*Program przeszukiwania przestrzeni stanów według algorytmu A**

WPROWADZENIE DO BUDOWY PROGRAMU

STRUKTURY DANYCH

Struktury danych przetwarzane przez program to
--kolejka węzłów drzewa poszukiwań
--zbiór zamknięty węzłów drzewa poszukiwań
implementowane jako listy.

Kolejka zawiera węzły osiągnięte w trakcie dotychczasowego przebiegu algorytmu, a jeszcze nie rozpatrzone. Zbiór zamknięty zawiera węzły już przetworzone - tzn. węzły, dla których wygenerowano już węzły potomne.

Elementami kolejki i zbioru zamkniętego są egzemplarze struktury opisującej węzeł:

node(***<struktura reprezent. stan >*** **,** ***<struktura reprezent. akcję >*** **,**
< struktura reprezent. stan poprzedzający > **,**
<dotychczasowy koszt> **,** ***<ocena wg funkcji f>*** **)**

GŁÓWNA PROCEDURA PROGRAMU

Główną procedurą programu jest rekurencyjna procedura **search_A_star,** ***która implementuje jeden krok algorytmu***

```
PROCEDURA search_A_star (Queue, ClosedSet, PathCost)
{
        < pobierz węzeł z kolejki Queue>

        <zbadaj stan przypisany do pobranego węzła>

          JEŚLI    < stan spełnia warunek stanu docelowego >

                < zbuduj ścieżkę—do wyniku PathCost wpisz
                 otrzymaną ścieżkę oraz koszt przypisany do węzła>

           W PRZECIWNYM PRZYPADKU
           {
                < konstruuj listę węzłów potomnych dla pobranego węzła >

                <węzły potomne wstaw do kolejki Queue,
                 przetworzony wezeł włącz do zbioru zamkniętego ClosedSet>

                < wywołaj rekurencyjnie search_A_star dla nowej kolejki węzłów
                  i nowego zbioru zamkniętego>

                 < jako wynik PathCost zwróć wynik
                   zwrócony przez wywołanie rekurencyjne>

           }
   }
```

ZAŁOŻENIA DOTYCZĄCE DEFINICJI PRZESTRZENI STANÓW DLA DZIEDZINY PROBLEMU

Niniejsza implementacja algorytmu zakłada, że definicja przestrzeni stanów dla dziedziny problemu, do której ma być zastosowany algorytm, jest dołączona do programu w postaci następującego zestawu procedur:

--procedura **succ** **–** ***procedura niedeterministyczna, która implementuje funkcję*** **następnik:**
dla podanego stanu procedura zwraca **(***niedeterministycznie***)** ***atrybuty JEDNEGO z możliwych stanów potomnych. W wyniku wymuszania nawrotów do tej procedury program uzyskuje wszystkie stany potomne dla stanu podanego w wywołaniu procedury*** **succ.** ***Atrybuty stanu potomnego zwracane przez*** **succ,** ***to:***

- ***--opis stanu***
- ***--opis akcji powodującej przejście od stanu zadanego do stanu potomnego***
- ***--koszt kroku***

--procedura **goal** **–** ***procedura implementująca sprawdzenie, czy podany stan spełnia kryterium/kryteria celu***

--procedura **hscore** **–** ***procedura implementująca składową*** **h** ***oceny heurystycznej***

****
KOD PROGRAMU Z OPISEM
****

Procedura start_A_star

***Procedura inicjalizująca: wywołuje główną procedurę programu,* search_A_star,**
z zainicjalizowaną kolejką węzłów i pustym zbiorem zamkniętym

Postać wywołania :

start_A_star(< *opis stanu poczatkowego*>, PathCost)

start_A_star (InitState, PathCost) :-

score(InitState, 0, 0, InitCost, InitScore) ,
< określ koszt początkowy **InitCost** *oraz ocenę stanu początkowego*
wg przyjętej heurystyki **InitScore** *>*

search_A_star([node(InitState, nil, nil, InitCost , InitScore)], [ ], PathCost) .
<wywołaj główną procedure programu z kjolejką zawierającą opis węzła
początkowego oraz pustym zbiorem zamkniętym.
Dla stanu początkowego jako stan poprzedzający przyjmuje się symbol **nil,**
podobnie jako akcję tworzącą sytan początkowy przyjmuje się symbol **nil** *>*

Wynik **PathCost** *zwrócony przez procedurę* **search_A_star** *zostanie zwrócony*
jako wynik procedury **start_A_star**

Procedura search_A_star

Główna pętla programu: procedura rekurencyjna, która na każdym poziomie rekurencji wykonuje jeden krok algorytmu

Postać wywołania :

search_A_star(< *kolejka węzłów* >, < *zbiór zamknięty*>, PathCost)

