Рассмотренный на прошлой лекции пример (*задача об обезьяне и банане*) показывает, сколь важен для успешного и эффективного решения задачи выбор представления задачи. Такое небольшое по размерам пространство состояний получено, в частности, вследствие того, что игнорировались все точки пола, кроме трех, соответствующих первоначальному расположению обезьяны, ящика и бананов.

Мощным приемом сужения пространств состояний является применение так называемых *схем состояний* и *схем операторов*, в которых для описаний состояний и операторов используются переменные. Тем самым схема состояния описывает целое множество состояний, а не только одно, так же как схема оператора определяет все множество действий некоторого типа. В рассмотренном нами представлении задачи об обезьяне использовались схемы операторов, но не схемы состояний.

Алгоритмы поиска решения

Классификация алгоритмов

Как уже отмечалось, поиск в пространстве состояний базируется на последовательном построении (переборе) вершин графа состояний – до тех пор, пока не будет обнаружено целевое состояние. Введем несколько терминов, которые будем использовать для описания различных алгоритмов поиска.

Вершину графа, соответствующую начальному состоянию, естественно назвать ***начальной*** вершиной, а вершину, соответствующую целевому состоянию – ***целевой***. Вершины, непосредственно следующие за некоторой вершиной, т.е. получившиеся в результате применения к последней допустимых операторов, будем называть *дочерними*, а саму исходную вершину – *родительской*. Основной операцией, выполняемой при поиске в графе, будем считать ***раскрытие вершины***, что означает порождение (построение) всех ее дочерних вершин, путем применения к соответствующему описанию состояния задачи всех допустимых операторов.

Поиск в пространстве состояний можно представить как процесс постепенного раскрытия вершин и проверки свойств порождаемых вершин. Важно, что в ходе этого процесса должны строиться (и запоминаться) *указатели* от всех возникающих дочерних вершин к их родительским. Именно эти указатели позволят восстановить путь назад к начальной вершине после того, как будет построена целевая вершина. Этот путь, взятый в обратном направлении, точнее, последовательность операторов, соответствующих дугам этого пути, и будет искомым решением задачи.

Вершины и указатели, построенные в процессе поиска, образуют поддерево всего неявно определенного при формализации задачи графа-пространства состояний. Это поддерево называется ***деревом перебора***.

Известные алгоритмы поиска в пространстве состояний можно классифицировать по различным характеристикам, а именно:

- использование эвристической информации;
- порядок раскрытия (перебора) вершин;
- полнота просмотра пространства состояний;
- направление поиска.

В соответствии с первой характеристикой алгоритмы делятся на два класса – ***слепые*** и ***эвристические***. В слепых алгоритмах поиска местонахождение в пространстве целевой вершины никак не влияет на порядок, в котором раскрываются (перебираются) вершины. В противоположность им, эвристические алгоритмы используют априорную, эвристическую информацию об общем виде графа-пространства и/или о том, где в пространстве состояний расположена цель, поэтому для раскрытия обычно выбирается более перспективная вершина. В общем случае это позволяет сократить перебор.

Два основных вида слепых алгоритмов поиска, различающихся порядком раскрытия вершин – это алгоритмы ***поиска вширь*** и ***поиска вглубь***.

Как слепые, так и эвристические алгоритмы поиска могут отличаться полнотой просмотра пространства состояний. ***Полные*** алгоритмы перебора при необходимости осуществляют полный просмотр графа-пространства и гарантируют при этом нахождение решения, если таковое существует. В отличие от полных, ***неполные*** алгоритмы просматривают лишь некоторую часть пространства, и если она не содержит целевых вершин, то искомое решение задачи этим алгоритмом найдено не будет.

В соответствии с направлением поиска алгоритмы можно разделить на ***прямые***, ведущие поиск от начальной вершины к целевой, ***обратные***, ведущие поиск от целевой вершины в направлении к начальной, и ***двунаправленные***, чередующие прямой и обратный поиск. Наиболее употребительными (отчасти, в силу их простоты) являются алгоритмы прямого поиска. Обратный поиск возможен в случае обратимости операторов задачи.

Методы слепого (полного) перебора

Слепые алгоритмы поиска вширь (*breadth_first_search*) и поиска вглубь (*depth_first_search*) отличаются тем, какая вершина выбирается для очередного раскрытия. В алгоритме перебора вширь вершины раскрываются в том порядке, в котором они строятся. В алгоритме же перебора в глубину прежде всего раскрываются те вершины, которые были построены последними.

Сначала рассмотрим эти алгоритмы для графов-пространств, являющихся деревьями (корнем дерева является начальная вершина). Затем покажем, как алгоритмы следует модифицировать для поиска в произвольных графах. Организовать перебор в деревьях проще, так как при построении нового состояния (и соответствующей вершины) можно быть уверенным в том, что такое состояние никогда раньше не строилось и не будет строиться в дальнейшем.

Перебор вширь

Базовый **алгоритм поиска вширь** состоит из следующей последовательности шагов (здесь и далее предполагаем, что начальная вершина не является целевой):

Шаг 1. Поместить начальную вершину в список нераскрытых вершин **`Open`**.

Шаг 2. Если список **`Open`** пуст, то окончание алгоритма и выдача сообщения о неудаче поиска, в противном случае перейти к следующему шагу.

