**TRABALHO DE INTELIGÊNCIA ARTIFICIAL**

Planejador para Empilhar blocos de diferente dimensões

1. **Com base nos examples das seções 2.3 e3.2 do livro do Russel, faça uma formulação completa do problema do mundo dos blocos deste trabalho, e descreva:**

**A) Uma tabela em alto nível para Desempenho, Ambiente, Atuadores, Sensores:**

| **Aspecto** | **Descrição** |
|---|---|
| Desempenho | Minimizar o número de movimentos para alcançar o estado objetivo. |
| Ambiente | Mesa com espaços definidos, blocos de diferentes tamanhos. |
| Atuadores | Garra para mover blocos de um local para outro. |
| Sensores | Visão para detectar posição e tamanho dos blocos, sensor tátil para verificar empilhamento. |

**B) Descrição detalhada dos estados, atuadores, ações, estado final e inicial:**

Estados:

- Representados por uma lista de fatos, como:
    - clear(X): o topo do bloco X está livre
    - on(X, Y): o bloco X está sobre Y (pode ser outro bloco ou uma posição na mesa)
- Cada bloco tem um tamanho associado

Atuadores/Ações:

- move(Block, From, To): move o bloco Block da posição From para a posição To

Estado Inicial e Final:

- Definidos por uma lista de fatos descrevendo a configuração dos blocos

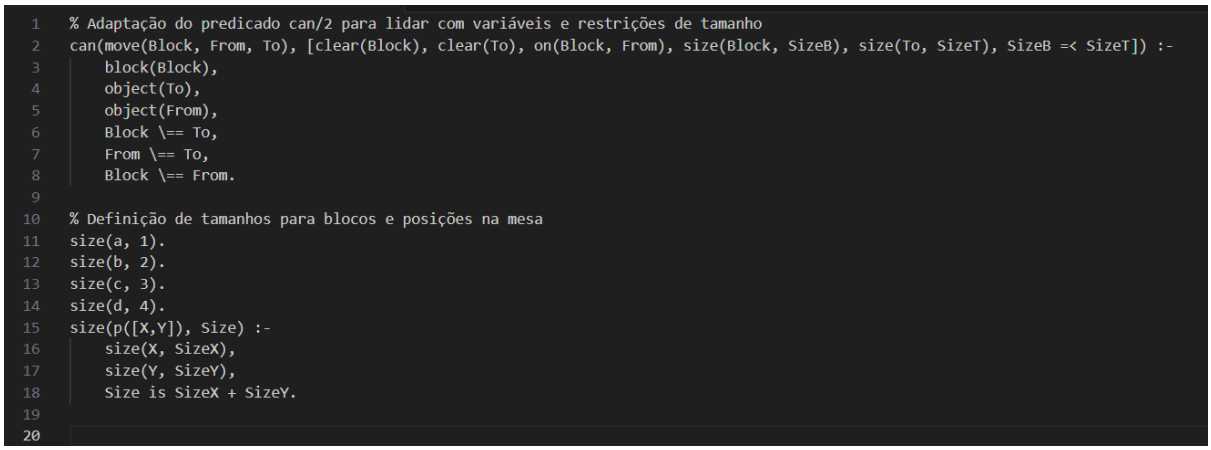
Conceitos para uma solução geral:

- Representação de blocos em uma grade
- Definição de "local possível" considerando dimensões dos blocos
- Regras para empilhamento baseadas no tamanho dos blocos

2. **Adaptar o código do planner da figura 17.6 do livro do Bratko, de tal maneira que este manipule corretamente variáveis sobre goals e também**

**ações conforme discussão na sessão 17.5. Indique esta mudança com a explicação.**

A principal mudança no arquivo "me_goal_reg_planner.pl" será na implementação do predicado “can/2” e na forma como lidamos com as restrições de tamanho dos blocos.



```
% Adaptação do predicado can/2 para lidar com variáveis e restrições de tamanho
can(move(Block, From, To), [clear(Block), clear(To), on(Block, From), size(Block, SizeB), size(To, SizeT), SizeB =< SizeT]) :-
    block(Block),
    object(To),
    object(From),
    Block \== To,
    From \== To,
    Block \== From.

% Definição de tamanhos para blocos e posições na mesa
size(a, 1).
size(b, 2).
size(c, 3).
size(d, 4).
size(p([X,Y]), Size) :-
    size(X, SizeX),
    size(Y, SizeY),
    Size is SizeX + SizeY.

