
# Relatório de ALGAV

**Sprint C**

**Turma 3DI - Grupo 50**
1191296 - Gabriel Almeida Gonçalves
1191369 - Tiago André Ferreira Leite
1200611 - Diogo Miguel Mota Carvalho
1201305 - Tiago Rafael Urze Afonso
1211304 - Francisco Lascasas Bogalho

**Data: 02/01/2023**

# Índice

# Índice de Figuras

# Introdução

Neste sprint, foi-nos proposto o desafio de gerar um plano de atendimento através de um Algoritmo Genético (*AG*). Estas tarefas, independentemente do seu tipo, são realizadas entre dois locais, e para gerar a ordem de realização de *N* tarefas, pretende-se minimizar o tempo gasto entre elas. Adicionalmente, realizou-se um pequeno estudo da eficácia e eficiência do algoritmo genético em comparação com o cálculo de todas as ordens de realização das tarefas, bem como um estudo sobre o estado da arte na geração de trajetórias de robots.

## Geração de todas as sequências de tarefas e escolha da melhor

Foi desenvolvida uma solução para que fosse possível gerar um plano de atendimento de tarefas de menor custo de tempo, em que este indicasse a sequência de execução dessas mesmas tarefas. Esta solução, passa por ser um *brute force*, pois é gerada todas as sequências de tarefas possíveis, e posteriormente analisada qual é que dura menos tempo.

Esta solução não é muito eficiente quando existe um grande número de tarefas que precisem de ser executadas. Obviamente que também esta não será uma solução viável nesse cenário, porque para além de demorar muito tempo, irá exceder o limite de utilização da *stack*. Isto será algo abordado mais à frente no relatório, onde é feita a análise deste algoritmo e de outro a implementar para a mesma finalidade. Lá também será decidido qual dos dois algoritmos deve ser aplicado em diferentes situações.

Para desenvolver então esta solução, foram criados predicados e utilizados outros já anteriormente implementados.

Primeiro, necessitávamos de criar uma lista com todas as tarefas a serem executadas com apenas a sua designação (por exemplo, t1, t2, …). Existe também a necessidade de gerar todas as combinações possíveis da ordem com que as tarefas são executadas. Tudo isto aqui indicado neste parágrafo, foi feito através do predicado criado com o nome "*generate_all_combinations_of_tasks/1*". Este predicado tinha essas duas responsabilidades, em que primeiro gera a lista das tarefas a realizar, e de seguida, faz todas as combinações possíveis. Para a parte de gerar todas as combinações possíveis, foi criado um predicado chamado "*my_permutation/2*", que é responsável por fazer a permutação dos valores de uma dada lista. Este predicado foi utilizado em conjunto com o predicado nativo do *Prolog* "*findall/3*", para que fosse gerada uma lista completa com as várias soluções de combinações diferentes da ordem de execução das tarefas.

O predicado principal responsável pela geração do plano completo de atendimento de tarefas é o "*get_best_solution_for_task_order_brute_force/2*". Neste predicado começamos por utilizar o predicado com nome "*generate_all_combinations_of_tasks/1*", de forma a obter uma lista com várias listas que representam as diferentes ordens de execução das tarefas. Posteriormente, esta lista gerada é passada para o predicado "*evaluate_population/2*", em que este é responsável por calcular o tempo de cada combinação de tarefas. Este predicado devolve uma lista com as várias listas que representam as diferentes combinações de execução de tarefas, mas agora com o tempo associado à execução dessas mesmas tarefas. O formato do resultado será algo como, por exemplo, R = [[t1, t2, t3, t4, t5]*87, [t1, t2, t3, t5, t4]*83, …].