Шаг 3. Выбрать первую вершину из списка **`Open`** (назовем ее **`Current`**) и перенести ее в список раскрытых вершин **`Closed`**.

Шаг 4. Раскрыть вершину **`Current`**, образовав все ее дочерние вершины. Если дочерних вершин нет, то перейти к шагу 2, иначе поместить все дочерние вершины (в любом порядке) в конец списка **`Open`** и построить указатели, ведущие от этих вершин к родительской вершине **`Current`**.

Шаг 5. Проверить, нет ли среди дочерних вершин целевых. Если есть хотя бы одна целевая вершина, то окончание алгоритма и выдача решения задачи, получающегося просмотром указателей назад от найденной целевой вершины к начальной. В противном случае перейти к шагу 2.

Конец алгоритма.

Основу этого алгоритма составляет цикл последовательного раскрытия (шаги 2-5) концевых вершин (листьев) дерева перебора, хранящихся в списке **`Open`**. Алгоритм поиска вширь является полным. Можно также показать, что при переборе вширь непременно будет найден самый короткий путь к целевой вершине, причем быстрее, чем другие решающие пути – при условии, что этот путь вообще существует. Если же решающего пути нет, то (в случае конечных деревьев-пространств) будет сообщено о неуспехе поиска, в случае же бесконечных пространств алгоритм не кончит свою работу.

На рис. 5 приведено дерево, построенное в результате применения алгоритма поиска вширь к некоторой начальной конфигурации игры в восемь, причем выполнение алгоритма прервано после построения первых 12 вершин (при этом раскрыто 6 вершин). В вершинах дерева помещены соответствующие описания состояний. Эти вершины занумерованы в том порядке, в котором они были построены в ходе поиска. На следующем шаге цикла алгоритма будет раскрываться одна из вершин с номерами 6, 7 или 8, поскольку они расположены в начале списка нераскрытых вершин.

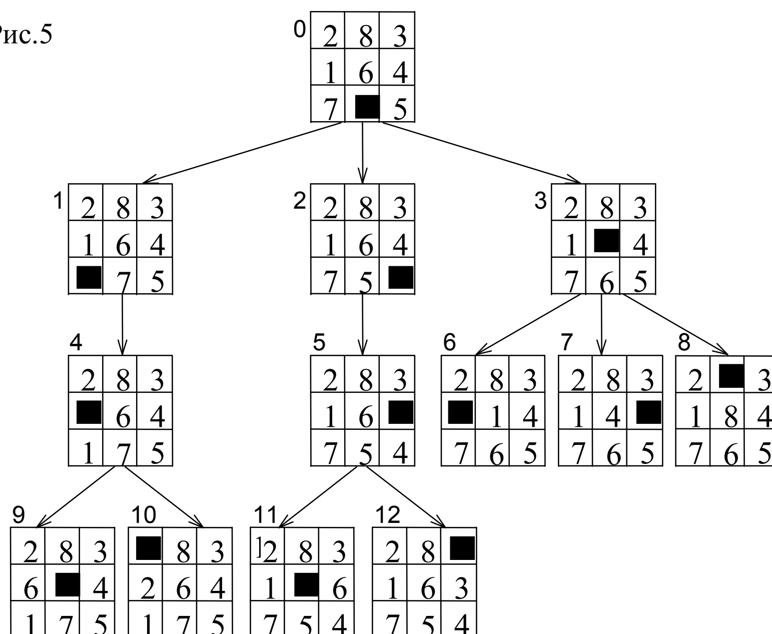


Рис.5

Считаем, что порядок построения дочерних вершин соответствует следующему зафиксированному порядку перемещения пустой клетки («пустышки»): влево/вправо/вверх/вниз. Предполагается также, что используемая алгоритмом операция раскрытия вершин организована таким образом, что она не порождает никакое состояние-вершину, построенную ранее и являющуюся родительской для раскрываемой вершины. Тем самым в дереве перебора нет дублирования одного и то же состояния в вершинах, имеющих общего соседа-вершину.

В приведенном примере алгоритм перебора вглубь, сформулированный для деревьев-пространств, применялся к пространству состояний, являющемуся графом (в котором могут быть циклы). В некоторых случаях это допустимо, т.е. алгоритм находит решение, если оно есть, и заканчивает работу. Построенная алгоритмом структура из вершин и указателей всегда образует дерево (дерево перебора), поскольку указатели от дочерних вершин ссылаются только на одну порождающую вершину. Но в случае поиска на произвольном графе (и в этом – отличие от деревьев-пространств) одно и тоже состояние может быть продублировано в разных частях полученного дерева перебора. В примере игры в восемь по принятому предположению об операции раскрытия исключалось только повторное возникновение состояний, встречавшихся два шага вверх по дереву перебора, другие же, более далекие друг от друга повторы одного и того же состояния остаются возможными. В случае поиска в графе состояний общего вида он как бы разворачивается при поиске в дерево путем дублирования некоторых его частей. Если это дублирование неоднократное (из-за циклов в графе), то оно может привести к зацикливанию базового алгоритма поиска вширь.

Перебор вглубь

Для формулировки алгоритма поиска вглубь необходимо определить понятие ***глубины вершины*** в дереве поиска. Это можно сделать следующим образом:

- глубина *корня* дерева равна нулю;
- глубина каждой *некорневой* вершины на единицу больше глубины ее родительской вершины.

