第 7 章　基于 AI 的问题求解

本章讲述一个有趣的程序设计主题：人工智能(Artificial Intelligence，AI)。本书的目的之一是显示 C++的适用范围以及它所具有的多种功能。或许一个需要人工智能领域的应用程序最能说明这个问题。

人工智能的领域由几个引人注目的部分组成，但是许多基于 AI 的应用程序的基础是问题求解。本质上，存在两类问题。第一种类型的问题可以用某种确定性的、一定能够成功的程序来解决，如计算某个角度的正弦或者某个值的平方根。这种类型的问题很容易转换为计算机可以处理的算术。然而，实际上很少有问题会这么直接。相反，许多类型的问题只有通过寻找一个解决方案来解决。AI 所关心的正是这类问题的解决方案。这也是本章讲述的搜索类型。

为了理解搜索对 AI 如此重要的原因，请考虑下面的问题。AI 搜索早期的目的之一是创建一个通用问题求解程序。通用问题求解程序是一个程序，这个程序可以给出各种不同问题的解决方案，而不需要有特定的、设计好的知识。可以理解，这种程序非常令人满意。遗憾的是，这种通用问题求解程序非常难于实现，它是可望而不可及的。一个麻烦是许多实际问题具有不同的尺寸和复杂度。因为通用问题求解程序必须在一个非常大的、具有复杂可能性的世界中搜索一个解决方案，所以首先必须找到某种在这种环境下搜索的优先方法。在本章我们并没有雄心勃勃地试图开发一个通用问题求解程序，我们将探索基于 AI 的搜索技术，这种技术可以应用于许多问题。

C++是非常适合于 AI 开发人员的语言。原因是 C++提供了对基于 AI 的应用程序通常使用到的程序元素的支持：递归、链表和堆栈。正如您所知道的那样，C++可以方便而高效地处理递归。加上各种 STL 容器所提供的功能，您具有一个有效的 AI 开发环境。

7.1　表示法和术语

假定您丢失了汽车钥匙。您知道它在房间的某个位置，如图 7-1 所示。

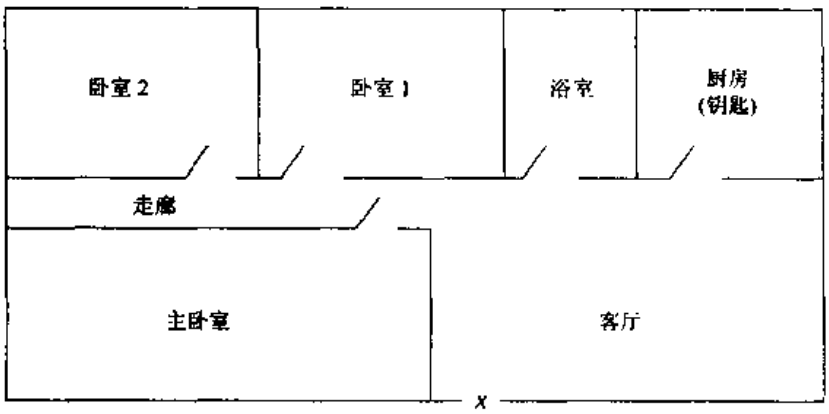

图 7-1　钥匙的位置

您正站在前门(X 所指的位置)，当您开始查找时，先检查了客厅。然后走过走廊，到第一间卧室，通过走廊到第二间卧室，再返回走廊，然后去主卧室。仍然没有找到钥匙，您又返回到客厅。最后，您发现钥匙在厨房。这种情况很容易用图表来表示，如图 7-2 用图表的形式来表示这个搜索问题是有帮助的，因为它提供了一种方便的方法来描述找到解决方案的方式。

记住前面的讨论，考虑表 7-1 所示的术语，这些术语将在本章使用：

表 7-1 搜索算法的相关术语

术 语	说 明
节点	一个不连续的点
终端节点	结束路径的节点
搜索空间	所有节点的集合
目标	代表搜索目标的节点
试探法	能够决定某个节点是否比另一个节点更好的信息
解决路径	通向目标的路径上的节点的有向图

在丢失钥匙的示例中，住宅中的每个房间都是一个节点；整个住宅是搜索空间；目标是厨房；解决路径如图 7-2 所示。卧室、厨房以及浴室都是终端节点，因为它们不通向任何地方。图中没有表示试探法，它们是您可能用来更好地选择路径的技术。

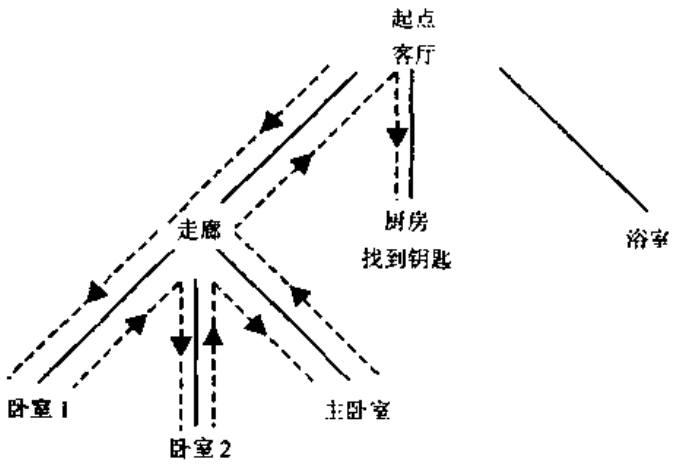

图 7-2 寻找丢失钥匙的解决路径

7.2 组合爆炸

通过前面的示例，您可能会认为解决方案的查找很容易——从起点开始，逐一搜索获取结果。在特别简单的丢失钥匙的示例中，这是一种不错的方法，因为搜索空间非常小。但是对于许多问题(特别是您需要使用计算机解决的那些问题)，搜索空间中节点的数目非常多，随着搜索空间的增加，通向目标的可能路径也在增加。问题在于，向搜索空间中加入一个节点时，往往加入了不止一条路径。也就是说，潜在的通向目标的路径数量随着搜索空间尺寸的增长以非线性的方式在增长。在非线性的情况下，可能的路径数量会迅速地变得很大。

例如，考虑在表格中排列下面 3 个对象——A、B 和 C——的方法的数量。有 6 种可能的排列方式如下所示。

A	B	C
A	C	B
B	C	A
B	A	C
C	B	A
C	A	B

可以迅速证明，这 6 种方式是 A、B 和 C 全部的排列方式。您可以使用组合数学——研究事物组合方法的学科——的定理得到相同的结果。根据这个定理，排列 N 个对象的方式的数量为 N!(N 的阶乘)。数字的阶乘等于这个数字和小于这个数字直到 1 的所有数字的乘积。即 3!是 3×2×1，等于 6。如果您排列 4 个对象，那么排列方式的数量就是 4!，即 24。如果有 5 个对象，那么这个数量为 120，6 个对象的数量为 720。1000 个对象的可能的排列方式的数量是巨大的。图 7-3 中的图表给出了被称为组合爆炸的直观的概念。一旦可能性多了起来，验证(甚至是穷举)所有的排列迅速地变得困难起来。