*Figura 1 - Exemplo de resultado do cálculo do tempo para a execução de diferentes combinações de 5 tarefas*

Depois é utilizado o predicado "*order_population/2*", que é responsável por ordenar esta lista para que no primeiro membro fique aquela, ou uma daquelas, soluções de combinação de execuções de tarefas com o menor tempo associado. Após realizar esta ordenação, é retirado o primeiro membro da lista, que contém a ordem de execução de tarefas com o menor tempo associado. Com isto, separamos a lista com a ordem de tarefas a executar do seu custo temporal associado, e é então estes os valores que se obtém ao utilizar este predicado principal.

```
:-consult("genetic algorithm improved.pl").

get_best_solution_for_task_order_brute_force(Tasks, Cost):-
    generate_all_combinations_of_tasks(L),
    evaluate_population(L, L1),
    order_population(L1, R1),
    R1 = [P|_],
    P = Tasks*Cost.

generate_all_combinations_of_tasks(R):-
    findall(
        X,
        task(X, _),
        L1
    ),
    findall(
        L,
        my_permutation(L1, L),
        R
    ).

my_permutation([], []).
my_permutation(L, [X|L1]):-
    member(X, L),
    delete(L, X, LR),
    my_permutation(LR, L1).
```

*Figura 2 - Predicados criados para gerar um plano de atendimento de tarefas através da geração de todas as sequências possíveis e escolha da que dura menos tempo*

## Aleatoriedade no cruzamento entre indivíduos e seleção da nova geração da população do AG

### Método de seleção que não seja puramente elitista

Em vez de selecionarmos apenas os melhores indivíduos para recombinação, foi implementada uma seleção não elitista. Para tal, utilizamos o método de seleção *'fitness proportionate selection'*, também conhecido como *'roulette wheel selection'*. Este método seleciona os indivíduos entre o pai e seus descendentes, obtidos por cruzamento e mutação, com base no *fitness*, calculado no predicado "sum_inverse_cost", ou seja, quanto menor o custo, maior a probabilidade de serem escolhidos. Isso permite que alguns indivíduos com maior *fitness* (neste problema, um valor maior é uma solução pior) sejam selecionados para a geração seguinte. Este é feito no predicato "roulette_wheel_selection".

```
roulette_wheel_selection(N, Pop, SelectedPop):-
    roulette_wheel_selection(N, N, Pop, SelectedPop).

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

roulette_wheel_selection(N, Pop, [SelectedInd*Fitness|Rest]):-
    N > 0,
    roulette_wheel_select_individual(Pop, Pop, SelectedInd, Fitness, RemainingPop),
    N1 is N - 1,
    roulette_wheel_selection(N1, RemainingPop, Rest).

roulette_wheel_select_individual(Pop, NPopOrd, SelectedInd, Fitness, RemainingPop):-
    sum_inverse_cost(Pop, NPopOrd, TotalInverseCost),
    random(0.0, TotalInverseCost, RandomValue),
    roulette_wheel_find_individual(Pop, NPopOrd, RandomValue, SelectedInd, Fitness, RemainingPop).

roulette_wheel_find_individual([Ind*Fitness|Rest], NPopOrd, RandomValue, Ind, Fitness, Rest):-
    RandomValue =< Fitness, !.

roulette_wheel_find_individual([_|Rest], NPopOrd, RandomValue, SelectedInd, Fitness, RemainingPop):-
    roulette_wheel_find_individual(Rest, NPopOrd, RandomValue, SelectedInd, Fitness, RemainingPop).

sum_inverse_cost([], _, 0).
sum_inverse_cost([Ind*Cost|Rest], NPopOrd, TotalInverseCost):-
    sum_inverse_cost(Rest, NPopOrd, RestTotal),
    TotalInverseCost is RestTotal + (1 / Cost).
```

*Figura 3 - Exemplo do código roulette_wheel_selection/3*

Este é usado no predicado "generate_generation_time":

```
generate_generation_time(N, TI, Pop):-
    get_time(TA),
    exec_time(TE),
    T is TA - TI,
    TE > T,
    write('Generation '), write(N), write(':'), nl, write(Pop), nl,
    random_permutation(Pop, PopRandom),
    crossover(PopRandom, NPop1),
    mutation(NPop1, NPop),
    evaluate_population(NPop, NPopValue),
    append(NPopValue, Pop, AllPop),
    remove_duplicates(AllPop, UniqueAllPop),
    order_population(UniqueAllPop, UniqueAllPopOrd),
    assert_first_element(UniqueAllPopOrd),
    evaluate_stop(UniqueAllPopOrd),
    population(PopSize),
    roulette_wheel_selection(PopSize, UniqueAllPopOrd, SelectedPop),
    order_population(SelectedPop, SelectedPopOrd),
    N1 is N + 1,
    generate_generation_time(N1, TI, SelectedPopOrd).
generate_generation_time(_, _, _).
```