В алгоритме перебора вглубь раскрытию в первую очередь подлежит вершина, имеющая наибольшую глубину. Такой принцип может привести к бесконечному процессу – это происходит, если пространство состояний бесконечно, и поиск вглубь пошел по ветви дерева, не содержащей целевую вершину. Поэтому необходимо то или иное ограничение этого процесса, самый распространенный способ – ограничить глубину просмотра дерева. Это означает, что в ходе перебора можно строить только вершины, глубина которых не превышает некоторую заданную ***граничную глубину***. Тем самым, раскрытию в первую очередь подлежит вершина наибольшей глубины, но расположенная выше фиксированной границы. Соответствующий алгоритм поиска называется ***ограниченным перебором вглубь***.

Основные шаги базового **алгоритма ограниченного перебора вглубь** (с граничной глубиной **D**) таковы:

Шаг 1. Поместить начальную вершину в список нераскрытых вершин **Open**. Установить ее глубину (0).

Шаг 2. Если список **Open** пуст, то окончание алгоритма и выдача сообщения о неудаче поиска, в противном случае перейти к следующему шагу.

Шаг 3. Выбрать первую вершину из списка **Open** (назовем ее **Current**) и перенести ее в список раскрытых вершин **Closed**.

Шаг 4. Если глубина вершины **Current** равна граничной глубине **D**, то перейти к шагу 2, в ином случае перейти к следующему шагу.

Шаг 5. Раскрыть вершину **Current**, построив все ее дочерние вершины. Если дочерних вершин нет, то перейти к шагу 2, иначе поместить все дочерние вершины (в произвольном порядке; с указанием их глубины) в начало списка **Open** и построить указатели, ведущие от этих вершин к родительской вершине **Current**.

Шаг 6. Если среди дочерних есть хотя бы одна целевая вершина, то окончание алгоритма и выдача решения задачи, получающегося просмотром указателей от найденной целевой вершины к начальной. В противном случае перейти к шагу 2.

Конец алгоритма.

Приведенное только что описание очень похоже на описание алгоритма поиска вглубь, разница заключается только в ограничении глубины (шаг 4) и в месте списка **Open**, куда помещаются построенные дочерние вершины (шаг 5).

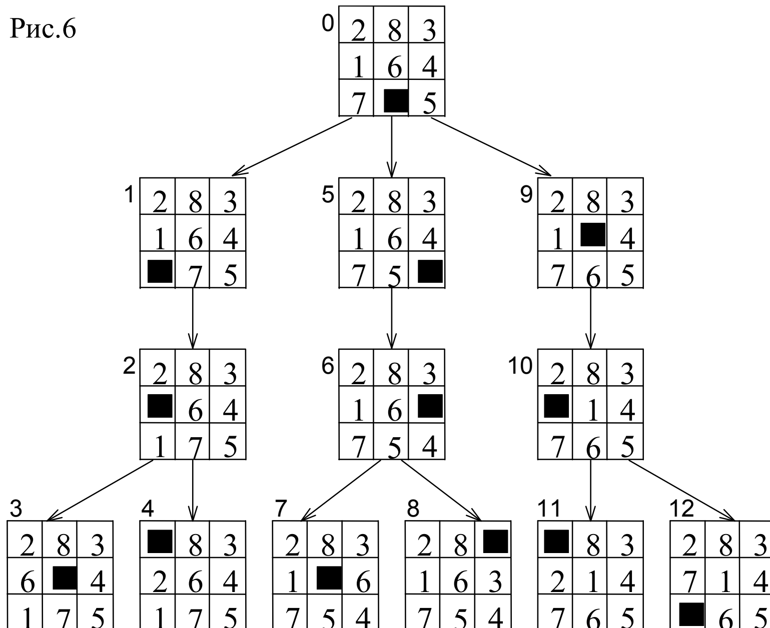
Поскольку глубина поиска ограничена, то будучи примененным к деревьям-пространствам состояний, описанный базовый алгоритм поиска вглубь всегда заканчивает работу. Но в отличие от алгоритма поиска вширь, он является неполным алгоритмом, поскольку вершины пространства

состояний, расположенные ниже граничной глубины, среди которых могут быть и целевые, так и останутся нерассмотренными.

На рис. 6 показано дерево перебора, построенное алгоритмом поиска вглубь; граничная глубина установлена равной 4. В качестве начального состояния взята та же самая, что и в примере на рис. 5, конфигурация игры в восемь. Вершины занумерованы в том порядке, в котором они были построены. В ходе поиска раскрыто 7 и построено 12 вершин, но, как нетрудно убедиться, сравнивая последние два рисунка, в целом это не те же самые 12 первых вершин, построенных алгоритмом поиска вширь.

Видно, что в алгоритме поиска в глубину сначала идет поиск вдоль одного пути, пока не будет достигнута установленная граничная глубина, затем рассматриваются альтернативные пути той же или меньшей глубины, которые отличаются от первого пути лишь последней (концевой) вершиной, после чего рассматриваются пути, отличающиеся последними двумя вершинами, и т.д.

