caminho_celulas_edificios(F2,S1,_,W2),
        ((D==1,write(' W2: '), write(W2), nl);true),
        asserta(tarefas(T1,T2,W1)),
        asserta(tarefas(T2,T1,W2)).
```

Figura 13: *mutacao*/2

```
mutacao([],[]).
mutacao([Ind|Rest],[NInd|Rest1]):-
        prob_mutacao(Pmut),
        random(0.0,1.0,Pm),
        ((Pm < Pmut,!,mutacao1(Ind,NInd));NInd=Ind),
        mutacao(Rest,Rest1).

mutacao1(Ind,NInd):-
        gerar_pontos_cruzamento(P1,P2),
        mutacao22(Ind,P1,P2,NInd).

mutacao22([G1|Ind],1,P2,[G2|NInd]):-
        !, P21 is P2-1,
        mutacao23(G1,P21,Ind,G2,NInd).
mutacao22([G|Ind],P1,P2,[G|NInd]):-
        P11 is P1-1, P21 is P2-1,
        mutacao22(Ind,P11,P21,NInd).

mutacao23(G1,1,[G2|Ind],G2,[G1|Ind]):-!.
mutacao23(G1,P,[G|Ind],G2,[G|NInd]):-
        P1 is P-1,
        mutacao23(G1,P1,Ind,G2,NInd).
```

Referências

[1] C. Zhou, B. Huang e P. Fränti, «A review of motion planning algorithms for intelligent robots», *Journal of Intelligent Manufacturing*, vol. 33, n.º 2, pp. 387–424, 2022.

[2] A. K. R. Nadikattu, «Influence of Artificial Intelligence on Robotics Industry», *International Journal of Creative Research Thoughts (IJCRT), ISSN*, pp. 2320–2882, 2021.

[3] S. Yin, W. Ji e L. Wang, «A machine learning based energy efficient trajectory planning approach for industrial robots», *Procedia CIRP*, vol. 81, pp. 429–434, 2019.

[4] R. Robotin, G. Lazea e C. Marcu, «Graph search techniques for mobile robot path planning», em *Engineering the Future*, IntechOpen, 2010.

[5] M. Elbanhawi e M. Simic, «Sampling-based robot motion planning: A review», *Ieee access*, vol. 2, pp. 56–77, 2014.

[6] P. Vadakkepat, K. C. Tan e W. Ming-Liang, «Evolutionary artificial potential fields and their application in real time robot path planning», em *Proceedings of the 2000 congress on evolutionary computation. CEC00 (Cat. No. 00TH8512)*, IEEE, vol. 1, 2000, pp. 256–263.

[7] M. E. M. Mafalda et al., «Desenvolvimento de sistema de navegação autônomo com exploração de fronteira», 2021.