ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ:

ОСНОВНЫЕ АЛГОРИТМЫ ЭВРИСТИЧЕСКОГО ПОИСКА

*** * ***

Процедура ***TOTAL_SEARCH*** полного перебора - поиска вширь от начального состояния StartState. Определения вспомогательных функций OPENING, SOLUTION, IS_GOAL, CONNECT зависят от конкретной поисковой задачи. Предполагается, что OPENING порождает для заданного состояния задачи список дочерних вершин-состояний. Каждое состояние задается списком, первый элемент которого - уникальный идентификатор состояния (число или атом), а второй элемент - собственно описание состояния задачи.

```
(defun TOTAL_SEARCH(StartState)
     (prog (Open  Closed  Current
              Deslist  Reflist  Goal)
         (setq Open (list (list 'S0 StartState)))
       TS (cond( (null Open) (return() )))
           (setq Current (car Open))
           (setq Open (cdr Open))
           (setq Closed (cons Current Closed))
          (setq Deslist (OPENING Current))
           (cond( (setq Goal(CHECK_GOALS Deslist))
                    (return (SOLUTION Goal Reflist)) ))
          (setq Deslist (RETAIN_NEW Deslist))
          (cond( (null Deslist) (go TS) ))
           (setq Open (append Open Deslist))
          (setq Reflist(append (CONNECT Deslist Current)
                               Reflist))
          (go TS) ))
```

Вспомогательная функция, проверяющая, есть ли среди дочерних состояний целевые. Значение функции - одно из целевых состояний, если таковое найдено, и () в ином случае.

```
(defun CHECK_GOALS (Dlist)
      (cond((null Dlist) nil)
           ((IS_GOAL(car Dlist)) (car Dlist))
           (t (CHECK_GOALS (cdr Dlist)) )) )
```

Вспомогательная функция, оставляющая в списке дочерних состояний только те, которые не порождались ранее - тем самым исключается зацикливание при поиске в произвольном графе.

```
(defun RETAIN_NEW (Dlist)
     (prog (D)
        (cond( (null Dlist) (return()) )
            (t (setq D (car Dlist))
              (cond((or (member D Open)(member D Closed))
                    (return(RETAIN_NEW (cdr Dlist)) ))
                 (t (return(cons D (RETAIN-NEW(cdr Dlist))
                           )) )) )) ))
```

*** * ***

Процедура ***LIMITED_DEEP_SEARCH*** ограниченного перебора вглубь от начального состояния StartState, LimDepth - граничная глубина. Предполагается, что в каждом списке-описании состояния (кроме идентификатора и собственно состояния задачи) содержится также третий элемент - глубина этого состояния-вершины в дереве поиска.

```
(defun LIMITED_DEEP_SEARCH (StartState LimDepth)
     (prog (Open Closed Deslist Reflist
             Current Depth)
         (setq Open (list (list 'S0 StartState 0)))
       LS (cond( (null Open) (return() )))
          (setq Current (car Open))
          (setq Open (cdr Open))
          (setq Closed (cons Current Closed))
          (setq Depth (caddr Current))
         (cond((IS_GOAL Current)
                   (return (SOLUTION Current Reflist)) ))
          (cond( (eql Depth LimDepth) (go LS) ))
          (setq Deslist (OPENING Current))
          (setq Deslist (RETAIN_NEW Deslist))
           (cond((null Deslist) (go LS) ))
          (setq Open (append (ADD_DEPTH(add1 Depth)Deslist)
                             Open))
          (setq Reflist(append (CONNECT Deslist Current)
                               Reflist))
          (go LS) ))
```

Вспомогательная функция, устанавливающая глубину всех дочерних состояний.

```
(defun ADD_DEPTH (Dn Dlist)
      (cond((null Dlist) ())
           (t (cons (list(caar Dlist)(cadar Dlist) Dn)
                    (ADD_DEPTH Dn (cdr Dlist)) ))) )
```