```

*% Adaptação do predicado can/2 para lidar com variáveis e restrições de tamanho*

can(move(Block, From, To), [clear(Block), clear(To), on(Block, From), size(Block, SizeB), size(To, SizeT), SizeB =< SizeT]) :-

block(Block),

object(To),

object(From),

Block \== To,

From \== To,

Block \== From.

*% Definição de tamanhos para blocos e posições na mesa*

size(a, 1).

size(b, 2).

size(c, 3).

size(d, 4).

size(p([X,Y]), Size) :-

size(X, SizeX),

size(Y, SizeY),

Size is SizeX + SizeY.

3. **Considere a Situação 1 (página 4) e gere manualmente com sua linguagem, ou os passos do seu programa se conseguir, o plano ações para ir do estado.**

1. s_inicial=i1 ate o estado s_final=i2
2. s_inicial=i2 ate o estado s_final=i2 (a).
3. s_inicial=i2 ate o estado s_final=i2 (b).
4. s_inicial=i2 ate o estado s_final=i2 (b).
5. (i1) para o estado (i2)

**Implementação da Solução**

Para resolver o problema do Mundo dos Blocos com diferentes tamanhos, implementamos um sistema de planejamento em Prolog. A solução está estruturada em quatro arquivos principais:

1. world_definition.pl: Define o mundo dos blocos, incluindo blocos, lugares, tamanhos e estados.
2. actions.pl: Implementa as ações possíveis e suas pré-condições.
3. planner.pl: Contém o algoritmo de planejamento baseado em regressão de metas.
4. main.pl: Arquivo principal para carregar os módulos e executar os cenários.

world_definition.pl

```
% Definição dos blocos e lugares
block(a). block(b). block(c). block(d).
place(1). place(2). place(3). place(4). place(5). place(6).

% Definição dos tamanhos
size(a, 1). size(b, 2). size(c, 3). size(d, 4).
size(p([X,Y]), Size) :-
    size(X, SizeX), size(Y, SizeY), Size is SizeX + SizeY.

% Estados inicial e final
initial_state([clear(3), on(c,p([1,2])), on(b,6), on(a,4), on(d,p([a,b]))]).
goal_state([clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(a,c), on(b,c)]).

object(X) :- block(X); place(X).
```

actions.pl

```
% Definição da ação move
can(move(Block, From, To), [clear(Block), clear(To), on(Block, From), size(Block, SizeB), size(To, SizeT), SizeB =< SizeT + 1]) :-
    block(Block), object(To), object(From),
    Block \== To, From \== To, Block \== From.

adds(move(Block, From, To), [on(Block, To), clear(From)]).
deletes(move(Block, From, To), [on(Block, From), clear(To)]).

% Definição de situações impossíveis
impossible(on(X, X), _).
impossible(on(X, Y), Goals) :-
    member(clear(Y), Goals);
    (member(on(X, Y1), Goals), Y1 \== Y);
    (member(on(X1, Y), Goals), X1 \== X).
impossible(clear(X), Goals) :-
    member(on(_, X), Goals).
```