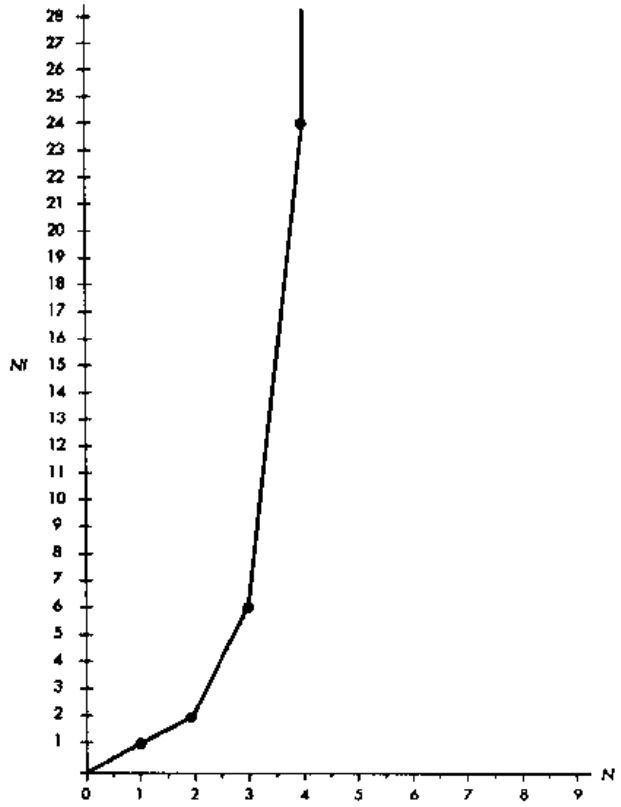

图 7-3　关于阶乘的组合激增

搜索空间的路径也存在类似的组合爆炸。因此，只有非常简单的问题可以用穷举搜索。穷举搜索是检查所有节点的搜索方法。因此，这是一种“暴力”的技术。暴力总是有效的，但是对于比较大的问题，它往往是不实际的，因为它消耗太多的时间，或者太多的计算机资源，或者二者兼有。为此提出了基于 AI 的搜索技术。

7.3 搜索方法

有多种方式搜索一个解决方案。最基本的4种方式为：

- 深度优先
- 广度优先
- 爬山法
- 最低成本法

本章讲述了每种搜索方法。

搜索方法性能评估

评估一个基于AI的搜索方法的性能非常复杂。幸运的是，对于本章的目的而言，我们只需要关心下面两个尺度：

- 发现解决方案的速度
- 解决方案的质量

有这样一些问题，对这些问题而言最关键的是用最小的努力找到一个解决方案，任何方案都可以。对于这种问题，第一个尺度尤其重要。在其他情况下，解决方案的质量更加重要。

搜索的速度受搜索空间的大小以及在寻找解决方案的过程中实际经过的节点数目的影响。由于从死胡同返回是一种不必要的消耗，因此需要一种很少折回的搜索方法。

在基于AI的搜索中，寻找最优解决方案与寻找好的解决方案不同。寻找最优的解决方案需要使用穷举搜索，因为有时这是确定已经找到最优的解决方案的惟一方法。相反，寻找一个好的解决方案意味着在一套约束内寻找解决方案——是否存在更好的解决方案并不重要。

正如您所看到的那样，本章所描述的搜索技术在某种情况下比在其他情况下好。很难说某种搜索方法总是比其他的方法优越，但是对于平均情况，有些搜索技术很可能会更好。另外，问题的定义方式有时也会有助于选择合适的搜索方法。

7.4 需要解决的问题

现在，考虑我们将使用不同的搜索方法来解决的问题。假定您是一位旅行代理人，一个相当挑剔的客户想让您预定XYZ航空公司的从纽约到洛杉矶的航班。您告诉客户XYZ没有直接从纽约到洛杉矶的航班，但是客户坚持说，XYZ是他惟一想要乘坐的飞机的航空公司。因此，您必须寻找连接纽约和洛杉矶之间的航线。您可以参考XYZ的定期航班，如表7-2所示。

表7-2 航班和距离

航　班	距　离
纽约到芝加哥	900英里
芝加哥到丹佛	1000英里
纽约到多伦多	500英里
纽约到丹佛	1800英里
多伦多到卡尔加里	1700英里

（续表）

航　班	距　离
多伦多到洛杉矶	2500 英里
多伦多到芝加哥	500 英里
丹佛到乌尔班纳	1000 英里
丹佛到休斯顿	1000 英里
休斯顿到洛杉矶	1500 英里
丹佛到洛杉矶	1000 英里

您很快看到可以使您的客户从纽约飞到洛杉矶的连接。问题在于如何把您脑海里想到的内容编写为 C++程序。

图的表示

XYZ 时间表上的航班信息可以转换为有向图，如图 7-4 所示。有向图是一幅简单的图形，其中使用线段来连接每个节点，并使用箭头指示动作的方向。在有向图中，不能逆着箭头的方向旅行。

图 7-4　XYZ 的定期航班的有向图

为了使得问题易于理解，在图 7-5 中以树形格式重新绘制了这幅图。本章的剩余部分使用图 7-5。目标(洛杉矶)被圈了起来。另外还要注意，为了简化图的结构，各个城市都出现了不止一次。因此，树形的表示不是表述了一个二叉树，而只是为了便于分析。

现在我们准备开发不同的搜索技术来查找从纽约到洛杉矶的航线。

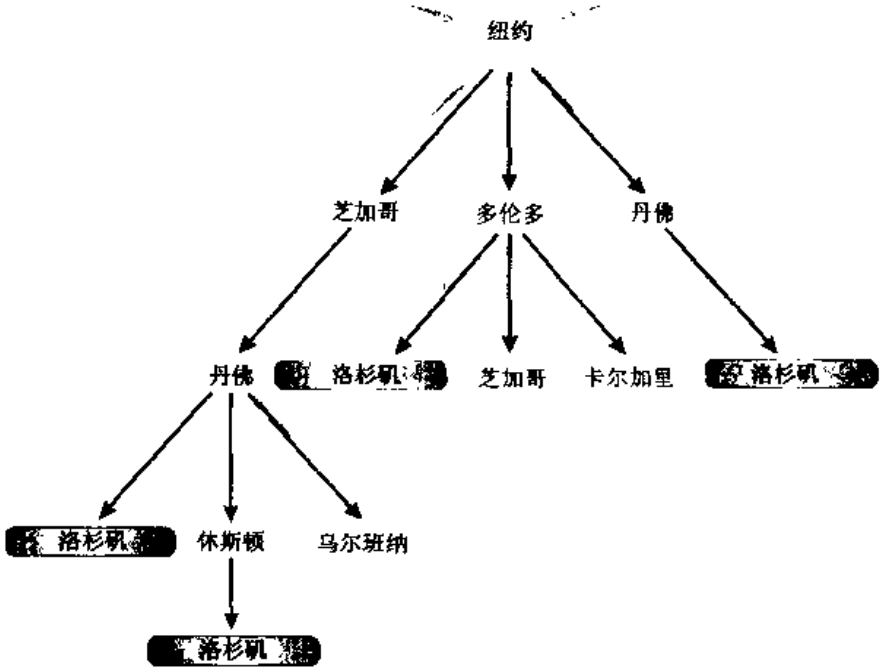

图 7-5 XYZ 定期航班的树形图

7.5 FlightInfo 结构和 Search 类

编写一个寻找从纽约到洛杉矶的航线的程序需要一个包含了航班信息的数据库。数据库中的每一个记录项都必须包含出发的城市和目标城市、两个城市之间的距离，以及帮助沿原路返回的标记。这些信息保存在名为 FlightInfo 的结构中，如下所示：

```
// Flight information.
struct FlightInfo {
  string from; // departure city
  string to; // destination city
  int distance; // distance between from and to
  bool skip; // used in backtracking

  FlightInfo() {
    from = "";
    to = "";
    distance = 0;
    skip = false;
  }

  FlightInfo(string f, string t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
```

```
  }
};
```