*** * ***

Процедура ***ORDERED_SEARCH_FOR_TREE*** упорядоченного перебора в пространствах-деревьях на основе эвристической оценочной функции ESTIMATE. Вычисляемая для каждого состояния эвристическая оценка хранится как четвертый элемент списка описания состояния, а третий элемент этого списка, как и прежде, - глубина состояния в дереве поиска (последняя часто используется при подсчете эвристической оценки).

```
(defun ORDERED_SEARCH_FOR_TREE(StartState)
     (prog (Open Closed Deslist Reflist
              Current Depth)
         (setq Open (list (list 'S0  StartState  0
                                (ESTIMATE StartState))))
       OT (cond( (null Open) (return() )))
          (setq Current (car Open))
          (setq Open (cdr Open))
          (setq Closed (cons Current Closed))
          (setq Depth (caddr Current))
         (cond( (IS_GOAL Current)
                   (return (SOLUTION Current Reflist)) ))
         (setq Deslist (OPENING Current))
         (cond( (null Deslist) (go OT) ))
        (setq Open (MERGE (EST_LIST (ADD_DEPTH(add1 Depth)
                                                Deslist))
                          Open))
         (setq Reflist(append (CONNECT Deslist Current)
                              Reflist))
         (go OT) ))
```

Вспомогательная функция, устанавливающая глубину дочерних вершин и вычисляющая их эвристическую оценку.

```
(defun ADD_DEPTH_EST (Dn Slist)
      (cond((null Slist) ())
          (t (cons(list (caar Slist) (cadar Slist) Dn
                     (ESTIMATE(list (cadar Slist) Dn)) )
                 (ADD_DEPTH_EST (cdr Slist)) ) )) )
```

Вспомогательная функция, выполняющая слияние двух упорядоченных (по невозрастанию эвристической оценки) списков состояний в результирующий упорядоченный список.

```
(defun MERGE (L1 L2)
      (cond( (null L1) L2)
           ( (null L2) L1)
           ((gt (cadddar L1) (cadddar L2))
                (cons(car L2) (MERGE L1 (cdr L2)) ))
           (t (cons(car L1) (MERGE (cdr L1) L2) ))))
```

*** * ***

ПРИМЕР реализации вспомогательных функций ***OPENING***, ***IS_GOAL***, ***SOLUTION***, ***CONNECT***, ***ESTIMATE*** для игры в "8". 8 фишек в квадрате 3x3 и одно пустое место, на которое можно двигать другие фишки (обозначается ниже как #).

Состояние задачи (конфигурация игры) задается списком из следующих элементов:

1) идентификатор состояния (используются идентификаторы S1, S2, S3, генерируемые встроенной функцией gensym);

2) собственно описание состояния - список из номеров фишек, записанных последовательно по рядам квадрата. В этот список включается также - в качестве первого элемента - название оператора движения пустого места, или "пустышки" ('right, 'left, 'up, 'down), который и привел к данному состоянию (этот элемент нужен, чтобы сделать функцию OPENING более разумной);

3) число-глубина состояния-вершины в дереве перебора;

4) числовая эвристическая оценка состояния

Например, (S3 ('left 2 8 3 1 6 4 # 7 5) 1 6) - цель S3, полученная сдвигом "пустышки" влево, находящаяся на глубине 1 и имеющая эвристическую оценку 6. Отметим, что оценка 4) используется только в алгоритме упорядоченного перебора, а глубина 3) - в алгоритмах упорядоченного перебора и ограниченного перебора вглубь.

Предлагаемая ниже реализация функций для игры в "8" может быть пригодна и для игры в 15", так как размер стороны игрового квадрата =3 используется в ней как переменная Size.

```
(defun OPENING (State)
     (prog(Op St  Dlist K  I J El)
          (setq St (cadr State))
         (setq Op (car St))  (setq St (cdr St))
          (setq State St) (setq K 0)
       ; поиск порядкового номера K "пустышки" в списке:
        OP (setq El (car St))
          (setq K (add1 K))
          (cond ((neq El '#) (setq St(cdr St)) (go OP) ))
       ; вычисление номера ряда и номера столбца "пустышки":
            (setq I(add1(mod K Size))) (setq J (rem K Size))
       ; поочередно проверка возможности движения "пустышки"
       ; вправо/влево/вверх/вниз (за счет анализа оператора
       ; Op исключаем в дереве поиска "простые циклы", т.е.
       ; случаи, когда после применения двух операторов
       ; возвращаемся в исходное состояние):
(cond((and (neq Op 'left) (< J Size))
                   (ADD_STATE 'right K (add1 K)) ))
          (cond((and (neq Op 'right) (> J 1))
                   (ADD_STATE 'left K (sub1 K)) ))
          (cond((and (neq Op 'down) (> I 1))
                   (ADD_STATE 'up K (- K Size)) ))
          (cond((and (neq Op 'up) (< I Size))
                   (ADD_STATE 'down K (+ K Size)) ))
          (return Dlist) ))
```