Wynik—ścieżka z kosztem--zostanie związany ze zmienną **PathCost**

Na każdym poziomie rekurencji procedura
---pobiera pierwszy węzeł z kolejki-w tym celu wywołuje procedurę **fetch**
---sprawdza, czy stan związany z pobranym węzłem spełnia warunki celu
---jeśli tak, to wywołuje procedurę **build_path** *w celu skonstruowania ścieżki od stanu początkowego do znalezionego stanu docelowego i kończy wykonanie*
---w przeciwnym przypadku wywołuje procedure **continue,** która
---wywołuje procedurę **expand** *w celu wygenerowania węzłów potomnych węzła pobranego z kolejki*
---wstawia otrzymane węzły potomne na właściwe pozycje do kolejki - w tym celu wywołuje procedurę **insert_new_nodes**
---wywołuje siebie rekurencyjnie podając w wywołaniu nową kolejkę i zaktualizowany zbiór zamknięty.

search_A_star(Queue, ClosedSet, PathCost) :-

fetch(Node, Queue, ClosedSet , RestQueue),
< z kolejki pobierz pierwszy węzeł, dla którego nie występuje w zbiorze zamkniętym węzeł związany z tym samym stanem
wynik: **Node** - *pobrany węzeł,* **RestQueue** - *reszta kolejki >*

continue(Node, RestQueue, ClosedSet, PathCost).
<*zbadaj stan przypisany do pobranego węzła: zakończ lub kontynuuj zależnie od tego, czy stan spełnia warunek stanu docelowego>*

Procedura continue

Postać wywołania :

continue(< *opis węzła*>, < *kolejka węzłów* >, < *zbiór zamknięty*>, PathCost)

Wynik zostanie związany ze zmienną **PathCost**

continue(node(State, Action, Parent, Cost, _) , _ , ClosedSet,
path_cost(Path, Cost)) :-

goal(State), ! ,

build_path(node(Parent, _ ,_ , _ , _) , ClosedSet, [Action/State], Path) .

<jeśli stan przypisany do pobranego węzła spełnia warunek stanu docelowego to zbuduj ścieżkę—do wyniku wpisz, jako elementy struktury **path_cost**, *otrzymaną ścieżkę oraz koszt przypisany do węzła>*

<Procedura **goal** *, sprawdzająca czy stan spełnia kryterium stanu celu, jest składnikiem dołączanej do programu definicji przestrzeni stanów –patrz założenia do program podane wyżej>*

<w przeciwnym przypadku>

continue(Node, RestQueue, ClosedSet, Path) :-

expand(Node, NewNodes),
< konstruuj listę węzłów potomnych dla węzła **Node***>*

insert_new_nodes(NewNodes, RestQueue, NewQueue),
<węzły z listy wygenerowanej przez procedurę **expand** *wstaw do kolejki >*

search_A_star (NewQueue, [Node | ClosedSet], Path).
< rekurencyjne wywołanie **search_A_star***: kontynuuj przeszukiwanie dla nowej kolejki węzłów. Opis rozpatrzonego węzła zostaje dołączony do zbioru zamkniętego >*

Procedura fetch

Procedura pobiera z kolejki pierwszy węzeł, który nie jest węzłem ze stanem powtórzonym

Postać wywołania :

fetch(Node, < *kolejka węzłów* >, < *zbiór zamknięty*>, RestQueue)

Wynik—pobrany węzeł i reszta kolejki-- --zostanie związany odpowiednio ze zmiennymi **Node, RestQueue**

Procedura pobiera pierwszy element z kolejki węzłów i sprawdza, czy w zbiorze zamkniętym nie występuje węzeł , do którego jest przypisany ten sam stan.
--jeśli nie, to procedura wiąże zmienną **Node** *z opisem tego węzła, a zmienną* **RestQueue** *z resztą kolejki, pozostałą po pobraniu węzła*
--w przeciwnym razie następuje rekurencyjne wywołanie procedury dla reszty kolejki, pozostałej po usunięciu zbadanego węzła.

```
fetch(node(State, Action,Parent, Cost, Score),
                    [node(State, Action,Parent, Cost, Score) |RestQueue],
                                                    ClosedSet,  RestQueue) :-

        \+member(node(State, _ ,_  , _ , _ ) , ClosedSet),    ! .


fetch(Node, [ _ |RestQueue], ClosedSet, NewRest) :-

        fetch(Node, RestQueue, ClosedSet , NewRest).
```

Procedura expand

Procedura konstruuje listę węzłów potomnych dla węzła podanego w pierwszym argumencie wywołania

Postać wywołania:

expand(<*opis węzła*>, NewNodes)