*Figura 4 - Exemplo do código generate_generation_time/3*

### Garantia que o melhor indivíduo passe para geração seguinte

Para garantir que o melhor indivíduo passe para a geração seguinte, a população anterior e os seus descendentes são juntos e ordenados, e no predicado "roulette_wheel_selection", o primeiro elemento, ou seja, o melhor indivíduo, é imediatamente colocado na lista de retorno. Isso foi feito ao chamar o predicado "roulette_wheel_select_individual" com Pop como ambos os primeiros argumentos, assegurando que o primeiro indivíduo selecionado faça parte da lista resultante. A recursão continua então com o restante da população.

```
roulette_wheel_selection(N, Pop, SelectedPop):-
    roulette_wheel_selection(N, N, Pop, SelectedPop).

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

roulette_wheel_selection(N, Pop, [SelectedInd*Fitness|Rest]):-
    N > 0,
    roulette_wheel_select_individual(Pop, Pop, SelectedInd, Fitness, RemainingPop),
    N1 is N - 1,
    roulette_wheel_selection(N1, RemainingPop, Rest).

roulette_wheel_select_individual(Pop, NPopOrd, SelectedInd, Fitness, RemainingPop):-
    sum_inverse_cost(Pop, NPopOrd, TotalInverseCost),
    random(0.0, TotalInverseCost, RandomValue),
    roulette_wheel_find_individual(Pop, NPopOrd, RandomValue, SelectedInd, Fitness, RemainingPop).

roulette_wheel_find_individual([Ind*Fitness|Rest], NPopOrd, RandomValue, Ind, Fitness, Rest):-
    RandomValue =< Fitness, !.

roulette_wheel_find_individual([_|Rest], NPopOrd, RandomValue, SelectedInd, Fitness, RemainingPop):-
    roulette_wheel_find_individual(Rest, NPopOrd, RandomValue, SelectedInd, Fitness, RemainingPop).

sum_inverse_cost([], _, 0).
sum_inverse_cost([Ind*Cost|Rest], NPopOrd, TotalInverseCost):-
    sum_inverse_cost(Rest, NPopOrd, RestTotal),
    TotalInverseCost is RestTotal + (1 / Cost).
```

*Figura 5 - Exemplo do código roulette_wheel_selection/3*

### Aleatoriedade no cruzamento entre indivíduos

Para evitar que a sequência de cruzamentos se dê sempre entre 1º e 2º elementos da população, depois entre 3º e 4º, etc., fazemos uma permutação aleatória á população antes do cruzamento e mutação.

```
generate_generation_time(N, TI, Pop):-
    get_time(TA),
    exec_time(TE),
    T is TA - TI,
    TE > T,
    write('Generation '), write(N), write(':'), nl, write(Pop), nl,
    random_permutation(Pop, PopRandom),
    crossover(PopRandom, NPop1),
    mutation(NPop1, NPop),
    evaluate_population(NPop, NPopValue),
    append(NPopValue, Pop, AllPop),
    remove_duplicates(AllPop, UniqueAllPop),
    order_population(UniqueAllPop, UniqueAllPopOrd),
    assert_first_element(UniqueAllPopOrd),
    evaluate_stop(UniqueAllPopOrd),
    population(PopSize),
    roulette_wheel_selection(PopSize, UniqueAllPopOrd, SelectedPop),
    order_population(SelectedPop, SelectedPopOrd),
    N1 is N + 1,
    generate_generation_time(N1, TI, SelectedPopOrd).
generate_generation_time(_, _, _).
```

*Figura 6 - Exemplo do código generate_generation_time/3*

## Parametrização da condição de término do AG

O Algoritmo Genético (AG) fornecido executa por um número N de gerações, definido pelo utilizador. Adicionalmente, foi desenvolvida o término por atingir uma solução com avaliação igual ou inferior a um dado valor, introduzido pelo utilizador. Caso uma solução não seja encontrada que satisfaça este critério, o término ocorre por tempo absoluto de execução.