本章剩余部分所描述的全部搜索方法都用到这个结构。

基于 AI 的搜索被封装到名为 Search 的类中。这个类精确的实现随搜索方法的不同而不同。然而，所有的版本都具有相同的常规架构，如下所示：

```
// An AI-based search class.
class Search {
  // This vector holds the flight information.
  vector<FlightInfo> flights;

  // This stack is used for backtracking.
  stack<FlightInfo> btStack;

  // If there is a flight between from and to,
  // store the distance of the flight in dist.
  // Return true if the flight exists and,
  // false otherwise.
  bool match(string from, string to, int &dist);

  // Given from, find any connection.
  // Return true if a connection is found,
  // and false otherwise.
  bool find(string from, FlightInfo &f);

public:

  // Put flights into the database.
  void addflight(string from, string to, int dist) {
    flights.push_back(FlightInfo(from, to, dist));
  }

  // Show the route and total distance.
  void route();

  // Determine if there is a route between from and to.
  void findroute(string from, string to);

  // Return true if a route has been found.
  bool routefound() {
    return !btStack.empty();
  }
};
```

Search 声明了两个私有实例变量。第一个是 FlightInfo 对象的 vector 对象，名为 flights，保存航班的数据。(vector 是一个实现动态数组的 STL 容器)。第二个是一个堆栈，名为 btStack，用它来沿原路返回。您将会看到，对于所有的搜索方法，这个堆栈都很重要。

Search 包含了两个内联函数：addflight()和 routefound()函数。当第一次创建 Search 对象时，它的 flights 向量是空的。连接是通过重复地调用 addflight()，加入指定出发城市、目标城市以

及两个城市之间的距离。addflight()函数简单地把每个连接放入 flights 向量的结尾，这个向量的大小会自动扩展以适应新的条目。

routefound()函数基于返回堆栈中的内容判断出发城市和目标城市之间的航线是否存在。如果堆栈返回为空，两个城市之间不存在航线。否则，返回堆栈包含这个航线(创建航线的过程随搜索方法的不同而不同)。

Search 其余成员的实现在下节描述。表 7-3 简要描述了这些成员函数的功能。

表 7-3 Search 其余成员的目的

Search 其余成员	目　　的
match()	判断两个城市之间是否有直接的连接
find()	尝试寻找指定城市到任何其他城市的连接
findroute()	尝试在出发城市和目标城市之间找到一条航线
route()	显示航线

7.6 深度优先搜索

深度优先搜索在尝试另一种路径之前，会探索每条可能的路径，直到结束。为了理解它确切的运行方式，考虑图 7-6 所示的树形结构。F 是目标。

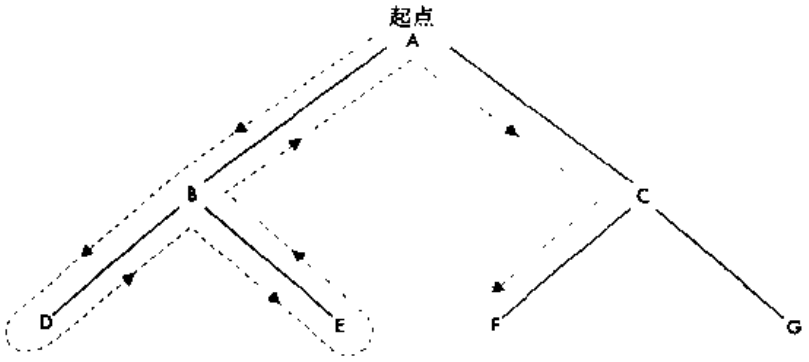

图 7-6 树形结构

深度优先搜索以下面的顺序遍历图 7-6：ABDBEBACF。如果您对树熟悉的话，应该知道这种搜索方法是树的中序遍历。也就是说，路径一直向左，直到遇到终端节点或者找到目标。如果到达终端节点，路径回退到上一层，转向右边，然后再向左边搜索，直到遇到目标或者终端节点。这个过程一直持续到发现目标或者检查到搜索空间中的最后一个节点。

正如您所看到的那样，深度优先搜索总是能够找到目标，因为在最坏的情况下，它退化为穷举搜索。在这个示例中，如果 G 是目标，那么就会导致穷举搜索。

整个深度优先搜索程序如下所示：

```
// Search for a route.
#include <iostream>
#include <stack>
#include <string>
#include <vector>
```

```
using namespace std;

// Flight information.
struct FlightInfo {
  string from; // departure city
  string to; // destination city
  int distance; // distance between from and to
  bool skip; // used in backtracking

  FlightInfo() {
    from = "";
    to = "";
    distance = 0;
    skip = false;
  }

  FlightInfo(string f, string t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
};

// Find connections using a depth-first search.
class Search {
  // This vector holds the flight information.
  vector<FlightInfo> flights;

  // This stack is used for backtracking.
  stack<FlightInfo> btStack;

  // If there is a flight between from and to,
  // store the distance of the flight in dist.
  // Return true if the flight exists and
  // false otherwise.
  bool match(string from, string to, int &dist);

  // Given from, find any connection.
  // Return true if a connection is found,
  // and false otherwise.
  bool find(string from, FlightInfo &f);

public:

  // Put flights into the database.
  void addflight(string from, string to, int dist) {
    flights.push_back(FlightInfo(from, to, dist));
  }
```

```
  // Show the route and total distance.
  void route();

  // Determine if there is a route between from and to.
  void findroute(string from, string to);

  // Return true if a route has been found.
  bool routefound() {
    return !btStack.empty();
  }
};

  // Show the route and total distance.
  void Search::route()
  {
    stack<FlightInfo> rev;
    int dist = 0;
    FlightInfo f;

    // Reverse the stack to display route.
    while(!btStack.empty()) {
      f = btStack.top();
      rev.push(f);
      btStack.pop();
    }

    // Display the route.
    while(!rev.empty()) {
      f = rev.top();
      rev.pop();
      cout << f.from << " to ";
      dist += f.distance;
    }

    cout << f.to << endl;
    cout << "Distance is " << dist << endl;
  }

  // If there is a flight between from and to,
  // store the distance of the flight in dist.
  // Return true if the flight exists and,
  // false otherwise.
  bool Search::match(string from, string to, int &dist)
  {
    for(unsigned i=0; i < flights.size(); i++) {
      if(flights[i].from == from &&
         flights[i].to == to && !flights[i].skip)
      {
        flights[i].skip = true; // prevent reuse
```

```
      dist = flights[i].distance;
      return true;
    }
  }

  return false; // not found
}

// Given from, find any connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from && !flights[i].skip) {
      f = flights[i];
      flights[i].skip = true; // prevent reuse

      return true;
    }
  }

  return false;
}

// Depth-first version.
// Determine if there is a route between from and to.
void Search::findroute(string from, string to)
{
  int dist;
  FlightInfo f;

  // See if at destination.
  if(match(from, to, dist)) {
    btStack.push(FlightInfo(from, to, dist));
    return;
  }

  // Try another connection.
  if(find(from, f)) {
    btStack.push(FlightInfo(from, to, f.distance));
    findroute(f.to, to);
  }
  else if(!btStack.empty()) {
    // Backtrack and try another connection.
    f = btStack.top();
    btStack.pop();
    findroute(f.from, f.to);
  }
}
```

```
int main() {
  char to[40], from[40];
  Search ob;

  // Add flight connections to database.
  ob.addflight("New York", "Chicago", 900);
  ob.addflight("Chicago", "Denver", 1000);
  ob.addflight("New York", "Toronto", 500);
  ob.addflight("New York", "Denver", 1800);
  ob.addflight("Toronto", "Calgary", 1700);
  ob.addflight("Toronto", "Los Angeles", 2500);
  ob.addflight("Toronto", "Chicago", 500);
  ob.addflight("Denver", "Urbana", 1000);
  ob.addflight("Denver", "Houston", 1000);
  ob.addflight("Houston", "Los Angeles", 1500);
  ob.addflight("Denver", "Los Angeles", 1000);

  // Get departure and destination cities.
  cout << "From? ";

  cin.getline(from, 40);
  cout << "To? ";

  cin.getline(to, 40);

  // See if there is a route between from and to.
  ob.findroute(from, to);

  // If there is a route, show it.
  if(ob.routefound())
     ob.route();

  return 0;
}
```