Вспомогательная функция, добавляющая в список дочерних вершин описание нового состояния (двухэлементный список, первый элемент - идентификатор Si - генерирует функция gensym).

```
(defun  ADD_STATE (Op K1 K2)
    (push(list(gensym)
          (cons Op
           (cond((< K1 K2)
                 (EXCHANGE State 1 '# K1 (NTH K2 State)K2))
            (t (EXCHANGE State 1(NTH K2 State)K2 '# K1))) ))
          Dlist))
```

Вспомогательная функция, выбирающая элемент списка List по его порядковому номеру K (в MULISPe - встроенная функция)

```
;  (defun NTH (K List)
;     (prog(El (M 0) )
;      F (cond((null List) 'error))
;         (setq El(car List))  (setq M(add1 M))
;         (cond((eql M K) (return El) ))
;        (go F)  ))
```

Рекурсивная вспомогательная функция, формирующая новое состояние игры путем перестановки заданных элементов исходного состояния-списка List, K служит для просмотра этого списка.

```
(defun EXCHANGE (List K Elem1 K1 Elem2 K2)
     (cond((eql K K1) (cons Elem2
            (EXCHANGE(cdr List)(add1 K)Elem1 K1 Elem2 K2) ))
          ((eql K K2)(cons Elem1 (cdr List)) )) )
```

Вспомогательный предикат: является ли состояние целевым?

```
(defun IS_GOAL(State)
       (equal (cdadr State) Goalstate ))
```

Вспомогательная функция, определяющая последовательность (названий) операторов, преобразующих начальное состояние (идентификатор - S0) в целевое Goal. Reflist - список указателей-связей между состояниями, каждый указатель есть трехэлементный список из идентификатора состояния, идентификатора дочернего состояния и названия оператора, переводящего первое состояние во второе.

```
(defun SOLUTION (Goal Reflist)
     (prog (Sollist Gi Edge)
         (setq Gi (car G))
       S (cond( (eq Gi 'S0) (return Sollist) ))
         (setq Edge (LOOK_FOR Gi Reflist))
         (setq Sollist (cons(caddr Edge) Sollist))
         (setq Gi (car Edge))
         (go S) ))

   (defun LOOK_FOR (Id List)
      (cond( (null List) (quote error))
           ((eq ID (cadar List)) (car List))
           (t (LOOK_FOR Id (cdr List)) )) )
```

Вспомогательная функция, формирующая список указателей от текущего состояния Curr к заданным (в списке Dlist) дочерним состояниям.

```
(defun CONNECT (Dlist Curr)
      (prog (D Di Ci Rlist)
         C (setq D (car Dlist)) (setq Dlist (cdr Dlist))
           (setq Di (car D))  (setq Ci (car Curr))
          (setq Rlist (cons (list Ci Di (caadr D))Rlist) )
          (cond ((null Dlist) (return Rlist) ))
           (go C) ))
```

Вспомогательная функция, вычисляющая эвристическую оценку заданного состояния. Значение функции - сумма числа фишек, стоящих не на своих местах, и длины пути (глубины) оцениваемого состояния в дереве поиска.

```
(defun ESTIMATE (S)
     (prog(Len G  N  E1 E2)
         (setq Len (cadr S)) (setq N 0)
         (setq S (cdar S)) (setq G Goalstate)
      ES (cond((null S) (return (+ N Len)) ))
         (pop E1 S) (pop E2 G)
         (cond( (eql E1 E2) (setq N (add1 N)) ))
         (go ES) ))
```

Блок, инициализирующий эвристический поиск решения игры в "8" (сначала устанавливаются переменные-параметры встроенной функции gensym)

```
(prog(Goalstate Initstate Size)
        (setq *gensym-prefix* 'S)
        (setq *gensym-count* 1)
        (setq Size 3)
        (setq Goalstate '(1 2 3 8 # 4 7 6 5))
        (setq Initstate '(? 2 8 3 1 6 4 7 # 5))
        (TOTAL_SEARCH Initstate) )
```