Рис.6



Анализ слепых алгоритмов. Бэктрекинг

Если продолжить выполнение алгоритмов перебора вширь и вглубь для рассмотренного начального состояния игры в восемь (для задачи, указанной на рис.1(б)), то на глубине 5 будет найдена целевая конфигурация. При этом алгоритмом поиска вширь будет раскрыто **26** и построено **46** вершин, а алгоритмом поиска вглубь – соответственно **18** и **35** вершин.

Сравнивая в общем алгоритмы поиска вширь и вглубь, можно утверждать, что они примерно сравнимы по эффективности (количеству построенных вершин). Но в ряде случаев второй алгоритм, несмотря на свою неполноту, может оказаться предпочтительнее: если он начат с удачной стороны, то целевая вершина будет обнаружена раньше, чем в алгоритме поиска вширь.

Подчеркнем, что как и в случае перебора вширь, при переборе вглубь формируется именно дерево, а не граф перебора, даже если пространство состояний представлялось графом с циклами. В последнем случае, однако, дерево перебора может содержать дубликаты состояний. Нельзя, к примеру, исключить ситуацию, когда некие две вершины являются друг для друга дочерними, и тогда они будут многократно дублироваться в списке **`Open`**, приводя к зацикливанию алгоритма.

Чтобы избежать такого дублирования вершин, и предотвратить тем самым возможное зацикливание алгоритма в случае перебора на графах общего вида, необходимо внести некоторые очевидные изменения в описанные базовые алгоритмы поиска вширь и вглубь..

В алгоритме перебора вширь следует дополнительно проверять, не находится ли каждая вновь построенная дочерняя вершина (точнее, соответствующее описание состояния) в списках **`Open`** и **`Closed`** по той причине, что она уже строилась раньше в результате раскрытия какой-то другой вершины. Если это так, то такую вершину не надо снова помещать в список **`Open`** (таким образом разрывается цикл графа-пространства, и обрывается соответствующая ветвь дерева перебора). В алгоритме же ограниченного поиска вглубь кроме рассмотренного изменения может оказаться необходимым пересчет глубины порожденной дочерней вершины, уже имеющейся либо в списке **`Open`**, либо в списке **`Closed`**.

Внесенные изменения дают гарантию, что алгоритм поиска вширь всегда завершит работу в случае существования решения, а алгоритм поиска вглубь закончится в любом случае, независимо от существования решения.

Немаловажно, что алгоритмы слепого перебора описаны нами в форме, пригодной для их программирования с использованием любого языка, не только языка программирования задач искусственного интеллекта. Алгоритм поиска вглубь демонстрирует также способ решения поисковых задач, называемый ***бэктрекингом*** (*backtracking*), или ***режимом возвратов***. Этот способ предлагает определенную организацию перебора всех возможных вариантов решения задачи, число которых может быть велико.

Суть бэктрекинга состоит в том, чтобы в каждой точке процесса решения, где существует несколько равноправных (априори) альтернативных путей дальнейшего продолжения, выбрать один из них и следовать ему, предварительно запомнив другие альтернативные пути – для того, чтобы в случае неуспешности выбранного пути решения вернуться в указанную точку и выбрать для продолжения поиска следующий альтернативный вариант-путь. В общем случае в процессе решения возможно возникновение многих подобных точек выбора (называемых *развилками*) со своими вариантами продолжения решения, и к каждой из точек необходимо совершать возвраты и пробовать другие варианты.

В базовом алгоритме поиска вглубь по существу проводится бэктрекинг: действительно, запоминание всех альтернатив продолжения поиска (нераскрытых вершин) осуществляется в списке **`Open`**, на шаге 3 производится выбор варианта-альтернативы, а возврат к этому шагу для выбора следующей альтернативы осуществляется на шагах 4 и 5.

Некоторые языки для задач искусственного интеллекта, как, например, Пролог и Плэнер имеют специальный встроенный механизм для реализации бэктрекинга. Это означает, что запоминание *развилок* – самих альтернатив и связанной с ними информации, а также реализация *возвратов* к нужным точкам (с восстановлением всей операционной обстановки этой точки) возложены на интерпретатор языка, т.е. делается автоматически. От программиста требуется лишь определение *развилок* с нужными альтернативами и инициация в необходимый момент процесса возврата (заметим попутно, что язык Плэнер, в отличие от Пролога предлагает более гибкие средства управления бэктрекингом).

В целом алгоритмы слепого перебора являются неэффективными методами поиска решения, и в случае нетривиальных задач их невозможно использовать из-за большого числа порождаемых вершин. Действительно, если L – длина решающего пути, а B – средне количество ветвей (дочерних вершин) у каждой вершины, то для нахождения решения надо исследовать B^L путей, ведущих из начальной вершины. Величина эта растет экспоненциально с ростом длины решающего пути, что приводит к ситуации, называемой комбинаторным взрывом.

Таким образом, для повышения эффективности поиска необходимо использовать информацию, отражающую специфику решаемой задачи и позволяющую более целенаправленно двигаться к цели. Такая информация обычно называется *эвристической*, а соответствующие алгоритмы и методы – эвристическими.