注意 main()提示您输入出发城市和目标城市。这意味着可以使用这个程序来寻找任何两个城市之间的航线。然而，本章的其余部分假定纽约是出发城市，洛杉矶是目标城市。

当程序以纽约作为出发城市、洛杉矶作为目标城市运行的时候，会显示如下的解决方案：

```
From? New York
To? Los Angeles
New York to Chicago to Denver to Los Angeles
Distance is 2900
```

如果您参考图 7-7，就会发现这实际上是深度优先搜索发现的第一个解决方案。这也是相当不错的一个解决方案。

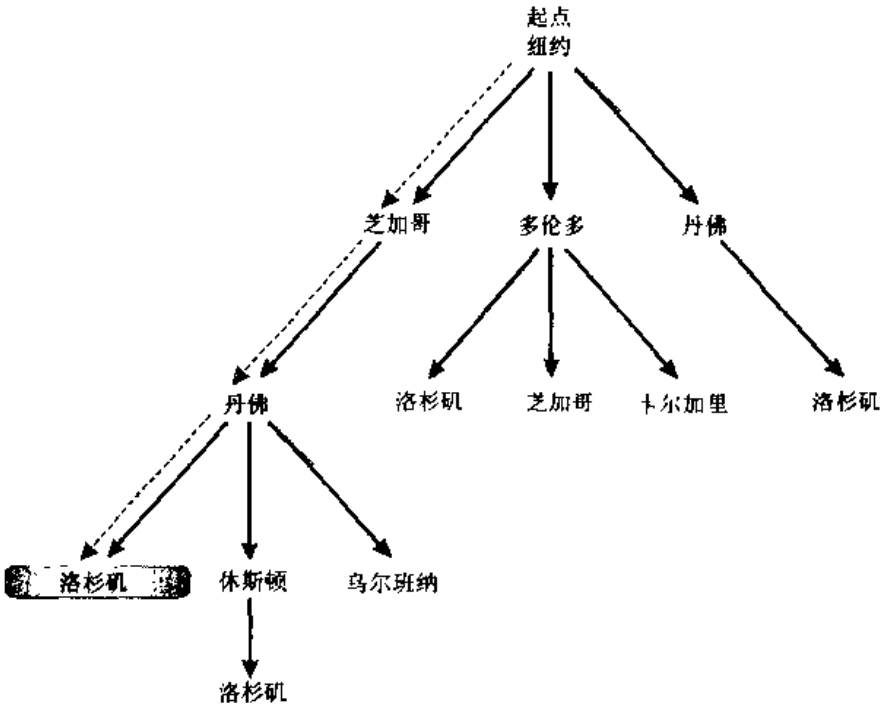

图 7-7 寻找解决方案的深度优先路径

正如 main()函数显示的那样，为了使用 Search，首先创建了一个 Search 对象。然后用这个连接加载航班数据库。随后，调用 findroute()试图在出发城市与目标城市之间寻找一条航线。为了判断是否找到了这样的航线，调用了 routefound()函数。如果存在这样的航线，这个函数返回 true。为了显示这条航线，调用了 route()函数。现在让我们仔细分析深度优先搜索的每个部分。

7.6.1 match()函数

这里所给出的 match()函数实现判断两个城市之间是否有航班，这两个城市分别由 from 和 to 指定：

```
// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool Search::match(string from, string to, int &dist)
{
  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from &&
       flights[i].to == to && !flights[i].skip)
    {
      flights[i].skip = true; // prevent reuse
      dist = flights[i].distance;
      return true;
    }
  }

  return false; // not found
}
```

match()函数通过搜索 flights 向量来操作，在其中查找是否存在与 from 和 to 匹配的出发城市与目标城市的条目。如果没有这样的航班，则返回 false。如果存在这样的航班，则返回 true，在此情况下还会获取两个城市之间的距离，并将这个值存储在 dist 参数所引用的变量中。

注意 match()忽略了 skip 字段为 true 的连接。另外，如果找到了某个连接，就会设置 skip 字段。用它来管理从死胡同的返回，从而避免一次又一次地测试一个相同的连接。

7.6.2 find()函数

find()函数如下所示，用来在数据库中搜索从给定的出发城市开始的连接：

```
// Given from, find any connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from && !flights[i].skip) {
      f = flights[i];
      flights[i].skip = true; // prevent reuse
      return true;
    }
  }

  return false;
}
```

出发城市传递给 from。如果找到了一个连接航班，则 find()返回 true，与这个连接相关的 FlightInfo 对象存储在第二个参数 f 引用的变量中。否则，find()返回 false。match()与 find()之间的区别是 match()判断指定的两个城市之间是否存在航班；find()判断是否存在从给定城市到其他任何城市的航班。

如同 match()一样，find()也使用 skip 字段来管理从死胡同的返回。

7.6.3 findroute()函数

实际寻找连接航班的代码包含在 findroute()函数中，它是寻找两个城市之间航线的关键例程。函数调用时需要出发城市和目标城市的名称。

```
// Determine if there is a route between from and to
// by using depth-first searching.
void Search::findroute(string from, string to)
{
  int dist;
  FlightInfo f;

  // See if at destination.
  if(match(from, to, dist)) {
    btStack.push(FlightInfo(from, to, dist));
    return;
  }
```

```
    // Try another connection.
    if(find(from, f)) {
      btStack.push(FlightInfo(from, to, f.distance));
      findroute(f.to, to);
    }
    else if(!btStack.empty()) {
      // Backtrack and try another connection.
      f = btStack.top();
      btStack.pop();
      findroute(f.from, f.to);
    }
  }
```

让我们仔细分析 findroute()。首先，通过 match()检查航班数据库，以判断是否有从 from 到 to 的航班。如果有，那么已经到达目标，这个连接被压入堆栈，函数返回。否则，findroute()调用 find()来查找 from 与其他任何城市之间的连接。如果找到这样的连接，则 find()函数返回 true，在此情况下，存储 f 中描述这个连接的 FlightInfo 对象。如果没有有效的连接航班，则 find()返回 false。如果存在连接航班，当前的航班被压入返回堆栈中，并递归调用 findroute()，f.to 中的城市变为新的出发城市。否则，发生沿原路返回。前面的节点从堆栈中删除，并递归调用 findroute()。这个过程一直持续到发现目标，或者穷举尽数据库。

