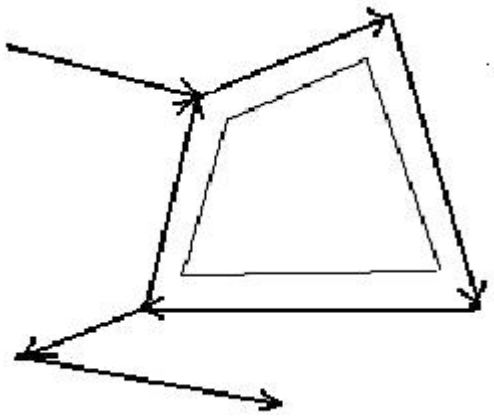
Экономный обход графа

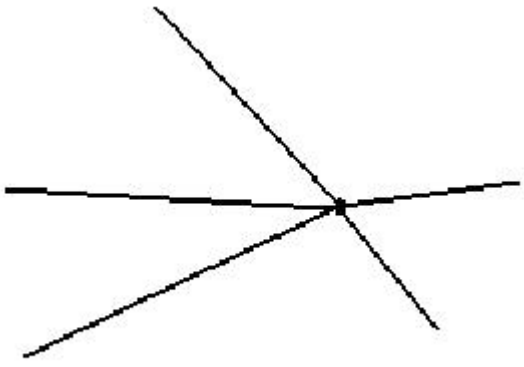
Условие задачи

Дан ориентированный граф. Необходимо совершить полный его обход, с использованием минимальных дополнительных данных.

Задача полного обхода графа достаточно понятна. От уже рассмотренных задач на обход деревьев она отличается тем, что в произвольном графе возможны циклы, что делает невозможным движение только вперед.



На рисунке показан граф с одним циклом. Жирные линии изображают ребра графа, а более тонкие его цикл. Двигаясь только вдоль стрелок, легко начать бесконечное движение вдоль единственного цикла. Это впрочем чисто техническая проблема. Достаточно, придя в очередной узел, выяснить, выходит ли из него хотя бы один неиспользованный путь. Если да, то идти по нему, а если нет, то вернуться назад. Однако с необходимостью возврата снова возникает проблема. Посмотрим на следующий рисунок.



Пусть ребра слева — это входящие, а ребра справа — исходящие. Тогда для узла есть три пути, по которым в него можно попасть, и если вдруг случится вернуться в этот узел, то встанет вопрос — а по какому ребру возвращаться?

Очевидно, необходимо выполнять возврат по тому же ребру, по которому пришли в узел последний раз. Возникшую проблему легко решить, если создать специальную структуру данных (для каждого узла), в которой каждый раз при входе в узел сохранялась бы информация об узле, из которого пришли в данный. Ясно, что это должен быть массив,

ведь если в узел входит десять ребер, то вход будет осуществлен десять раз, все десять входов из разных узлов и все они должны быть запомнены. Таким образом, приходим к необходимости создать для каждого узла дополнительную структуру данных, которая будет весить почти столько же, сколько и основная информация о связях узла. Плохо ли это? И если плохо, то насколько? Чтобы оценить наши потери рассмотрим следующую достаточно обычную задачу.

Дана сеть населенных пунктов связанных между собой различными дорогами (сухопутными, воздушными, водными). Необходимо оценить стоимость передвижения из пункта ***A*** в пункт ***B***. Пусть всего населенных пунктов 100 и каждый соединен с 10 другими. Это не слишком много. Для организации связей пронумеруем их в произвольном порядке, а связь между пунктом и соседями пометим однобайтными номерами соседей. Арифметика говорит,, что дополнительная информация для всего графа потребует тысячи байт. Вроде бы немного, но для тысячи пунктов это около 20 Кбайт (т. к. однобайтными номерами уже не обойтись). Объем дополнительной структуры данных стремительно растет!

Можно конечно пока не волноваться, в абсолютных величинах объем дополнительной информации не велик. Но абсолютные величины здесь не самое главное. Главный фактор — это равнозначность объема основной информации и дополнительной. Если появится задача, в которой объем информации о связях будет исчисляться десятками мегабайт, а в этом нет ничего невероятного, то она станет почти не решаемой, так как дополнительная информация также потребует десятки мегабайт. Оперативной памяти не хватит, а использование файлов резко затормозит работу программы. Если вы не верите, что могут быть такие задачи, то вот пример.