Эвристические методы поиска

Идея, лежащая в основе большинства эвристических алгоритмов, состоит в том, чтобы оценивать с помощью эвристической информации перспективность нераскрытых вершин пространства состояний (с точки зрения достижения цели), и выбирать для продолжения поиска наиболее перспективную вершину. Самый обычный способ использования эвристической информации – введение так называемой ***эвристической оценочной функции***. Эта функция определяется на множестве вершин пространства состояний и принимает числовые значения. Значение эвристической оценочной функции Est(V) может интерпретироваться как перспективность раскрытия вершины (иногда – как вероятность ее расположения на решающем пути). Обычно считают, что меньшее значение Est(V) соответствует более перспективной вершине, и вершины раскрываются в порядке увеличения (точнее, неубывания) значения оценочной функции.

Алгоритм эвристического перебора

Последовательность шагов формулируемого ниже базового ***алгоритма эвристического (упорядоченного) перебора*** похожа на последовательность шагов алгоритмов слепого перебора, отличие заключается в использовании эвристической оценочной функции. После порождения нового состояния-вершины производится его оценивание (т.е. вычисление значения этой функции), и списки открытых и закрытых вершин должны содержать кроме самих вершин их оценки, которые и используются для упорядочения поиска.

Для раскрытия каждый раз в цикле выбирается наиболее перспективная концевая вершина дерева перебора. Также как и в случае алгоритмов слепого поиска множество порождаемых алгоритмом вершин и указателей образует дерево, в листьях которого находятся нераскрытые вершины.

Предполагаем, что исследуемое алгоритмом пространство состояний представляет собой дерево. Тогда основные шаги **алгоритма эвристического перебора** (*best_first_search*) таковы:

Шаг 1. Поместить начальную вершину в список нераскрытых вершин `Open` и вычислить ее оценку.

Шаг 2. Если список `Open` пуст, то окончание алгоритма и выдача сообщения о неудаче поиска, в противном случае перейти к шагу 3.

Шаг 3. Выбрать из списка `Open` вершину с минимальной оценкой (среди вершин с одинаковой минимальной оценкой выбирается любая); перенести эту вершину (назовем ее `Current)` в список `Closed`.

Шаг 4. Если `Current` – целевая вершина, то окончание алгоритма и выдача решения задачи, получающегося просмотром указателей от нее к начальной вершине, в противном случае перейти к следующему шагу.

Шаг 5. Раскрыть вершину `Current`, построив все ее дочерние вершины. Если таких вершин нет, то перейти к шагу 2, в ином случае – к шагу 6.

Шаг 6. Для каждой дочерней вершины вычислить оценку (значение оценочной функции), поместить все дочерние вершины в список `Open`, и построить указатели, ведущие от этих вершин к родительской вершине `Current`. Перейти к шагу 2.

Конец алгоритма.

Заметим, что поиск в глубину можно рассматривать как частный случай упорядоченного поиска с оценочной функцией Est(V) = d(V) , а поиск в ширину – с функцией Est(V) = 1/d(V) , где d(V) – глубина вершины V.

Чтобы модифицировать рассмотренный алгоритм для перебора на произвольных графах-пространствах состояний, необходимо предусмотреть в нем реакцию на случай построения дочерних вершин, которые уже имеются либо в списке раскрытых, либо в списке нераскрытых вершин.

В принципе эвристическая оценочная функция может зависеть не только от внутренних, собственных свойств самого оцениваемого состояния (т.е., свойств входящих в описание состояния элементов) но и от характеристик всего пространства состояний, например, от глубины местонахождения оцениваемой вершины в дереве перебора или других свойств пути к этой вершине. Поэтому значение оценочной функции для вновь построенной дочерней вершины, входящей в список `Open` или `Closed`, может понизиться, и надо скорректировать старую оценку вершины, заменив ее на новую, меньшую. Если вновь построенная вершина с меньшей оценкой входит в список `Closed`, необходимо вновь поместить ее в список `Open`, но с меньшей оценкой. Потребуется также изменить направления указателей от всех вершин списков `Open` и `Closed`, оценка которых уменьшилась, направив их к вершине `Current`.

Впрочем, если оценочная функция учитывает только внутренние характеристики вершин-состояний, то для предотвращения зацикливания требуется более простая модификация алгоритма – надо просто исключить дублирование состояний в списках `Open` и `Closed`, оставляя в них лишь по одному состоянию.

Проиллюстрируем работу алгоритма эвристического поиска опять же на примере игры в восемь для той же начальной ситуации. Воспользуемся в качестве оценочной следующей простой функцией:

$$Est1(V) = d(V) + k(V)$$, где

d(V) – глубина вершины V, или число ребер дерева на пути от этой вершины к начальной вершине;
k(V) – число фишек позиции-вершины V, стоящих не на «своем» месте (фишка стоит не на «своем» месте, если ее позиция отлична от позиции в целевом состоянии).

На рис.7 показано дерево, построенное алгоритмом эвристического перебора с указанной оценочной функцией. Оценка каждой вершины приведена рядом с ней внутри кружка. Отдельно стоящие цифры, как и раньше, показывают порядок, в котором строились вершины. Двойной рамкой обведена найденная целевая вершина, она построена двенадцатой.

Видно, что поскольку каждый раз выбор вершины с минимальной оценкой производится внутри всего построенного к текущему моменту дерева перебора, то раскрываемые друг за другом вершины могут располагаться в отдаленных друг от друга частях дерева. Применяемая оценочная функция такова, что при прочих равных преимущество имеет менее глубокая вершина.