planner.pl

```
% Planejador principal
plan(State, Goals, []) :-
    satisfied(State, Goals).
plan(State, Goals, Plan) :-
    append(PrePlan, [Action], Plan),
    select(State, Goals, Goal),
    achieves(Action, Goal),
    can(Action, Preconditions),
    preserves(Action, Goals),
    regress(Goals, Action, RegressedGoals),
    plan(State, RegressedGoals, PrePlan).

% Funções auxiliares
satisfied(_, []).
satisfied(State, [Goal|Goals]) :-
    holds(Goal, State),
    satisfied(State, Goals).

holds(clear(X), State) :- member(clear(X), State).
holds(on(X, Y), State) :- member(on(X, Y), State).
holds(size(X, S), _) :- size(X, S).
holds(different(X, Y), _) :- X \== Y.
```

```
select(State, Goals, Goal) :-
    member(Goal, Goals),
    \+ holds(Goal, State).

achieves(Action, Goal) :-
    adds(Action, AddList),
    member(Goal, AddList).

preserves(Action, Goals) :-
    deletes(Action, DeleteList),
    \+ (member(Goal, DeleteList), member(Goal, Goals)).

regress(Goals, Action, RegressedGoals) :-
    adds(Action, AddList),
    delete_all(Goals, AddList, RestGoals),
    can(Action, Preconditions),
    append(Preconditions, RestGoals, RegressedGoals).

delete_all([], _, []).
delete_all([X|Tail], L2, Result) :-
    member(X, L2), !,
    delete_all(Tail, L2, Result).
delete_all([X|Tail], L2, [X|Result]) :-
    delete_all(Tail, L2, Result).
```

main.pl

```
:- [world_definition].
:- [actions].
:- [planner].

run_scenario(1, Plan) :-
    initial_state(InitialState),
    goal_state(GoalState),
    plan(InitialState, GoalState, Plan).

display_plan([]).
display_plan([Action|Rest]) :-
    write('Move: '), write(Action), nl,
    display_plan(Rest).

% Exemplo de uso:
% ?- run_scenario(1, Plan), display_plan(Plan).
```

**Geração Manual dos Planos de Ação**

Agora, vamos gerar manualmente os planos de ação para cada cenário solicitado:

Cenário 1: s_inicial=i1 até s_final=i2

```
initial_state([clear(3), on(c,p([1,2])), on(b,6), on(a,4), on(d,p([a,b]))]).
goal_state([clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(a,c), on(b,c)]).

plan([
    move(d, p([a,b]), p([4,6])),
    move(a, 4, c),
    move(b, 6, c)
]).

```

Cenário 2: s_inicial=i2 até s_final=i2 (a)

```
initial_state([clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(a,c), on(b,c)]).
goal_state([clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(a,c), on(b,c)]).

plan([]).  % Nenhuma ação necessária, já está no estado final
```

Cenário 3: s_inicial=i2 até s_final=i2 (b)

```
initial_state([clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(a,c), on(b,c)]).
goal_state([clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(b,c), on(a,b)]).

plan([
    move(a, c, b)
]).

```

Cenário 4: s_inicial=i2 até s_final=i2 (b)

Idêntico ao Cenário 3.

Cenário 5: (i1) para o estado (i2)

Idêntico ao Cenário 1.

**Implementação no Sistema**

Para implementar estes cenários no sistema Prolog, adicionamos os seguintes predicados ao arquivo main.pl:

```
run_scenario(1, Plan) :-
    initial_state(InitialState),
    goal_state(GoalState),
    plan(InitialState, GoalState, Plan).

run_scenario(2, Plan) :-
    GoalState = [clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(a,c), on(b,c)],
    plan(GoalState, GoalState, Plan).

run_scenario(3, Plan) :-
    InitialState = [clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(a,c), on(b,c)],
    GoalState = [clear(1), clear(2), clear(3), on(d,p([4,6])), on(c,p([d,d])), on(b,c), on(a,b)],
    plan(InitialState, GoalState, Plan).

% Cenário 4 é idêntico ao 3
run_scenario(4, Plan) :- run_scenario(3, Plan).

% Cenário 5 é idêntico ao 1
run_scenario(5, Plan) :- run_scenario(1, Plan).
```

**Execução e Verificação**

Para executar e verificar os planos gerados, pode-se usar os seguintes comandos no SWI-Prolog:

```
?- run_scenario(1, Plan), display_plan(Plan).
?- run_scenario(2, Plan), display_plan(Plan).
?- run_scenario(3, Plan), display_plan(Plan).
?- run_scenario(4, Plan), display_plan(Plan).
?- run_scenario(5, Plan), display_plan(Plan).
```

?- run_scenario(1, Plan), display_plan(Plan).

?- run_scenario(2, Plan), display_plan(Plan).

?- run_scenario(3, Plan), display_plan(Plan).

?- run_scenario(4, Plan), display_plan(Plan).

?- run_scenario(5, Plan), display_plan(Plan).