*** * ***

%

Плэнерская функция ***SEARCH_DEEP_DOWN***, реализующая перебор вглубь от заданного состояния St (без ограничений), ее значение - список всех состояний-вершин, лежащих на (решающем) пути от St до цели;

```
[define SEARCH_DEEP_DOWN (lambda(St)
      [prog (Dst Reslist)
           [set Dst .St] [set Reslist (.St)]
         S [set Dst [among [OPENING .Dst]]]
           [set Reslist (!.Reslist .Dst)]
           [cond( [IS_GOAL .Dst] [return .Reslist])
                ( t [go S]) ]   ])]
```

%

МИНИМАКСНАЯ ПРОЦЕДУРА. Instate - исходная позиция игры, для которой ищется лучший ход, N - глубина поиска (т.е. количество ходов вперед). Значение функции MIN_MAX - дочерняя для Instate позиция, соответствующая наилучшему ходу

```
[define MIN_MAX (lambda (Instate N)
   [SELECT_MAX [ VALUE_POS .Instate  0 T ]] )]
```

%

VALUE_POS - основная рекурсивная функция, которая оценивает позицию Position, находящуюся на глубине Depth и имеющую тип Deptype ("И" при Deptype=T и "ИЛИ" при Deptype=()). Значение функции VALUE_POS:

- числовая оценка Position при Depth>0
- список ходов (точнее, дочерних для Position позиций) с их числовыми оценками при Depth=0

При работе используются вспомогательные функции, определение которых зависит от конкретной игры:

- OPENING - вычисление для заданной позиции игры списка дочерних вершин/позиций,
- STAT_EST - статическая оценка заданной позиции

```
[define VALUE_POS (lambda (Position Depth Deptype)
   [prog (D                    %дочерняя позиция;
           (Movelist()) %список ходов(дочерних позиций);
         Pvalue Dvalue) %оценки текущей и дочерней позиций;
% 1: развилка, включающая все дочерние позиции (список ()
 добавлен в развилку, чтобы "поймать"  момент ее закрытия);
    [set D [among (<OPENING .Position> ())]]
% 2: закрытие развилки и возврат подсчитанной оценки ;
    [cond ([empty .D]
            [return [cond ([eq .Depth 0] .Movelist)
                          (t .Pvalue) ]] )]
% 3: вычисление оценки очередной дочерней позиции ;
    [cond ([eq .Depth [- .N 1]] [set Dvalue [STAT_EST .D]])
          (t [set Dvalue [VALUE_POS .D
                                    [+ 1 .Depth]
                                    [not .Deptype]]] )]
% 4: пересчет предварительной оценки текущей позиции по
     минимаксному принципу ;
    [cond ([hasval Pvalue] [pset Pvalue
                    [cond(.Deptype [max .Pvalue .Dvalue])
                        (t [min .Pvalue .Dvalue]) ]])
          (t [pset Pvalue .Dvalue]) ]
% 5: для исходной позиции пересчитываем список ходов
     (дочерних позиций) с их оценками ;
    [cond ([eq .Depth 0]
             [pset Movelist (!.Movelist (.Dvalue .D)) ])]
% 6: возврат к другой альтернативе развилки ;
    [fail]  ] )]
```

%

SELECT_MAX - вспомогательная процедура выбора из списка List, содержащего ходы-позиции с их оценками, элемента с наибольшей оценкой

```
[define SELECT_MAX (lambda (List)
  [prog (Elem Max_elem )
     [fin Max_elem .List]
   SM [cond ([fin Elem List] [return [2 .Max_elem]])
            ([gt [1 .Elem] [1 .Max_elem]]
               [set Max_elem .Elem]) ]
     [go SM]  ] )]
```

%

АЛЬФА-БЕТА ПРОЦЕДУРА. Instate - позиция игры, для которой ищется лучший ход. N - глубина поиска (количество ходов вперед). Значение функции ALPHA_BETA - дочерняя для Instate позиция, соответствующая наилучшему ходу

```
[define ALPHA_BETA (lambda (Instate N)
   [SELECT_MAX [NEXT_MOVE .Instate 0 T () ]] )]
```

%

NEXT_MOVE - основная рекурсивная функция, которая оценивает позицию Pos, находящуюся на глубине Depth и имеющую тип Deptype ("И" при Deptype=T и "ИЛИ" при Deptype=()). Последний аргумент функции, Way, - это список-путь из всех позиций (вместе с их предварительными оценками, если они уже есть), начиная от позиции, предшествующей Pos, двигаясь вверх в дереве игры до исходной позиции Instate.