例如，如果使用纽约和芝加哥作为参数来调用 findroute()，第一个 if 将成功，findroute()将终止，因为纽约到芝加哥有直接的航班。当使用纽约和卡尔加里作为参数来调用 findroute()时，情况变得复杂起来。在此情况下，第一个 if 将会失败，因为这两个城市之间没有直接的航班。随后，尝试寻找纽约与任何其他城市之间的连接来测试第二个 if。在此情况下，find()首先找到纽约到芝加哥的连接，这个连接压入到返回堆栈中，然后使用芝加哥作为起点来递归调用 findroute()。遗憾的是，在此没有从芝加哥到卡尔加里的路线，随后是几条错误的路线。最终，在几次递归调用 findroute()以及沿原路返回之后，发现了从纽约到多伦多的连接，以及多伦多与卡尔加里的连接。从而返回 findroute()，并将这个过程中所有的递归调用解开。最后，返回对 findroute()的原始调用。您或许会想要在 findroute()中加入 cout 语句，来准确地观察使用其他不同的出发城市和目标城市时，这个函数的运行方式。

findroute()实际上并没有返回解决方案——它生成了解决方案，理解这一点很重要。一旦从 findroute()退出，返回堆栈中就包含了从出发点到目标点的路线。也就是说，解决方案保存在 btStack 中。另外，findroute()的成功或者失败由堆栈的状态决定。空的堆栈意味着失败，否则，堆栈会保存一个解决方案。

通常，返回是基于 AI 的搜索技术中的很重要因素。在 Search 中，返回是通过使用递归和返回堆栈来完成的(这就是 C++对递归和 STL 的支持使得它成为 AI 开发一个好的选择的原因)。几乎所有的返回的情况都类似于堆栈操作——也就是先进后出。当探索路径时，遇到节点就将其压入堆栈。在每个死胡同，最后的节点弹出堆栈，从这个点开始尝试新的路径。这个过程一直持续，直到到达目标或者搜索完所有的路径。

7.6.4 显示路线

route()函数显示路径和总的距离。这个函数如下所示：

```
// Show the route and total distance.
void Search::route()
{
  stack<FlightInfo> rev;
  int dist = 0;
  FlightInfo f;

  // Reverse the stack to display route.
  while(!btStack.empty()) {
    f = btStack.top();
    rev.push(f);
    btStack.pop();
  }

  // Display the route.
  while(!rev.empty()) {
    f = rev.top();
    rev.pop();
    cout << f.from << " to ";
    dist += f.distance;
  }

  cout << f.to << endl;
  cout << "Distance is " << dist << endl;
}
```

注意第二个名为 rev 的堆栈的使用。在 btStack 中存储的解决方案顺序相反，这个堆栈的栈顶保存着最后的连接，栈底保存着第一个连接。为此，必须将其反转，从而能够以正确的顺序来显示这个连接。为了将这个解决方案的顺序调整正确，必须将这个连接从 btStack 中弹出，并压入 rev 中。

7.6.5 深度优先搜索分析

深度优先搜索找到了一个好的解决方案。另外，对于这个特定的问题，深度优先搜索在第一次尝试的时候就找到了这个解决方案，而没有返回——这很好。然而，如果这些数据以不同的方式组织，寻找一个解决方案可能会涉及到相当多的返回。因此，这个示例的结果不能推广。更重要的是，当探索一个在末端没有解决方案的大的分支时，深度优先搜索的性能相当差。在此情况下，深度优先搜索不仅在探索这个分支上浪费了时间，在向目标返回时也是如此。

7.7 广度优先搜索

与深度优先搜索相对的是广度优先搜索。在这种方法中，在搜索下一层的节点之前，会检查这个层的每个节点。这种遍历方法如图 7-8 所示，C 为目标。

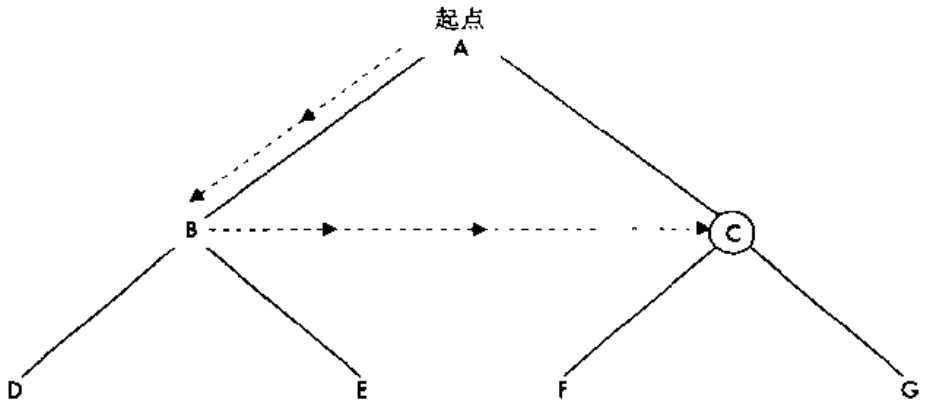

图 7-8 广度优先搜索的遍历方法

尽管二叉树结构的搜索空间很容易描述广度优先搜索的动作，然而许多的搜索空间，包括航班示例，并不是二叉树。因此，“广度”确切的组成有一点主观，是由手头的问题决定的。当涉及到航班示例时，广度优先搜索是通过检查是否存在离开出发城市的航班与到达目标城市的航班之间的连接来实现的。换句话说，在前进到另一层之前，会检查连接航班的所有连接的目的地。

为了使得 Search 执行广度优先搜索，需要修改 findroute()函数如下所示：

```
// Breadth-first version.
// Determine if there is a route between from and to.
void Search::findroute(string from, string to)
{
  int dist;
  FlightInfo f;

  // This stack is needed by the breadth-first search.
  stack<FlightInfo> resetStck;

  // See if at destination.
  if(match(from, to, dist)) {
    btStack.push(FlightInfo(from, to, dist));
    return;
  }

  // Following is the first part of the breadth-first
  // modification. It checks all connecting flights
  // from a specified node.
  while(find(from, f)) {
    resetStck.push(f);
    if(match(f.to, to, dist)) {
      resetStck.push(FlightInfo(f));
      btStack.push(FlightInfo(from, f.to, f.distance));
      btStack.push(FlightInfo(f.to, to, dist));
      return;
    }
  }

  // The following code resets the skip fields set by
```

```
  // preceding while loop. This is also part of the
  // breadth-first modification.
  while(!resetStck.empty()) {
    resetSkip(resetStck.top());
  resetStck.pop();
  }

  // Try another connection.
  if(find(from, f)) {
    btStack.push(FlightInfo(from, to, f.distance));
    findroute(f.to, to);
  }
  else if(!btStack.empty()) {
    // Backtrack and try another connection.
    f = btStack.top();
    btStack.pop();
    findroute(f.from, f.to);
  }
}
```