Para a condição de termino quando é atingida uma solução considerada aceitável, este valor é definido pelo utilizador no facto "exec_limit/1". Em seguida, depois de cada mutação e cruzamento, é avaliado se o melhor elemento tem *fitness* (tempo de execução das tarefas) maior que o definido pelo utilizador. Quando falha, ou seja, quando tivermos um indivíduo com *fitness* igual ou menor que o definido, a geração de novas gerações para. Isto é feito no predicado "evaluate_stop/1".

Para a condição de término por tempo, o tempo definido pelo utilizador é armazenado no facto "exec_time/1". Em seguida, é passado como parâmetro para o predicado "generate_generation_time/3" o tempo inicial (TI), e a diferença entre este e o tempo atual (T) é calculada. Essa diferença (T) é então comparada com o valor do facto "exec_time/1" (TE). Quando TE > T, ou seja, quando ultrapassamos o tempo de execução definido pelo utilizador, o predicado falha, parando a geração de novas gerações.

O indivíduo com melhor fitness é guardado no facto *result*. Depois de cada cruzamento e mutação, é avaliado se existe algum indivíduo com melhor fitness, entre a população e os seus descendentes, e o melhor atual. Se tal for verdade, este é substituído. Isto é feito no predicado “assert_first_element/1”.

```
generate_generation_time(N, TI, Pop):-
    get_time(TA),
    exec_time(TE),
    T is TA - TI,
    TE > T,
    write('Generation '), write(N), write(':'), nl, write(Pop), nl,
    random_permutation(Pop, PopRandom),
    crossover(PopRandom, NPop1),
    mutation(NPop1, NPop),
    evaluate_population(NPop, NPopValue),
    append(NPopValue, Pop, AllPop),
    remove_duplicates(AllPop, UniqueAllPop),
    order_population(UniqueAllPop, UniqueAllPopOrd),
    assert_first_element(UniqueAllPopOrd),
    evaluate_stop(UniqueAllPopOrd),
    population(PopSize),
    roulette_wheel_selection(PopSize, UniqueAllPopOrd, SelectedPop),
    order_population(SelectedPop, SelectedPopOrd),
    N1 is N + 1,
    generate_generation_time(N1, TI, SelectedPopOrd).
generate_generation_time(_, _, _).

assert_first_element([FirstElement|_]) :-
    result(Best),
    extract_fitness(Best, BF),
    extract_fitness(FirstElement, CF),
    CF < BF,
    retract(result(Best)),
    asserta(result(FirstElement)),
    !.
assert_first_element([FirstElement|_]):-
    !.

evaluate_stop([FirstElement|_]) :-
    exec_limit(Limit),
    extract_fitness(FirstElement, FT),
    Limit < FT.
```

*Figura 7 - Exemplo do código generate_generation_time/3, assert_first_element/1, evaluate_stop/1*

Exemplo de execução de generate_time/1:

| | | |
|---|---|---|
| Tempo (s) | 1 | |
| Alvo | 20 | |
| População | 10 | |
| Cruzamento (%) | 0.7 | |
| Mutação (%) | 0.001 | |

```
Generation 260:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 261:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 262:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 263:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 264:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 265:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 266:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 267:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 268:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 269:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 270:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 271:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 274:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
Generation 275:
[[t4,t5,t2,t1,t3]*21,[t4,t2,t3,t1,t5]*23,[t5,t4,t1,t2,t3]*25,[t5,t2,t3,t1,t4]*26,[t5,t4,t2,t1,t3]*26,[t1,t2,t3,t5,t4]*28,[t5,t4,t1,t3,t2]*28,[t1,t2,t5,t4,t3]*29,[t4,t1,t5,t3,t2]*30,[t3,t4,t1,t5,t2]*31]
true.

2 ?- listing(result).
:- dynamic result/1.

result([t4, t5, t1, t2, t3]*17).

true.
```

*Figura 8 - Exemplo de execução de generate_time*

.

Como podemos observar, assim que um individuo com *fitness* menor ou igual que o alvo, a execução pára.

## Adaptação do algoritmo genético para o problema do planeamento da trajetória do robot dentro de edifícios conectados considerando várias tarefas

Como base para a nossa solução, foi utilizado o código fornecido em ALGAV. Este foi adaptado para responder ao nosso problema, e foram feitas melhorias ao AG de modo a potenciar melhores soluções.