Wynik zostanie związany ze zmienną **NewNodes**

```
expand(node(State, _ ,_ , Cost, _ ), NewNodes), :-

        findall(node(ChildState, Action, State, NewCost, ChildScore) ,
                    (succ (State, Action, StepCost, ChildState),
                       score(ChildState, Cost, StepCost, NewCost, ChildScore) ) ,

                                                                    NewNodes),

                                                                    ! .
```

Węzły potomne są uzyskiwane przez kolejne nawroty do procedury **succ** , *wymuszane przez procedurę* **findall**– *po każdym wykonaniu* **succ** *jest wykonywana procedura* **score**, *w celu określenia dla uzyskanego węzła potomnego oceny wg przyjętej heurystyki typu A*: dotychczasowego kosztu,* **g(<*stan*>)** , *oraz sumarycznej oceny* **f(<*stan*>)** . *Jeśli pierwsze odwołanie do procedury* **succ** *zakończy się niepowodzeniem (brak węzłów potomnych dla rozpatrywanego węzła), to wynik* **NewNodes** *zwracany przez procedurę jest listą pustą. Procedura* **succ** *jest składnikiem dołączanej do programu definicji przestrzeni stanów –patrz założenia do program podane wyżej*

Procedura score

Procedura określa składniki oceny stanu **f(<*stan*>)**

Postać wywołania :

score(<*opis stanu*>, <*koszt przypisany do stanu rodzica*>,
<*koszt przejścia między stanami*>, Cost, Score)

Wynik : sumaryczna ocena **f(<*stan*>)** *zostaje związana ze zmienną* **Score,**
a.*składnik* **g(<*stan*>)** *zostaje zwiazany ze zmienną* **Cost**

```
score(State, ParentCost, StepCost, Cost, FScore) :-

        Cost  is ParentCost + StepCost ,

        hScore(State, HScore),

        FScore  is  Cost  + HScore  .
```

Składnik **g(<*stan*>)** *zostaje wyznaczony jako całkowity dotychczasowy koszt.*
Składnik **h(<*stan*>)** *zostaje wyznaczony przez procedurę* **hScore***, implementującą przyjętą heurystykę. Procedura* **hScore** *jest składnikiem dołączanej do programu definicji przestrzeni stanów –patrz założenia do program podane wyżej*

Procedura **insert_new_nodes**

***Procedura wstawia (kolejno) węzły z listy podanej w pierwszym argumencie do kolejki priorytetowej, w której węzły są uszeregowane wg rosnących wartości oceny* f(<*stan*>).**

Postać wywołania :

insert_new_nodes (<*lista węzłów do wstawienia*>, <*kolejka priorytetowa*>, NewQueue)

Wynik zostaje związany ze zmienną **NewQueue.**

```
insert_new_nodes( [ ], Queue, Queue) .

insert_new_nodes( [Node|RestNodes], Queue, NewQueue) :-

        insert_p_queue(Node, Queue, Queue1),

        insert_new_nodes( RestNodes, Queue1, NewQueue) .
```

Jak widać, procedura **insert_new_nodes** *jest procedurą rekurencyjną: na każdym poziomie rekurencji pobiera pierwszy wezeł z listy i wywołuje procedurę* **insert_p_queue** *w celu wstawienia węzła do kolejki priorytetowej.*

Procedura insert_p_queue

***Procedura wstawia węzeł podany w pierwszym argumencie do kolejki priorytetowej, w której węzły są uszeregowane wg rosnących wartości oceny* f(<*stan*>).**

Postać wywołania :

insert_p_queue (<*opis węzła*>, <*kolejka priorytetowa*>, Queue1)

Wynik zostaje związany ze zmienną **Queue1.**

```
insert_p_queue(Node,  [ ], [Node] )     :-   ! .


insert_p_queue(node(State, Action, Parent, Cost, FScore),
                [node(State1, Action1, Parent1, Cost1, FScore1)|RestQueue],
                    [node(State1, Action1, Parent1, Cost1, FScore1)|Rest1] )   :-

        FScore > = FScore1,  ! ,
        < ocena węzła jest zapisana w elemencie Fscore struktury opisującej wezeł >

        insert_p_queue(node(State, Action, Parent, Cost, FScore), RestQueue, Rest1) .


insert_p_queue(node(State, Action, Parent, Cost, FScore),  Queue,
                                [node(State, Action, Parent, Cost, FScore)|Queue]   .
```

Procedura build_path

Procedura buduje ścieżkę prowadzącą od stanu początkowego do stanu przypisanego do węzła zadanego w pierwszym argumencie wywołania

Postać wywołania :

build_path(<*węzeł*>, <*zbiór zamknięty*>,
<*dotychczas zbudowany odcinek ścieżki*>, Path)