Решение задачи длиною в пять ходов найдено в результате раскрытия 6 и построения 13 вершин – это существенно меньше, чем при использовании слепого перебора (соответствующие числа были: 26 и 46, 18 и 35). Таким образом, использование эвристической информации приводит к существенному сокращению перебора.

Существует несколько критериев оценки качества работы алгоритмов перебора. Один из них называется *целенаправленностью* и вычисляется как P = L / N , где

L – длина найденного пути до цели (она равна глубине целевой вершины), а

N – общее число вершин, построенных в ходе перебора.

P = 1, если строятся только вершины решающего пути, в остальных случаях P < 1, вообще, эта величина тем меньше, чем больше строится бесполезных вершин. Таким образом, этот критерий показывает, насколько дерево, построенное при переборе, вытянуто, а не кустисто. К сожалению, величина P зависит от длины решающего пути, что затрудняет порой сравнение алгоритмов. Другой критерий оценки, *фактор эффективного ветвления*, зависит от длины решающего пути гораздо меньше.

Ясно, что алгоритм эвристического поиска с хорошо подобранной оценочной функцией обнаруживает решение задачи быстрее алгоритмов слепого перебора. Однако подбор удачной эвристической функции, существенно сокращающей поиск, – наиболее трудный момент при формализации задачи, и часто подходящая оценочная функция выявляется в результате многих экспериментов.

Принято сравнивать различные оценочные функции для одной и той же задачи по их *эвристической силе*, т.е. по тому, насколько они убыстряют поиск, делают его эффективным. Заметим, что эвристическая сила функции должна учитывать общий объем вычислительных затрат при поиске, поэтому кроме числа раскрытых и построенных вершин важен и такой фактор, как сложность вычисления самой оценочной функции.

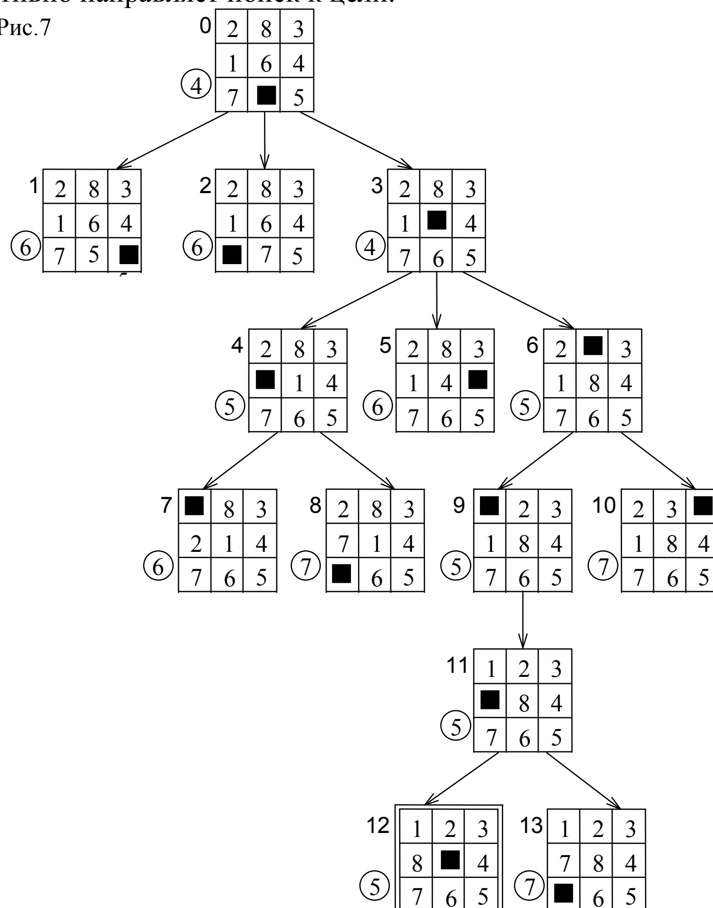
Для игры в восемь можно предложить еще одну эвристическую функцию:

Est2(V) = d(V) + s(V).

Первое слагаемое d(V) этой функции имеет тот же смысл, что и для функции Est1. Второе слагаемое получается, если для каждой из восьми фишек подсчитать сумму двух расстояний – по вертикали и горизонтали – между клетками, где находится эта фишка в оцениваемом и целевом состояниях, а затем подсчитать общую сумму s(V) таких расстояний для всех восьми фишек (тем самым получим «суммарное расстояние» всех фишек от их целевого положения).

Например, для начальной конфигурации на рис.7 расстояние текущего положения фишки с номером 8 от ее положения в целевой конфигурации равно 1 и по вертикали, и по горизонтали, а сумма их равна 2. Общая же сумма таких расстояний для всех фишек равна 5 (фишки 3, 4, 5, 7 стоят уже на «своем» месте, поэтому их вклад в суммарное расстояние равен 0). Интуитивно ясно, и это можно показать на примерах, что новая эвристическая функция имеет большую эвристическую силу, т.е. более эффективно направляет поиск к цели.