Значение функции NEXT_MOVE:

- числовая оценка позиции Pos при Depth>0
- список ходов (точнее, дочерних для Pos позиций) с их числовыми оценками при Depth=0

При работе используются вспомогательные функции, определение которых зависит от конкретной игры:

- OPENING - вычисление списка дочерних вершин/позиций;
- STAT_EST - статическая оценка заданной позиции.

```
[define NEXT_MOVE (lambda (Pos  Depth Deptype  Way)
  [prog(D (Movelist()) Pvalue Dvalue)
% 1: развилка, включающая все дочерние позиции ( список ()
 добавлен в развилку, чтобы "поймать"  момент ее закрытия);
    [set D [among (<OPENING .Pos> ())]]
% 2: закрытие развилки и возврат подсчитанной оценки ;
    [cond ([empty .D]
            [return[cond([eq .Depth 0] .Movelist)
                         (t .Pvalue)] ] )]
% 3: проверка возможности отсечения и отсечение-возврат на
 предыдущий уровень рекурсии  (встроенная функция  tempex
 реализует досрочный выход с заданным значением из заданной
 функции, при этом уничтожает все развилки и выполняет все
 обратные операторы,возникшие после входа в заданную функцию;
    [cond([and [neq .Depth 0] [hasval Pvalue]]
            [cond(.Deptype [cond([NOT_LESS .Pvalue .Way]
                                 [tempex .Pvalue NEXT_MOVE])])
                 (t [cond([NOT_GREATER .Pvalue .Way]
                          [tempex .Pvalue NEXT_MOVE])] )])]
% 4: вычисление оценки очередной дочерней позиции ;
    [cond ([eq .Depth[- .N 1]] [set Dvalue [STAT_EST .D]])
         (t [set Dvalue [NEXT_MOVE .D [+ 1 .Depth]
                                    [not .Deptype]
                          [cond([hasval Pvalue]
                                  ((.Pvalue .Pos) !.Way))
                               (t ((.Pos) !.Way))] ])]
% 5: пересчет предварительной оценки текущей позиции по
   минимаксному принципу ;
    [cond([hasval Pvalue] [Pset  Pvalue
                   [cond(.Deptype[max .Pvalue .Dvalue])
                         (t [min .Pvalue .Dvalue])] ])
         (t[pset Pvalue .Dvalue] )]
% 6: для начальной позиции пересчитываем список ходов
      (дочерних позиций) с их оценками ;
    [cond([eq .Depth 0]
               [pset Movelist (!.Movelist(.Dvalue .D))] )]
% 7: возврат к другой альтернативе развилки ;
    [fail] ] )]
```

%

NOT_LESS - вспомогательная функция проверки выполнения условия бета-отсечения, List - список предыдущих позиций с оценками (если они есть).

Замечание: вместо условия [eq[length .Elem]2] (имеет ли позиция-элемент списка List предварительную оценку?) можно использовать более эффективное [num[1 .Elem]] , эти условия эквивалентны, если описание позиции для конкретной игры имеет списочный вид

```
[define NOT_LESS (lambda (Value List)
  [prog (Elem)
      [fin Elem List]
    NL [cond([and [eq[length .Elem]2] [le[1 .Elem] .Value]]
             [return t] )
          ([fin Elem List] [return ()] )
          ([fin Elem List] [return ()] )]
       [go NL] ] )]
```

%

NOT_GREATER - аналогичная функция проверки выполнения условия альфа-отсечения

```
[define NOT_GREATER (lambda (Value List)
  [prog (Elem)
      [fin Elem List]
    NG [cond([and [eq[length .Elem]2] [ge[1 .Elem] .Value]]
             [return t] )
          ([fin Elem List] [return ()] )
          ([fin Elem List] [return ()] )]
       [go NG] ] )]
```