Wynik zostanie związany ze zmienną **Path**

```
build_path(node(nil, _, _, _, _ ), _, Path, Path) :-   ! .

build_path(node(EndState, _ , _ , _, _ ), Nodes, PartialPath, Path)  :-

        del(Nodes, node(EndState, Action, Parent , _ , _  ) , Nodes1) ,

        build_path( node(Parent,_ ,_ , _ , _ ) , Nodes1,
                                              [Action/EndState|PartialPath],Path) .
```

Jak widać, procedura **build_path** *jest procedurą rekurencyjną. Ścieżka jest budowana od końca: opis stanu zadanego wraz z opisem akcji generującej ten stan zostaje dodany do ścieżki, w zbiorze zamkniętym zostaje odszukany węzeł poprzedzający węzeł zadany (węzeł rodzic), a następnie w rekurencyjnym wywołaniu* **build_path** *jest konstruowana dalsza (wcześniejsza) część ścieżki. Warunkiem zakończenia rekurencji jest wykrycie symbolu* **nil***, który występuje w opisie węzła początkowego jako stan poprzedzający.*

Procedura pomocnicza del

Postać wywołania :

del(<*lista*>, <*element*>, ResultList)

Procedura usuwa element podany w drugim argumencie z listy podanej w pierwszym argumencie. Wynik zostaje związany ze zmienną podaną w trzecim argumencie wywołania.

```
del([X|R],X,R).

del([Y|R],X,[Y|R1]) :-

        X\=Y,

        del(R,X,R1).
```

****

KOD BEZ OBJAŚNIEŃ

****

```
start_A_star( InitState, PathCost) :-
        score(InitState, 0, 0, InitCost, InitScore) ,
        search_A_star( [node(InitState, nil, nil, InitCost , InitScore ) ], [ ], PathCost) .


search_A_star(Queue, ClosedSet, PathCost) :-
        fetch(Node, Queue, ClosedSet , RestQueue),
        continue(Node, RestQueue, ClosedSet, PathCost).


continue(node(State, Action, Parent, Cost, _ ) , _  ,  ClosedSet,
                                                    path_cost(Path, Cost) ) :-
        goal( State), ! ,
        build_path(node(Parent, _ ,_ , _ , _ ) , ClosedSet, [Action/State], Path) .

continue(Node, RestQueue, ClosedSet, Path)   :-
        expand( Node, NewNodes),
        insert_new_nodes(NewNodes, RestQueue, NewQueue),
        search_A_star(NewQueue, [Node | ClosedSet ], Path).
```

```
fetch(node(State, Action,Parent, Cost, Score),
                    [node(State, Action,Parent, Cost, Score) |RestQueue],
                                                        ClosedSet,  RestQueue) :-

        \+member(node(State, _ ,_  , _ , _ ) , ClosedSet),   ! .


fetch(Node, [ _ |RestQueue], ClosedSet, NewRest) :-

        fetch(Node, RestQueue, ClosedSet , NewRest).




expand(node(State, _ ,_ , Cost, _ ), NewNodes)  :-

        findall(node(ChildState, Action, State, NewCost, ChildScore) ,
                    (succ(State, Action, StepCost, ChildState),
                       score(ChildState, Cost, StepCost, NewCost, ChildScore) ) ,

                                                                        NewNodes),

                                                                        ! .




score(State, ParentCost, StepCost, Cost, FScore)  :-

        Cost is ParentCost + StepCost ,

        hScore(State, HScore),

        FScore is Cost + HScore .
```

```
insert_new_nodes( [ ], Queue, Queue) .

insert_new_nodes( [Node|RestNodes], Queue, NewQueue) :-

        insert_p_queue(Node, Queue, Queue1),

        insert_new_nodes( RestNodes, Queue1, NewQueue) .


insert_p_queue(Node,  [ ], [Node] )     :-   ! .


insert_p_queue(node(State, Action, Parent, Cost, FScore),
                [node(State1, Action1, Parent1, Cost1, FScore1)|RestQueue],
                        [node(State1, Action1, Parent1, Cost1, FScore1)|Rest1] )  :-

        FScore >= FScore1,  ! ,

        insert_p_queue(node(State, Action, Parent, Cost, FScore), RestQueue, Rest1) .


insert_p_queue(node(State, Action, Parent, Cost, FScore),  Queue,
                                [node(State, Action, Parent, Cost, FScore)|Queue]) .




build_path(node(nil, _, _, _, _ ), _, Path, Path) :-   ! .

build_path(node(EndState, _ , _ , _, _ ), Nodes, PartialPath, Path)  :-

        del(Nodes, node(EndState, Action, Parent , _ , _  ) , Nodes1) ,

        build_path( node(Parent,_ ,_ , _ , _ ) , Nodes1,
                                                [Action/EndState|PartialPath],Path) .



del([X|R],X,R).

del([Y|R],X,[Y|R1]) :-

        X\=Y,

        del(R,X,R1).
```