在此做了两个修改。首先，for 循环检查离开出发城市的所有航班，以判断它们是否与到达目标城市的航班相连接。随后，如果没有找到目标，通过调用 resetSkip()，将这些连接航班的 skip 字段清除，resetSkip()是必须加入到 Search 的一个新函数。需要重新设置的连接存储在它们自己的名为 resetStck 的堆栈中，这是 findroute()的局部变量。为了能够使用可能涉及到这些连接的其他路径，必须重新设置 skip 标志。

resetSkip()函数如下所示：

```
// Reset the skip fields in flights vector.
void Search::resetSkip(FlightInfo f) {
  for(unsigned i=0; i < flights.size(); i++)
    if(flights[i].from == f.from &&
       flights[i].to == f.to)
         flights[i].skip = false;
}
```

您需要在 Search 中加入这个函数(及其原型)。

为了尝试广度优先搜索，在前面的搜索程序中用新版本的 findroute()来替换，并加入 resetSkip()函数。当运行的时候，产生了如下的解决方案：

```
From? New York
To? Los Angeles
New York to Toronto to Los Angeles
Distance is 3000
```

图 7-9 显示了这个解决方案的广度优先路径。

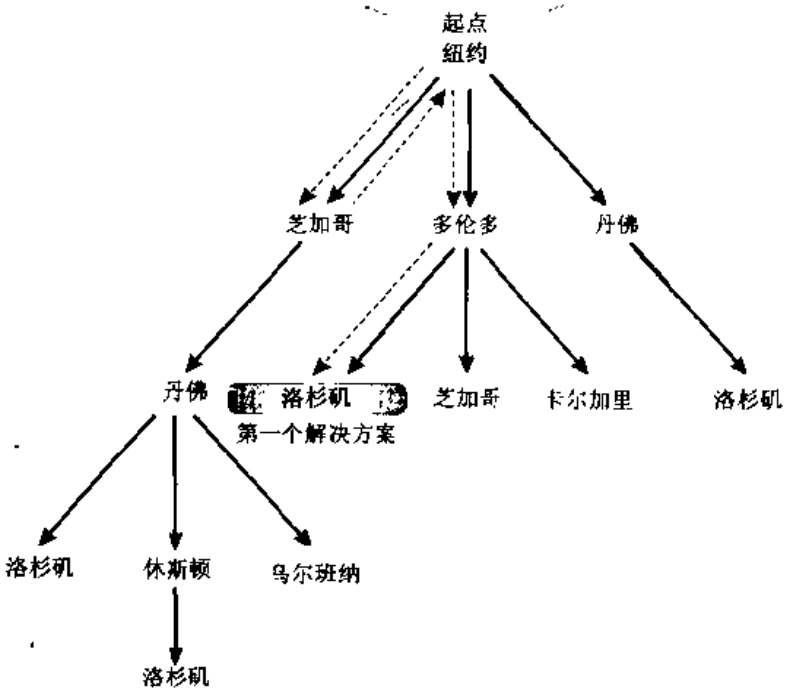

图 7-9 广度优先路径的解决方案

广度优先搜索分析

在这个示例中，广度优先搜索运行的相当好，并找到了一个合理的解决方案。与前面一样，这个结果不能被推广，因为第一条被发现的路径依赖于信息的组织方式。在相同的搜索空间内，深度优先搜索和广度优先搜索经常发现不同的路径，然而这个示例没有显示出这一点。

当目标在搜索空间埋藏地不是很深时，广度优先搜索运行良好。当目标有好几层深的时候，其运行情况比较差。在此情况下，广度优先搜索在返回阶段花费了许多功夫。

7.8 添加启发信息

深度优先搜索和广度优先搜索都没有对搜索空间中的某个节点是否比其他节点更接近于目标做出任何有根据的猜测。相反，它们只是使用先前的模式从一个节点移动到另一个节点，直到最后发现目标。在某些情况下，或许您只能做这么多，但是搜索空间经常包含了一些信息，可以使用这些信息来增加某个搜索比较快速地接近目标的可能性。为了利用这种信息，可以向搜索中增加启发的能力。

启发是用来增加某个搜索朝正确方向前进的可能性的规则。例如，假设您在森林中迷了路，并且需要喝水。森林非常茂密，以至于您不能远眺，这些树木很粗，您不能爬上去观望四周。然而您知道：河流、小溪以及池塘很可能在山谷中；动物们经常制造到水源的路径。当您接近水源时，可能会“闻到”它；您可以听见水流的声音。因此，您首先下山，因为水不大可能在山上。然后您偶然发现鹿的足迹也通向山下。您知道这些足迹可能通向水源，您就沿着这些足迹走。您开始听到细微的流动的声音出现在左边。您知道这可能是水，因此继续朝这个方向移

动。当您移动时，您开始发现空气变得潮湿；您可以闻到水了。最后，您找到了一条小溪，并且喝到了水。在这种情况下，用来找水的启发信息不一定会成功，但是它的确增加了提前成功的可能性。通常，启发信息增加了快速找到目标的可能性。

通常，启发式搜索方法是基于最小化或者最大化某个限制的信息。在安排从纽约到洛杉矶航班的示例中，旅客可能想要最小化两个限制。首先是必须创建的连接的数量。其次是航线的长度。记住，最短的航线不一定意味着最少的连接，反之亦然。在这节中，开发了两个启发式搜索。第一个最小化了连接的数量，第二个最小化了航线的距离。两种启发搜索都创建在深度优先搜索的架构上。

7.8.1 爬山搜索法

试图寻找最小化连接数量的航线的搜索算法使用了这样的启发信息：航班越长，将旅客带往距目标比较近的地方的可能性就越大；因此，连接的数目就会被最小化。用 AI 的语言来说，这是一个爬山的示例。

爬山算法选择距离目标比较近的节点作为下一步(也就是说，距离当前位置最远的节点)。这个名称来源于黑夜中半山腰迷路旅客的类比。假定这个旅客的营地在山顶，即使在黑夜中，这个旅客也知道每一点向上的步伐都是方向正确的一步。

只使用包含在航班安排的数据库中的信息，下面是将爬山启发信息整合到路线程序的方法：选择距离当前位置尽可能远的连接航班，以期望它能够距离目标比较近。为此，以 Search 的深度优先搜索版本开始，并修改 find()例程，如下所示：

```
// Hill-climbing version.
// Given from, find the farthest away connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
  int pos = -1;
  int dist = 0;

  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from && !flights[i].skip) {
      // Use the longest flight.
      if(flights[i].distance > dist) {
        pos = i;
        dist = flights[i].distance;
      }
    }
  }

  if(pos != -1) {
    f = flights[pos];
    flights[pos].skip = true; // prevent reuse

    return true;
  }
```

```
  return false;
}
```

