**Лабораторная работа №4. Разработка интеллектуальной компоненты с использованием подхода в пространстве состояний**

**Цель работы:**использовать методы, существующие в пространстве состояний, чтобы построить игровую компьютерную программу.

**Постановка задачи.**Для примера рассмотрим пример игра в «8» и «15». Дано начальное состояние в виде массивов 3х3 и 4х4, целевое состояние точно такой же форме, но с другими данными [10]. Эти состояния показаны на рисунке 2.1.а,б.

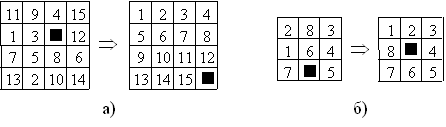


Рисунок 2.1- Состояния игры «8» и «15»   в виде массива чисел

В качестве формы описания *состояний* в данном примере выступает массив размером 3х3. А в качестве *оператора* движение пустой клетки - влево, вверх, вправо, вниз. *Методы полного перебора.* *Поиск в ширину.*(BFS - *Breadth-first search*). В методе полного перебора вершины раскрываются в том порядке, в котором они строятся. В качестве механизма реализаций или в качестве обработки списков для метода поиска в ширину в алгоритмах программирования выбран механизм «*очередь*». Для списка *open* реализуется механизм *очередь*, или его называют структура *FIFO* *«first-in-first-out»*,  т.е. «первым пришел – первым обслужен». Состояния добавлются в список справа  и удаляются с левого конца списка. *Поиск в глубину* (*Depth-first search, DFS*).  В методах перебора в глубину  прежде всего раскрываются те   вершины, которые были построены последними. Одним из классических алгоритмов поиска на графе пространства состояний служит метод поиска с возвратами *бектрекинг*[11]. Этот способ предлагает опреде­ленную организацию перебора всех возможных вариантов решения задачи, число которых может быть велико. Суть бектрекинга состоит в том, чтобы в каждой точке процесса решения задачи, где существует несколько априори равноправных альтернативных вариантов (путей) дальнейшего продолжения, выбрать один из них и следовать ему, предварительно запомнив другие альтернативы,  для того, чтобы в случае неуспешности выбранного варианта-пути вернуться в точку выбора и выбрать для продолжения решения другой возможный вариант-путь. От программиста требуется лишь определение точек бектрекинга с нужными альтернативами и инициация в необходимый момент процесса возврата. *Эвристический метод.*Этот способ позволяет во всех случаях найти некоторый путь от начальной вершины к целевой, стоимость которого минимальна. Предполагается, что нам задана функция стоимости *c(ni,nj),* дающая стоимость перехода от вершины *ni* к некоторой следующей за ней вершине *nj*. В методе равных цен для каждой вершины *n* в дереве перебора нам нужно помнить стоимость пути, построенного от начальной вершины *s* к вершине *n*. Пусть *g(n)* -  стоимость от вершины *s* к вершине *n* в дереве перебора. В случае дерева перебора мы можем быть уверены, что *g(n*) является к тому же стоимостью того пути, который имеет *минимальную* стоимость (т.к. этот путь единственный). Алгоритм, работающий по методу равных цен, может быть также использован для поиска путей минимальной длины, если просто положить стоимость каждого ребра равной*единице*. Если сделать резюме, то в данном методе выбираем единичную функцию *f(n)=g(n)+w(n)* – та цена, которую дают каждой вершине.  Здесь *g(n)* – длина пути, *g(0)* – начальный путь, *g(n)= g(n-1)+1=> g(1)=0+1.**w(n)* – число фишек, которые находятся не на своем месте. Как правило, выбираем тот, что дешевле. Рассмотрим реализацию этого метода  на примере игры в «8» на Лиспе. Исходные тексты на Лиспе приведены в [11]. Он выполнен с помощью  лисповской функции *HEURISTIC\_SEARCH*, реализующей эвристический поиск в пространстве состояний с использованием эвристической оценочной функции *EST*, зависящей от конкретной поисковой задачи. Отметим, что на шаге 3, выбирая из списка *Open* первый элемент-вершину, мы тем самым выбираем вершину с минимальной оценкой, поскольку список *Open* всегда упорядочен по неубыванию хранящихся в нем оценок вершин-состояний (это обеспечивается вспомогательной функцией *MERGE*).