Известно, что каждая книга, несущая знания (научная, техническая и т. д.), содержит ссылки на первоисточники информации и таких ссылок может быть сотни. Предположим, что была создана электронная библиотека, в которой количество книг исчисляется десятками миллионов наименований (в этом нет ничего невероятного, человечество накопило огромные знания), и создатели такой библиотеки решили предоставить своим пользователям возможность ссылочного поиска. Как только такая задача будет поставлена, тут же и встанет озвученная проблема.

Задач такого рода можно придумать достаточно много. Впрочем, проблема есть и для менее масштабных задач. Предположим, что та задача, которую предстоит решать именно вам, не требует десятков мегабайт. Это не означает, что вы не столкнетесь с нехваткой памяти. Возможно, придется считаться с необходимостью делиться ресурсами с другими задачами, работающими в данной вычислительной системе. В общем, сейчас, наверное, уже понятно, почему в условии поставлено требование о минимальных дополнительных структурах данных.

Заметим, что рассуждения, приведенные ранее, достаточно основательны. Необходимость дополнительного массива адресов ссылок почти логически выведена, и, наверное, сейчас трудно представить, что без этого массива можно обойтись, но все же попробуем.

Замечание

Наша методологическая цель — минимизировать интуицию и максимизировать логику. Конечно, полностью логически вывести решение из условия невозможно, но рассчитывать на творческий прорыв, в борьбе с каждой технической деталью вряд ли разумно. Поэтому мы все же стремимся искать логические основания, исходя из которых можно рассуждать в направлении решения. Кроме того, интуитивный прорыв всегда субъективен, он опирается на наши личные знания и способности, которым не всегда можно доверять. *Логика всегда объективна, логика не зависит от личных качеств.*

Последнее предложение специально выделено. Оно показывает место, в котором наше мышление легко попадает в ловушку. На самом деле логика тоже субъективна и тоже опирается на личные знания и убеждения в той же мере, в какой и на объективные факты. Можно смело утверждать, что любой логический вывод — это в какой-то мере вывод из фактов, а в какой-то мере вывод из наших субъективных знаний.

Именно в такую ловушку мы и попали, выведя необходимость дополнительного массива. На самом деле есть железная необходимость запоминать адрес узла источника (далее так будем называть узел, из которого пришли в текущий), а необходимость использовать для этого массив не следует ниоткуда, кроме как из привычки использовать массивы для подобных целей. Логическая ошибка в рассуждениях обнаружена, можно двигаться далее.

Наша цель придумать, где хранить дополнительные данные. Если не придумывать новых структур для каждого узла, то можно подумать о дополнительной структуре для всего графа, так сказать дополнительный граф. Однако сама идея дополнительного графа не обещает возможности сокращения потребности в памяти, скорее всего новый граф должен будет в какой-то степени повторять структуру уже существующего, и весьма вероятно мы получим не сокращение, а увеличение потребности в памяти.

Если же не создавать ничего нового, а попробовать обойтись тем, что есть, то единственной возможностью для хранения дополнительных данных остается структура, используемая для хранения основных данных, речь идет о структуре, содержащей адреса связей.

Естественно, что использовать для дополнительных данных можно только то, что уже не будет применяться для хранения основных. Это хорошая зацепка. Действительно, путь вглубь по графу не повторяется. Иначе говоря, если некоторый узел ***C***, был источником для некоторого другого узла ***D***, то как бы не осуществлялось дальнейшее движение, узел ***C*** уже никогда не будет источником для ***D***. Отсюда следует, что данное предназначенное для хранения связи $\boldsymbol{C \rightarrow D}$ используется только один раз.

После сказанного идея становится совершенно прозрачной. Для хранения адресов возврата можно использовать адреса ссылок. А именно, в каждую ссылку после ее использования возможно записать адрес узла источника.

Остальное дело техники, но техники достаточно сложной, поэтому обсудим подробности, запишем алгоритм и снабдим программу необходимым количеством ремарок.

Главная техническая деталь — это конечно структура данных. Естественная структура для построения большого графа — это дерево, созданное с помощью связных списков, но мы воспользуемся обычными массивами, чтобы не загружать программу проблемами создания структуры данных, в программе и так достаточно сложная логика. Напомним, наш граф может иметь циклы, поэтому уже отработанные ранее процедуры создания деревьев здесь работать не будут. Если появится интерес, вы можете попытаться реализовать разработанную логику на динамической структуре данных.