现在，find()例程搜索整个数据库，查找距离出发城市最远的连接。
为了清楚起见，整个爬山程序如下所示：

```
// Search for a connection by hill climbing.
#include <iostream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

// Flight information.
struct FlightInfo {
  string from; // departure city
  string to; // destination city
  int distance; // distance between from and to
  bool skip; // used in backtracking

  FlightInfo() {
    from = "";
    to = "";
    distance = 0;
    skip = false;
  }

  FlightInfo(string f, string t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
};

// This version of search finds connections
// through the heuristic of hill climbing.
class Search {
  // This vector holds the flight information.
  vector<FlightInfo> flights;

  // This stack is used for backtracking.
  stack<FlightInfo> btStack;

  // If there is a flight between from and to,
  // store the distance of the flight in dist.
  // Return true if the flight exists and
```

```
  // false otherwise.
  bool match(string from, string to, int &dist);

  // Hill-climbing version.
  // Given from, find the farthest away connection.
  // Return true if a connection is found,
  // and false otherwise.
  bool find(string from, FlightInfo &f);

public:

  // Put flights into the database.
  void addflight(string from, string to, int dist) {
    flights.push_back(FlightInfo(from, to, dist));
  }

  // Show the route and total distance.
  void route();

  // Determine if there is a route between from and to.
  void findroute(string from, string to);

  // Return true if a route has been found.
  bool routefound() {
    return btStack.size() != 0;
  }
};

// Show the route and total distance.
void Search::route()
{
  stack<FlightInfo> rev;
  int dist = 0;
  FlightInfo f;

  // Reverse the stack to display route.
  while(!btStack.empty()) {
    f = btStack.top();
    rev.push(f);
    btStack.pop();
  }

  // Display the route.
  while(!rev.empty()) {
    f = rev.top();
    rev.pop();
    cout << f.from << " to ";
    dist += f.distance;
```

```
  cout << f.to << endl;
  cout << "Distance is " << dist << endl;
}

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool Search::match(string from, string to, int &dist)
{
  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from &&
       flights[i].to == to && !flights[i].skip)
    {
      flights[i].skip = true; // prevent reuse
      dist = flights[i].distance;
      return true;
    }
  }

  return false; // not found
}

// Hill-climbing version.
// Given from, find the farthest away connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
  int pos = -1;
  int dist = 0;

  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from && !flights[i].skip) {
      // Use the longest flight.
      if(flights[i].distance > dist) {
        pos = i;
        dist = flights[i].distance;
      }
    }
  }

  if(pos != -1) {
    f = flights[pos];
    flights[pos].skip = true; // prevent reuse

    return true;
  }
```

```
  return false;
}

// Determine if there is a route between from and to.
void Search::findroute(string from, string to)
{
  int dist;
  FlightInfo f;

  // See if at destination.
  if(match(from, to, dist)) {
    btStack.push(FlightInfo(from, to, dist));
    return;
  }

  // Try another connection.
  if(find(from, f)) {
    btStack.push(FlightInfo(from, to, f.distance));
    findroute(f.to, to);
  }
  else if(!btStack.empty()) {
    // Backtrack and try another connection.
    f = btStack.top();
    btStack.pop();
    findroute(f.from, f.to);
  }
}

int main() {
  char to[40], from[40];
  Search ob;

  // Add flight connections to database.
  ob.addflight("New York", "Chicago", 900);
  ob.addflight("Chicago", "Denver", 1000);
  ob.addflight("New York", "Toronto", 500);
  ob.addflight("New York", "Denver", 1800);
  ob.addflight("Toronto", "Calgary", 1700);
  ob.addflight("Toronto", "Los Angeles", 2500);
  ob.addflight("Toronto", "Chicago", 500);
  ob.addflight("Denver", "Urbana", 1000);
  ob.addflight("Denver", "Houston", 1000);
  ob.addflight("Houston", "Los Angeles", 1500);
  ob.addflight("Denver", "Los Angeles", 1000);

  // Get departure and destination cities.
  cout << "From? ";
```

```
    cin.getline(from, 40);
    cout << "To? ";

    cin.getline(to, 40);

    // See if there is a route between from and to.
    ob.findroute(from, to);

    // If there is a route, show it.
    if(ob.routefound())
       ob.route();

    return 0;
}
```

当程序运行时，得到的解决方案如下：

```
From? New York
To? Los Angeles
New York to Denver to Los Angeles
Distance is 2800
```

这个方案相当好。这条路线需要在路上停留的次数最少(只有一次)，并且这是最短的航线。因此，这个算法找到了可能是最好的航线。

然而，如果丹佛到洛杉矶的连接不存在，这个解决方案就不会如此完美。被找到的航线将会是从纽约到丹佛到休斯顿到洛杉矶，一共 4300 英里。在此情况下，这个解决方案爬上了一个“假山头”，因为到休斯顿的连接没有把我们带到距目标洛杉矶更近的地方。图 7-10 显示了第一个解决方案以及到“假山头”的路线。

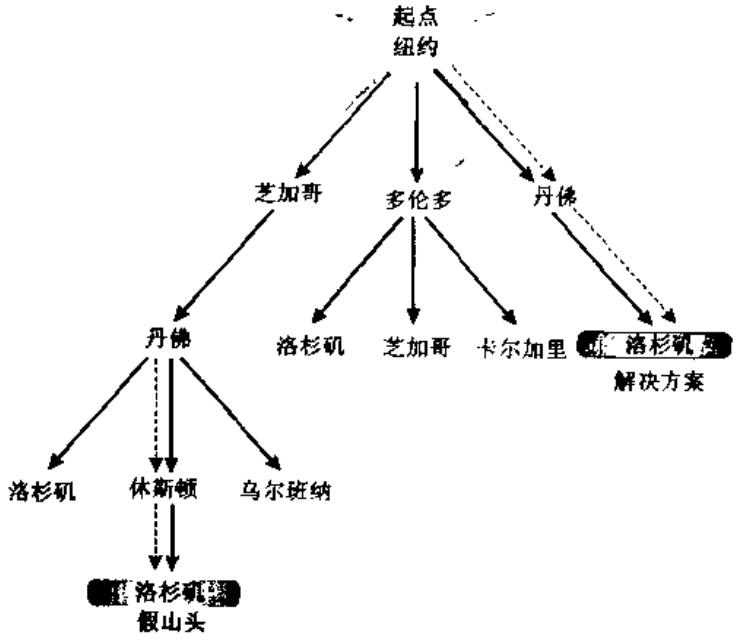

图 7-10　爬山法找到的解决方案的路径以及到假山头的路径

7.8.2 爬山法分析

在许多情况下，爬山法都能提供相当不错的解决方案，因为它试图降低在找到解决方案之前需要访问的节点的数量。然而，这个方法有三个缺点。首先是假山头的问题，如刚才所描述的那样。在此情况下，可能会导致很多的返回。第二个问题涉及到平稳状态，这是一种所有的下一步看上去都同样好(或者差)的情况。在此情况下，爬山法并不比深度优先搜索好。最后一个问题是山脊。在此情况下，爬山法的性能很差，因为在返回时，这个算法导致多次穿越山脊。尽管有这些潜在的问题，爬山法还是经常会增加找到好的解决方案的可能性。

7.9 最低成本搜索

与爬山搜索相对的是最低成本搜索。这种策略类似于穿着四轮旱冰鞋站在一座大山的路中间。毫无疑问您会觉得下山要比上山容易的多。换句话说，最低成本搜索法将采用阻力最小的路线。

将最低成本搜索法应用于航班安排问题，意味着将采用最短的连接航班，从而被找到的路线很有可能覆盖了最短的距离。不同于试图最小化连接数量的爬山法，最低成本搜索法试着最小化距离。

为了使用最低成本搜索，必须修改前面程序的 find()函数，如下所示：

```
const int MAXDIST = 100000;

// Least-cost version.
// Given from, find the closest connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
  int pos = -1;
  int dist = MAXDIST; // longer than longest flight

  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from && !flights[i].skip) {
      // Use the shortest flight.
      if(flights[i].distance < dist) {
        pos = i;
        dist = flights[i].distance;
      }
    }
  }

  if(pos != -1) {
    f = flights[pos];
    flights[pos].skip = true; // prevent reuse

    return true;
  }

  return false;
}
```