 (defun HEURISTIC\_SEARCH(StartState)

   (prog (Open Closed Current

          Deslist ;список дочерних вершин;

          Reflist ;список указателей;

          Depth   ;глубина текущей вершины;)

 ;Шаг 1:; (setq Open (list(list 'S0 StartState 0

 (EST (list StartState 0)) )))

 ;Шаг 2:;   HS (cond ((null Open) (return())))

 ;Шаг 3:;   (setq Current (car Open))

                                  (setq Open (cdr Open))

                                  (setq Closed (cons Current Closed))

                                  (setq Depth (caddr Current))

 ;Шаг 4:;   (cond ((IS\_GOAL Current)

            (return (SOLUTION Current Reflist))))

 ;Шаг 5:;   (setq Deslist (OPENING Current))

 ; Исключение повторных вершин-состояний:;

                                  (setq Deslist (RETAIN\_NEW Deslist))

                                  (cond ((null Deslist) (go НS)))

 ;Шаг 6:;   (setq Open (MERGE (ADD\_DEPTH\_EST (add1 Depth) Deslist) Open))

  (setq Reflist (append (CONNECT Deslist Current) Reflist))(go HS) ))

Данная лисповская функция использует вспомогательные функции *OPENING, SOLUTION, IS\_GOAL, CONNECT, EST* и функцию *RETAIN\_NEW*.  Функция *OPENING* порождает для своего единственного аргумента список дочерних вершин-состояний (он может оказаться и пустым). Описание каждого состояния задается списком, первый элемент которого – это уникальный идентификатор состояния, а второй элемент – собственно описание состояния задачи. Идентификаторы состояний необходимы для построения указателей функцией *CONNECT*. Функция *CONNECT*с двумя аргументами,  списком порожденных дочерних вершин и текущей (только что раскрытой, родительской) вершиной – генерирует список указателей от каждой дочерней вершины к исходной родительской. Функция *SOLUTION*с двумя аргументами, найденными целевым состоянием и списком всех указателей дерева перебора, вырабатывает список-решение задачи (решающий путь). Предикат *IS\_GOAL* проверяет, является ли ее аргумент целевым состоянием. В случае положительного исхода проверки значением функции является само проверяемое состояние, в ином случае значение равно цели. В алгоритмах поиска применяется также вспомогательная рекурсивная функция *RETAIN\_NEW*, которая оставляет в списке дочерних состояний *Dlist* 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 ()) ))

    (setq D (car Dlist))

    (cond ((or (member D Open) (member D Closed))

             (return (RETAIN\_NEW (cdr Dlist))) ))

    (return (cons D (RETAIN-NEW (cdr Dlist)))) ))

Другая вспомогательная рекурсивная функция *CHECK\_GOALS* проверяет, есть ли среди дочерних состояний целевые. Значением этой функции является целевое состояние, если таковое найдено, и пустой список в ином случае:

 (defun CHECK\_GOALS (Dlist)

   (cond ((null Dlist) ())

         ((IS\_GOAL (car Dlist)) (car Dlist))

         (t (CHECK\_GOALS (cdr Dlist))) ))

 Используются также еще две вспомогательные функции – *ADD\_DEPTH\_EST и MERGE*. Первая функция устанавливает глубину дочерних вершин и вычисляет их эвристическую оценку. Вторая функция выполняет слияние двух упорядоченных (по невозрастанию эвристической оценки) списков состояний в результирующий упорядоченный список:

 (defun *ADD\_DEPTH\_EST* (Dn Slist)

   (cond ((null Slist) ())

         (t (cons (list (caar Slist) (cadar Slist) Dn

                        (EST (list (cadar Slist) Dn)) )

                  (ADD\_DEPTH\_EST Dn (cdr Slist))) ) ))

  (defun *MERGE* (L1 L2)

   (cond ((null L1) L2)

         ((null L2) L1)

         ((> (car (cdddar L1)) (car(cdddar L2)))

            (cons (car L2) (MERGE L1 (cdr L2))))

         (t (cons (car L1) (MERGE (cdr L1) L2))) ))

К основным терминам для этой лабораторной работы относятся следующие термины: состояние, формы описания состояний, оператор, описание начального состояния, описание целевого состояния, запись в виде графа,  раскрытие вершин, метод поиска в глубину (*Depth-first search, DFS****),***списки *open и closed*в методах поиска, метод поиска в ширину*(BFS*, *Breadth*-*first search*), структура очередь (*FIFO* *«first-in-first-out»*), структура стек (*LIFO* *«last-in-first-out»*),эвристический метод поиска*.*

**Варинты задания**

1.     Создать компьютерную программу с методом поиска в глубину.

2.     Создать компьютерную программус методом поиска в ширину.

3.     Создать программу, использующую эвристический метод.

4.     Создать программу, использующий «жадный» алгоритм поиска.

5.     Создать программу, использующий процедуру  «Минимакса».

**Контрольные вопросы**

1.     В чем различие методов поиска в глубину и ширину?

2.     Преимущество эвристических алгоритмов.

3.     Как осуществляется стратегия при поиске от данных?

4.     Как осуществляется стратегия при поиске от цели?

5.     Что означает структура: стек?

6.     Какой механизм поиска применяется при методе поиска в глубину?

7.     Какой механизм поиска применяется при методе поиска в ширину?

8.     Что означает структура: очередь?

9.     Что означает раскрытие вершин?

10. Какие бывают формы описания состояния?