No nosso problema, temos tarefas, o tempo de execução entre todas as tarefas, e o número de tarefas é representado pelos factos “task“, “time_between“, e “tasks“ respetivamente. O tempo de execução entre tarefas é calculado pelo facto “path_finder/7“, desenvolvido no sprint anterior.

```
% task(Id,ProcessTime).
task(t1).
task(t2).
task(t3).
task(t4).
task(t5).

% time_between(Task1, Task2, TimeBetween).
time_between(t1, t2, 6).
time_between(t1, t3, 9).
time_between(t1, t4, 10).
time_between(t1, t5, 2).

time_between(t2, t3, 5).
time_between(t2, t4, 7).
time_between(t2, t5, 2).

time_between(t3, t4, 17).
time_between(t3, t5, 13).

time_between(t4, t5, 4).

% tasks(NTasks).
tasks(5).
```

*Figura 9 - Exemplo dos factos task, tasks, time_between*

Para que o código fornecido funcione para o nosso problema, foi alterado o predicado "evaluate", que calcula a *fitness* de cada indivíduo. Aqui, apenas é tomado em conta o tempo que demora a ir de uma tarefa para outra, visto que os tempos dentro da própria tarefa irá ser sempre igual para todas as combinações de tarefas.

```
evaluate_population([],[]).
evaluate_population([Ind|Rest],[Ind*V|Rest1]):-
    evaluate(Ind,V),
    evaluate_population(Rest,Rest1).

evaluate([_], 0).

evaluate([T1, T2|Rest], TotalCost) :-
    (time_between(T1, T2, Cost);time_between(T2, T1, Cost)),
    evaluate([T2|Rest], RestCost),
    TotalCost is Cost + RestCost.
```

*Figura 10 - Exemplo do código evaluate_population/2*

## Estudo do estado da arte de métodos de planeamento de trajetória e movimento de robots

A linguagem Prolog é uma linguagem de programação lógica com amplo uso em várias áreas da robótica, nomeadamente – mas não exclusivamente – na inteligência artificial. A sua criação data de 1972, onde terá sido desenvolvida na Universidade de Marselha, em França. Desde então, sabemos que a Prolog tem sido utilizada para os mais diversos fins, tais como a compreensão de linguagens naturais, a automação de projetos, a análise de estruturas bioquímicas, entre muitos outros.

Ao contrário de uma vasta parte das linguagens de programação, Prolog é uma linguagem criada com um intuito maioritariamente declarativo, o que significa que a lógica é expressa por meio de factos e regras, que servem como veículo para a descrição do problema que se pretende computar (Dantas, s. d.). Ademais, a Prolog não possui as habituais estruturas de controlo presentes na larga maioria das linguagens de programação, o que faz com que seja necessário recorrer a métodos lógicos para fazer com que o programa cumpra aquilo que lhe é requerido (Dantas, s. d.). Tal como referimos anteriormente, este programa tem sido amplamente reconhecido pela fascinante área da robótica como um recurso de enorme utilidade, com destaque para o ramo de geração de trajetórias de robôs, que é aquele ao qual este relatório se irá referir, visto que é nele que se foca a presente investigação.

O planeamento de trajetórias de robôs é uma das mais fundamentais áreas de pesquisa científica no campo da robótica, dado que é algo tem potencial para se tornar imensamente útil ao ser humano, seja sob a forma de carros automatizados, de ferramentas de limpeza autónomas ou até mesmo de robôs de salvamento, por exemplo. Esta tecnologia é definida como o método ao qual se recorre para encontrar o melhor caminho de um ponto para o outro no espaço, otimizando o trajeto entre origem e destino ao determinar o menor percurso entre ambos (Agarwal, Guruji & Parsedyia, 2016). Não raramente, esta tecnologia é apelidada de *"motion planning"*, uma vez que se refere à definição dos movimentos de um dado objeto no seio do ambiente de estudo (Agarwal, Guruji & Parsedyia, 2016). Um objeto pode ser um robô que é autónomo por natureza, pois utiliza o algoritmo de determinação do caminho para determinar os pontos de passagem no espaço. Este tipo de robô é designado por robô móvel, que é exatamente o tipo de instrumento com o qual estamos a trabalhar neste projeto.