Мы же договоримся о структуре данных состоящей из следующих полей:

- числовое поле, являющееся содержательным значением узла графа. (содержательные данные, привязанные к узлу, могут быть сколь угодно сложными, но для демонстрации логики достаточно самого простого поля — числа);

- числовой массив, содержащий номера узлов, с которыми связан данный.

Организовать такой граф несложно. Достаточно ввести количество узлов, затем внутри цикла по параметру для каждого узла ввести содержательное значение и номера узлов с ним связанных. Естественно для этого номера узлов графа должны быть каким-то способом (совершенно любым) пронумерованы.

Техническая проблема, связанная с сохранением дополнительной информации в массиве ссылок выражается двумя вопросами.

- На основании чего принимается решение о возврате в узел-источник?

- Как узнать, в каком элементе массива ссылок хранится нужный адрес возврата?

Не думайте, что ответ на первый вопрос уже есть. Да, понятно, что возврат осуществляется тогда, когда пути вперед уже нет. Но это смысловой ответ. Мы же сейчас решаем проблемы технического характера и, следовательно, должны все ответы давать в терминах используемых структур данных.

Чтобы ответить на поставленные вопросы, необходимо договориться о каких-то правилах обхода графа. Выбор этих правил — дело вкуса, но они должны быть. Пусть наше правило будет таким: придя в узел, выбираем ссылку, стоящую в массиве ссылок следующей за той, которой воспользовались в предыдущее посещение узла.

Для того чтобы учесть, какое ребро будет следующим, введем в описание узла еще одну дополнительную переменную — счетчик, роль которого заключается в подсчете количества вхождений в данный узел. Далее все очень просто. Значение счетчика в момент вхождения в узел — это количество предыдущих вхождений. Увеличивая счетчик при каждом вхождении на 1, мы обеспечиваем соответствие количества вхождений значению счетчика. И теперь значение счетчика можно использовать как номер очередной ссылки, по которой возможно идти вглубь графа.

Счетчик позволяет достаточно легко сохранять информацию и о том, где находится нужная ссылка на узел возврата. Ссылки возврата отсчитываются в обратном порядке по отношению к ссылкам движения в глубину. Если ссылки для движения в глубину отсчитываются от 1 в сторону увеличения, то ссылки возврата считаются от последней в сторону уменьшения. Еще один важный нюанс в возвратном движении. В программе после использования ссылки для возврата элемент массива, содержащий ссылку, обнуляется. Это сделано для того, чтобы выделить ребро графа, по которому уже было выполнено движение в обе стороны и, следовательно, его необходимо полностью исключить из использования.

На второй вопрос мы уже ответили. Ответ на первый вопрос будет таким — найдем неиспользованную ссылку (в программе поиск осуществляется с конца), и если номер ссылки меньше количества вхождений в данный узел, то выполняем движение в глубину, иначе выполняем возврат.

Полностью процесс движения по графу завершаем тогда, когда уже нет неиспользованных ссылок в исходном узле. Каждая ссылка была использована как для движения в глубину, так и для возврата.

Алгоритм обхода графа:

```
Индекс текущего узла = 1
Пока есть хотя бы одна неиспользованная ссылка, делать
  Найти в текущем узле последнюю неиспользованную ссылку
  Если ее номер больше количества вхождений в узел,
  То движение вперед
    Увеличить значение счетчика вхождений для текущего узла.
    Запомнить узел-источник.
    Перейти по ссылке
    Записать адрес источника вместо использованной ссылки.
  Иначе возврат
    Уменьшить значение счетчика вхождений.
```

Определить ссылку возврата и запомнить в промежуточную Переменную.
Обнулить ссылку.
Вернуться в узел источник.

Текст программы приведен в листинге.