Рис.7



Допустимость алгоритма эвристического перебора

Важным является вопрос, может ли алгоритм эвристического перебора с оценочной функций общего вида (т.е. выбираемой произвольно) гарантировать нахождение решающего пути за конечное число шагов в тех случаях, когда решение существует (как алгоритм поиска вширь). Понятно, что такой уверенности нет прежде всего для задач с бесконечными пространствами состояний. Вообще же, нередка ситуация, когда эвристика, сильно сокращающая перебор для большинства начальных состояний, в то же время для других начальных конфигураций либо не может уменьшить необходимую переборную работу (и решение задачи может искаться даже дольше, чем с использованием слепого метода), либо вовсе не может обеспечить обнаружение решающего пути.

Математическое исследование алгоритма эвристического поиска – условий, гарантирующих нахождение им решения – было проведено для эвристических оценочных функций специального вида и для более сложной задачи, чем до сих пор рассматриваемая задача поиска любого решающего пути до целевой вершины.

Предположим, что на множестве дуг пространства состояний определена функция стоимости:

$c(V_A, V_B)$ – стоимость дуги-перехода от вершины V_A к вершине V_B .

Определим также *стоимость* любого пути в графе-пространстве как сумму стоимостей входящих в путь дуг. Пусть целью поиска будет не просто нахождение решающего пути, а нахождение *оптимального решающего пути* – решающего пути с минимальной стоимостью.

Предположим также, что эвристическая оценочная функция Est(V) построена таким образом, чтобы оценивать стоимость оптимального решающего пути, идущего из начальной вершины к одной из целевой вершин, при условии, что этот путь проходит через вершину V. Тогда значение оценочной функции можно представить в виде суммы двух слагаемых:

$$Est(V) = g(V) + h(V) \qquad (*)$$

где g(V) – оценка оптимального пути от начальной вершины до вершины V,

а h(V) – оценка оптимального пути от вершины V до целевой вершины.

Если в процессе поиска уже построена вершина V, то путь до нее найден, и его стоимость может быть вычислена. Найденный путь не обязательно оптимален (возможно, существует более дешевый, еще не найденный путь из начальной вершины в V), однако стоимость найденного пути может быть использована в качестве оценки искомого пути минимальной стоимости из начальной вершины до V, т.е. в качестве первого слагаемого g(V) эвристической функции. Второе же слагаемое h(V) может быть предложено исходя из эвристических соображений, свойственных конкретной решаемой задаче, как некоторая характеристика-оценка текущей вершины V (близости ее к цели). Таким образом, собственно эвристическая информация будет воплощена только во втором слагаемом оценочной функции.

Разновидность алгоритма эвристического поиска, применяемого для поиска оптимального решающего пути и использующего при этом оценочную функцию указанного выше вида (*), известен в литературе как *А-алгоритм*. Были доказаны важные свойства этого алгоритма, прежде всего, утверждение о его допустимости.

Алгоритм перебора называют *допустимым* (или *состоятельным*), если для произвольного графа он всегда заканчивает свою работу построением оптимального пути к цели, при условии, что такой путь существует.

Пусть h*(V) – стоимость оптимального пути из произвольной вершины V в целевую вершину.

Верна следующая **теорема о допустимости А-алгоритма**:

А-алгоритм, использующий некоторую эвристическую функцию вида (*), где

g(V) – стоимость пути от начальной вершины до вершины V в дереве перебора, а

h(V) – эвристическая оценка оптимального пути из вершины V в целевую вершину,

является допустимым, если $h(V) \le h^*(V)$ для всех вершин V пространства состояний.

А-алгоритм эвристического поиска, применяющий функцию h(V), удовлетворяющую этому условию, получил название ***А*-алгоритма***.

Практическое значение этой теоремы в том, что для допустимости А-алгоритма достаточно найти какую-либо нижнюю грань функции h*(V) и использовать ее в качестве h(V) – тогда оптимальность найденного алгоритмом решения будет гарантирована.

Если взять тривиальную нижнюю грань, т.е. установить h(V) = 0 для всех вершин пространства состояний, то допустимость будет обеспечена. Однако этот случай соответствует полному отсутствию какой-нибудь эвристической информации о задаче, и оценочная функция Est не имеет никакой эвристической силы, т.е. не сокращает возникающий перебор. А*-алгоритм ведет себя при этом аналогично поиску вширь.

Точнее, при Est(V) = g(V) (где g(V) – стоимость пути от начальной вершины до вершины V), мы получаем алгоритм, известный как ***алгоритм равных цен (или Алгоритм Дейкстры)***. Алгоритм равных цен представляет собой более общий вариант метода перебора в ширину, при котором вершины

раскрываются в порядке возрастания стоимости g(V) , т.е. в первую очередь раскрывается вершина из списка нераскрытых вершин, для которой величина g имеет наименьшее значение.

Если же, кроме того, положить стоимость $c(V_A, V_B) = 0$ для всех дуг пространства состояний, то A*-алгоритм просто превращается в неэффективный слепой поиск вширь.

Обе предложенные для игры в восемь эвристические функции Est1(V) и Est2(V) удовлетворяют условию допустимости A*-алгоритма. Первое их слагаемое d(V) есть стоимость пути к вершине V при стоимости всех дуг $c(V_A, V_B) = 1$. Функции отличаются лишь вторым слагаемым, и можно показать, что значение второй функции всегда (т.е. для всех состояний), больше значения первой функции: Est1(V) ≤ Est2(V) , что равнозначно k (V) ≤ s (V) .