Nos dias de hoje, a aplicabilidade prática do tipo de tecnologia que temos vindo a mencionar tem já vindo a dar os seus frutos, mas as perspetivas para o futuro permanecem em aberto, principalmente devido ao facto de ser possível conjugar diversas ferramentas de programação de modo a aproveitar os

pontos fortes de cada um delas, fazendo assim com que se obtenham os melhores e mais eficazes resultados possíveis (Elbanhawi & Simic, 2014). No nosso caso, combinamos a linguagem de programação Prolog com o algoritmo A* – leia-se, A-Star – para tentar definir o trajeto mais eficiente e rápido de um robô que parte de um determinado ponto A para um determinado ponto B. A* é um algoritmo de busca que pode ser usado para encontrar caminhos no que toca à navegação robótica. Baseado numa função heurística, o propósito desta ferramenta é calcular o valor da função heurística em cada nó da área de trabalho e verificar os nós adjacentes para encontrar a solução ótima com probabilidade zero de colisão (Agarwal, Guruji & Parsedyia, 2016).

A demanda por robôs móveis em vários aspetos das nossas vidas quotidianas levou a um rápido desenvolvimento da tecnologia de computadores. Contudo, a literatura contemporânea tem vindo a demonstrar que estes apresentam frequentemente problemas na rapidez com que procuram o caminho ótimo, bem como na sua eficácia de trabalho. Apesar de A* ser consensualmente considerado como o algoritmo mais eficaz no que diz respeito ao aprimoramento das capacidades de planeamento motor da inteligência artificial (Gao, Yan, Shen & Ma, 2022) (Jiang, Wang & Wang, 2023) (Bao, Li, Pan & Yu, 2022) (Wang, 2021) (Lan, Lv, Liu, He & Zhang, 2021), a verdade é que apresenta alguns constrangimentos, designadamente a redundância dos pontos de inflexão nos trajetos planeados pelo algoritmo tradicional e o facto de que o comprimento dos caminhos tende a ser relativamente longo. Como tal, os académicos têm-se debruçado sobre as implicações destas lacunas e, simultaneamente, tentado arranjar soluções para estes problemas. Assim, inúmeros estudos têm sido publicados recentemente onde são testadas e estabelecidas novas alternativas de algoritmos baseados no A*, mas com formulações mais inovadoras e adaptadas às necessidades da realidade da robótica atual.

# Conclusão

O problema da geração de planos de atendimento de tarefas pode ser resolvido de forma simples, mas ineficiente, pela abordagem de força bruta, que tem uma complexidade computacional elevada para um grande número de tarefas (≤7).

Uma abordagem mais sofisticada é o Algoritmo Genético (AG), que foi adaptado e melhorado da solução proposta em ALGAV. As melhorias incluíram a alteração do método de avaliação, a utilização de uma seleção não elitista e a introdução de aleatoriedade na recombinação dos indivíduos, melhorando a qualidade dos resultados.

A comparação entre a força bruta e o AG mostrou que o AG melhorado tem custos mais baixos em várias instâncias, demonstrando a sua eficácia na procura da melhor solução. No entanto, a eficiência do AG em relação à força bruta varia com o número de tarefas envolvidas. A tabela comparativa indica que a força bruta pode ser mais vantajosa para até 7 tarefas, enquanto o AG se sobressai para um número maior de tarefas.

## Referências Bibliográficas


Agarwal, H., Guruji, A. K., & Parsedyia, D. K. (2016). Time-Efficient A* Algorithm for Robot Path Planning. *Procedia Technology, 23,* 144-149.

https://doi.org/10.1016/j.protcy.2016.03.010

Bao, W., Li, J., Pan, Z. & Yu, R. (2022, novembro, 25-27). *"Improved A-star Algorithm for Mobile Robot Path Planning Based on Sixteen-direction Search". 2022 China Automation Congress (CAC)*, Xiamen, China.