Листинг

```
program example;
 uses crt;
 type
 rec=record
      count:byte;
      num:integer;
      uk:array[1..255]  of integer;
     end;
 var
  uzel:array[1..100] of rec;
  pred_index,tek_index,i,j,n,m,c:integer;
  q:boolean;
procedure print;
begin
  if uzel[tek_index].num>0 then
    begin
     write(uzel[tek_index].num,'  ');
     uzel[tek_index].num:=0;
     readkey;
    end;
end;
begin
 {создание сети}
 clrscr;
 write('Введите количество узлов сети -');read(n);
 for i:=1 to n do
   begin
    write('Узел номер -',i);
    write('   Введите значение узла -');read(uzel[i].num);
    {Инициализация массива ссылок}
    for j:=1 to 255 do uzel[i].uk[j]:=0;
    uzel[i].count:=0;
    write('Введите количество ссылок -');read(m);
    for j:=1 to m do
      begin
       write('ссылка номер ',j,'=');
       read(uzel[i].uk[j]);
      end;
   end;
 {прохождение сети}
 tek_index:=1;pred_index:=1;
 repeat
  {Поиск последней ссылки}
  m:=1;
  while (uzel[tek_index].uk[m]<>0)and(m<255) do m:=m+1;
  {Цикл проскакивает значимую ссылку, поэтому надо вернутся назад на шаг}
  if m=255 then m:=0 else m:=m-1;
 if (uzel[tek_index].count<m) then {Движение вперед}
  begin
  print;
   {Расчет индекса массива для сохранения адреса узла источника}
   m:=m-uzel[tek_index].count;
   if tek_index>1 then uzel[tek_index].count:=uzel[tek_index].count+1;
```

```
   c:=tek_index;
   tek_index:=uzel[tek_index].uk[m];
   {Сохранение адреса узла источника}
   if c>1 then uzel[c].uk[m]:=pred_index
    else uzel[c].uk[m]:=0;
   pred_index:=c;
  end
 else {отход назад}
  begin
  print;
   if uzel[tek_index].count>0
   then
    begin
     m:=uzel[tek_index].count;
     uzel[tek_index].count:=uzel[tek_index].count-1;
     c:=uzel[tek_index].uk[m];
     uzel[tek_index].uk[m]:=0;
     tek_index:=c;
    end
   else tek_index:=pred_index;
  end;
 if tek_index=1 then
 begin
 q:=true;
 {Проверка, есть ли еще неиспользованные ссылки}
  for j:=1 to 255 do
   if uzel[1].uk[j]<>0 then q:=false;
 end;
 until q; {Завершение процесса}
end.
```

В заключение.

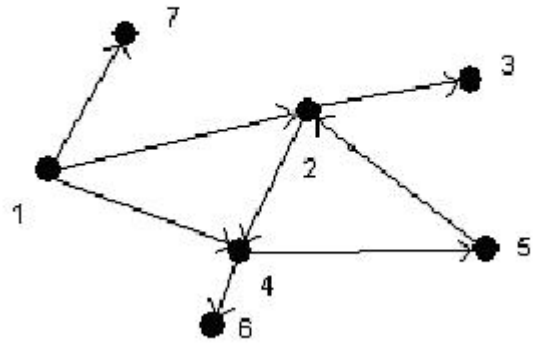
Мы получили работающую программу. Но это еще не все. Если вы тщательно ее протестируете, то вполне возможно сможете найти неработающий пример. Но с другой стороны работающих примеров можно привести сколько угодно много. Что это означает? Можно ли говорить, что программа ошибочна? Или возможно сделать другой вывод?

Все зависит от установки на то, что требовалось. Можно сказать, что алгоритм и соответственно программа ошибочны, а можно сказать, что программа верна, но существуют определенные ограничения. С такой ситуацией мы уже встречались в задаче о раскладке большой кучи камней на две. Давайте и данную ситуацию воспринимать так же. Все хорошо, но есть ограничения, которые надо четко определить.

Формулировка ограничений не займет у нас много сил. Обратите внимание на ключевой пункт нашей идеи. Это запись ссылок возврата вместо ссылок на следующие узлы. Отсюда следует и ограничение. Сформулируем его для графа: алгоритм будет работоспособным только в том случае, когда для любого его узла количество входящих ребер не меньше чем количество исходящих.

Технически задача достаточно сложная. При поиске решения использовались две различные терминологии. Мы рассуждали в терминах графов и при этом говорили о входящих и исходящих ребрах, рассуждали в терминах ссылок движения в глубину и ссылок возврата. Возможно, это не для всех оказалось удобно и может быть вы запутались в терминологии и неправильно соотнесли понятия ребер и ссылок. Если это случилось, то придется тщательно поработать с программой. А в помощь попробуйте проанализировать два примера. Слева приведен работающий пример. Пример справа немного отличается от примера слева, но он не работает корректно. Анализ этих примеров поможет вам более точно понять работу программы. Заметим, что любые деревья у нашей программы проблем не вызывают.

Работающий пример



Не работающий пример