Действительно, во второй функции вклад каждой фишки в общую оценку-сумму s(V) либо равен 0 (фишка стоит уже на «своем» месте), либо не меньше 1 (в противном случае), в первой же функции этот вклад в k(V) соответственно либо равен 0, либо равен 1.

Из последнего неравенства следует, что условие допустимости достаточно доказать только для второй функции Est2. Справедливость нужного условия
s(V) ≤ h*(V) следует из следующего соображения. Если бы фишки не мешали друг другу и могли двигаться до «своего» места по кратчайшему пути, как если бы других фишек на квадрате не было, то сумма длин таких путей для всех фишек была бы в точности равна значению s(V) . На самом же деле фишки редко когда могут двигаться по кратчайшей траектории из-за того, что на ней расположены другие фишки, поэтому длина (стоимость) оптимального решения h*(V) будет не меньше s(V).

Заметим, что s(V) не учитывает должным образом трудность обмена местами двух соседних фишек, а поэтому ее эвристическая сила в принципе может быть повышена. В ряде случаев эвристическая сила некоторой оценочной функции может быть повышена просто путем умножения на положительную константу, большую единицы, однако часто такое повышение осуществимо только за счет отказа от допустимости алгоритма. Например, если для игры в восемь в качестве второй составляющей эвристической функции взять h(V) = 2•s(V), то в ряде случаев такая функция будет убыстрять поиск и позволит решать более трудные задачи, но условие допустимости перестанет выполняться (так как для начального состояния на рис.15: h*(V) ≤ 2•s(V)).

Вообще в случае, когда верно неравенство $h_1(V) \le h_2(V)$ для всех вершин пространства состояний, не являющихся целевыми, A*-алгоритм, использующий эвристическую составляющую $h_2(V)$, называется *более информированным*, чем A*-алгоритм с функцией $h_1(V)$. Показано, что если эти функции статичны (т.е. не изменяются в процессе поиска), то более информированный алгоритм раскрывает всегда меньшее число вершин, прежде чем находит путь минимальной стоимости. Это значит, что более информированный алгоритм осуществляет более направленный, а значит, более эффективный (при прочих равных) поиск целевой вершины. Таким образом, понятие информированности отражает один из аспектов понятия эвристической силы оценочной функции при поиске в пространстве состояний.

Итак, желательно подобрать такую эвристическую функцию h(V), которая была бы нижней границей h*(V) (чтобы гарантировать допустимость алгоритма) и была бы как можно ближе к h*(V) (чтобы обеспечить эффективность поиска). К сожалению, существуют задачи, для которых нельзя найти оценочную функцию, обеспечивающую во всех случаях как эффективность, так и допустимость эвристического поиска. Поэтому часто приходится останавливаться на эвристических функциях, сокращающих поиск во многих случаях ценой отказа от гарантии найти оптимальный решающий путь.

Заметим, что в идеальном случае, когда известна оценка h*(V), и она используется в качестве h(V), A*-алгоритм находит оптимальный решающий путь сразу, без раскрытия ненужных вершин.

Упрощенные варианты эвристического перебора

Сильным упрощением базового алгоритма эвристического поиска с произвольной оценочной функцией является алгоритм ***«подъема на холм»***. Этот алгоритм при каждом раскрытии вершины производит упорядочение (по значению оценочной функции) только порожденных дочерних вершин, и выбирает для последующего раскрытия дочернюю вершину с наименьшей оценкой (а не вершину с наименьшей оценкой среди всех нераскрытых вершин дерева поиска, как в базовом алгоритме). Очевидно, что такой локальный выбор среди только что построенных дочерних вершин реализовать гораздо проще, чем глобальный выбор вершины во всем дереве перебора.

Идея этого алгоритма аналогична идее известного вне области искусственного интеллекта метода «подъема на гору», применяемого для поиска максимума (или минимума) функции. Для того, чтобы в конечном счете найти максимум функции, на каждом шаге метода производится движение в направлении наибольшей крутизны функции. Для определенного класса функций (имеющих единственный максимум и некоторые другие свойства роста) такое использование локальной

информации, т.е. знания направления наиболее крутого подъема в текущей точке, позволяет найти глобальное решение, т.е. максимум функции.

В алгоритме «подъема на холм» в пространстве состояний роль функции метода «подъема на гору» играет эвристическая оценочная функция, взятая с обратным знаком. Поиск продолжается всегда от той дочерней вершины, которая имеет меньшее значение эвристической функции (при этом случай, когда вершин с одинаковой минимальной оценкой несколько, является нежелательным).

Алгоритм «подъема на холм» дает тот же результат, что и базовый алгоритм эвристического поиска в тех случаях, когда оценочная функция обладает определенными свойствами, в частности, имеет один (глобальный) экстремум. Алгоритм становится несостоятельным, если у эвристической функции имеется несколько локальных экстремумов. Бывают и другие случаи бесперспективности «подъема на холм»: если поверхность-множество значений функции имеет равнинный участок («плато») или же участки узкого и длинного возвышения (в виде горного «хребта»), и процесс поиска вывел как раз на них. Таким образом, этот алгоритм имеет ограниченную применимость, но иногда возникающие проблемы можно разрешить, построив более подходящую эвристическую функцию.