Dantas, L. A. (s. d.). Descobrindo o Prolog. *Linha de Código.* http://www.linhadecodigo.com.br/artigo/1697/descobrindo-o-prolog.aspx

Elbanhawi, M. & Simic, M. (2014). Sampling-Based Robot Motion Planning: A Review. *IEEE Access, 2(1),* 56-77. http://dx.doi.org/10.1109/ACCESS.2014.2302442

Gao, H., Yan, L., Shen, R., & Ma, S. (2022, janeiro, 15-16). *Design and Implementation of Mobile Robot Path Planning Based on A-Star Algorithm. 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA),* Changsha, China.

Jiang, C., Wang, C., & Wang, M. (2023, setembro, 15-17). *Research on Path Planning for Mobile Robots Based on Improved A-star Algorithm. 2023 IEEE 7th Information Technology and Mechatronics Engineering Conference (ITOEC)*, Chongqing, China.

Lan, X., Lv, X., Liu, W., He, Y., & Zhang, X. (2021, março, 12-14*). Research on Robot Global Path Planning Based on Improved A-star Ant Colony Algorithm*. *2021 IEEE 5th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC),* Chongqing, China.

Wang, B. (2021, agosto, 29-29). *Path Planning of Mobile Robot Based on A* Algorithm. 2021 IEEE International Conference on Electronic Technology, Communication and Information (ICETCI),* Changchun, China

# Anexos

## Estudo da complexidade da geração de todas as sequências de tarefas e escolha da melhor

Para avaliar a eficácia e eficiência do algoritmo genético, elaboramos uma tabela comparativa destacando os custos obtidos em três instâncias distintas: a versão inicial, a versão aprimorada e por fim a melhor solução. Esta abordagem permitirá uma compreensão clara da evolução do desempenho do algoritmo ao longo das iterações, oferecendo *insights* valiosos sobre as melhorias implementadas e a eficácia global do processo de otimização.

*Tabela 1 - Análise da eficácia e eficiência do algoritmo genético*

| Nº Tarefas | AG inicial | AG melhorado | Melhor solução | Tempo – Melhor solução | Ordem de Execução das Tarefas |
|---|---|---|---|---|---|
| **5** | 78 | 75 | 72 | 0.0135 segundos | [t3,t2,t1,t5,t4] |
| **6** | 97 | 95 | 93 | 0.0899 segundos | [t1,t5,t6,t3,t2,t4] |
| **7** | 105 | 104 | 100 | 3.6072 segundos | [t3,t2,t6,t5,t1,t7,t4] |
| **8** | 128 | 118 | 115 | 232.8564 segundos | [t3,t2,t8,t4,t7,t1,t5,t6] |
| **9** | 138 | 132 | N/A | N/A | [t8,t5,t4,t9,t7,t1,t2,t3,t6] |
| **10** | 150 | 147 | N/A | N/A | [t1,t6,t4,t5,t3,t7,t10,t8,t9,t2] |

Ao analisarmos os dados gerados, observamos que, de modo geral, o algoritmo genético aprimorado demonstra custos mais baixos em comparação com a versão inicial. Essa redução significativa nos custos destaca a eficácia notável do algoritmo genético melhorado na procura pela melhor solução, contribuindo para uma otimização mais eficiente dos recursos.

No que diz respeito à análise da complexidade do problema, é crucial considerar os limites de utilização de cada algoritmo. Neste contexto, é relevante ressaltar que o algoritmo genético revela vantagens em cenários com um número mais elevado de tarefas. Contudo, é importante destacar que, como indicado, a abordagem de força bruta é a melhor solução, sendo mais vantajosa até que o número de tarefas seja menor ou igual a 7.

Esta análise enfatiza a importância de selecionar o algoritmo mais apropriado com base no contexto específico, considerando a relação entre o número de tarefas e a eficiência computacional. Em situações em que o número de tarefas é limitado a 7 ou menos, a melhor solução destaca-se como a opção mais vantajosa em termos de custos, indicando um ponto crítico em que a transição para o algoritmo genético se torna mais benéfica. A melhoria na eficiência e na qualidade das soluções oferece uma clara vantagem ao algoritmo genético aprimorado, destacando a sua capacidade de encontrar soluções mais precisas e rápidas, tornando-o uma escolha mais vantajosa em termos de desempenho e utilização de recursos.