使用这个版本的 find()函数，找到的解决方案如下：

```
From? New York
To? Los Angeles
New York to Toronto to Los Angeles
Distance is 3000
```

正如您所看到的那样，这个搜索找到了一条好的航线，虽然不是最好的，但是可以接受。图 7-11 显示了到达目标的最低成本路径。

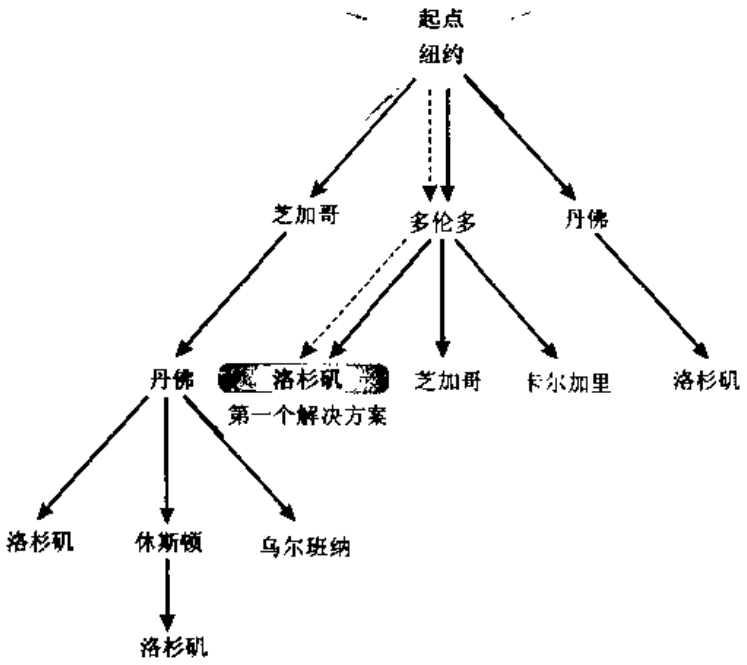

图 7-11　最低成本搜索法的解决方案路径

最低成本搜索分析

最低成本搜索和爬山法有着类似的优缺点，但刚好相反。在最低成本搜索中，可能有虚假的山谷、低地以及峡谷。在这个特殊的示例中，这个方法与爬山法运行的同样好。

7.10　寻找多解

有时寻找相同问题的几种解决方案是有意义的。这并不是寻找所有的解决方案(如同穷举搜索那样)。相反，多解提供了搜索空间中解决方案的代表性样本。

有多种方法生成多解，但是在此只使用两种方法。第一种方法是路径删除，第二种方法是节点删除。正如其名称所显示的那样，生成没有冗余的多解要求将已经发现的解决方案从系统中删除。记住，所有的这些方法都没有试图寻找全部的解决方案。寻找全部的解决方案是另一个问题，通常不会这么尝试，因为这样做意味着穷举搜索。

7.10.1 路径删除

生成多解的路径删除方法将当前解决方案中的所有节点都从数据库中删除，然后尝试寻找另一种解决方案。基本上，路径删除是从树上剪除分支。为了使用路径删除来寻找多解，您需要修改 main()函数，如下所示：

```
// Path removal version.
int main() {
  char to[40], from[40];
  Search ob;

  // Add flight connections to database.
  ob.addflight("New York", "Chicago", 900);
  ob.addflight("Chicago", "Denver", 1000);
  ob.addflight("New York", "Toronto", 500);
  ob.addflight("New York", "Denver", 1800);
  ob.addflight("Toronto", "Calgary", 1700);
  ob.addflight("Toronto", "Los Angeles", 2500);
  ob.addflight("Toronto", "Chicago", 500);
  ob.addflight("Denver", "Urbana", 1000);
  ob.addflight("Denver", "Houston", 1000);
  ob.addflight("Houston", "Los Angeles", 1500);
  ob.addflight("Denver", "Los Angeles", 1000);

  // Get departure and destination cities.
  cout << "From? ";

  cin.getline(from, 40);
  cout << "To? ";

  cin.getline(to, 40);

  // Find multiple solutions.
  for(;;) {
    // See if there is a connection.
    ob.findroute(from, to);

    // If no new route was found, then end.
    if(!ob.routefound()) break;

    ob.route();
  }

  return 0;
}
```

在此，加入了一个 for 循环，这个循环一直迭代，直到返回堆栈为空。当返回堆栈为空时，不会发现解决方案(在此情况下，没有另外的解决方案)。不需要进行另外的修改，因为作为这个解决方案一部分的任何连接都会使得 skip 字段被标记。因此，这种连接再也不会被 *find()*函数发现，也不会是下一个解决方案的一部分；它不会被再次发现。

如果您使用了 Search 的原始深度优先版本(本章一开始所示)以及前面的 main()函数，会找到如下的解决方案：

```
From? New York
To? Los Angeles
New York to Chicago to Denver to Los Angeles
Distance is 2900
New York to Toronto to Los Angeles
Distance is 3000
New York to Denver to Houston to Los Angeles
Distance is 4300
```

这个搜索找到了 3 个解决方案。然而需要注意，它们都不是最好的解决方案。

7.10.2 节点删除

第二种强制生成另外的解决方案的方法是节点删除法，这种方法只是简单地删除当前解决方案的最后一个节点，并再次尝试。为此，需要修改 main()函数，从航班数据库中删除最后一个连接，清除所有的 skip 字段，并获取一个新的空堆栈以保存下一个解决方案。更新的 main()函数如下所示：

```
// Node removal version
int main() {
  char to[40], from[40];
  Search ob;
  FlightInfo f;

  // Add flight connections to database.
  ob.addflight("New York", "Chicago", 900);
  ob.addflight("Chicago", "Denver", 1000);
  ob.addflight("New York", "Toronto", 500);
  ob.addflight("New York", "Denver", 1800);
  ob.addflight("Toronto", "Calgary", 1700);
  ob.addflight("Toronto", "Los Angeles", 2500);
  ob.addflight("Toronto", "Chicago", 500);
  ob.addflight("Denver", "Urbana", 1000);
  ob.addflight("Denver", "Houston", 1000);
  ob.addflight("Houston", "Los Angeles", 1500);
  ob.addflight("Denver", "Los Angeles", 1000);

  // Get departure and destination cities.
  cout << "From? ";

  cin.getline(from, 40);
  cout << "To? ";

  cin.getline(to, 40);

  // Find multiple solutions.
  for(;;) {
    // See if there is a connection.
```

```
    ob.findroute(from, to);

    // If no new route was found, then end.
    if(!ob.routefound()) break;

    // Save the flight on top-of-stack.
    f = ob.getTOS();

    ob.route(); // display the current route.

    ob.resetAllSkip(); // reset the skip fields

    // Remove last flight in previous solution
    // from the flight database.
    ob.remove(f);
  }

  return 0;
}
```

为了从航班数据库中删除前面解决方案的最后一个连接，main()函数首先调用 getTOS()来获取这个连接，getTOS()是在 Search 类中声明的内联函数，如下所示：

```
// Return flight on top of stack.
FlightInfo getTOS() {
  return btStack.top();
}
```

这个函数返回回溯堆栈的栈顶处的航班信息，这是路线中的最后一个连接。

为了删除这个连接，main()函数调用 remove()函数，如下所示：

```
// Remove a connection.
void Search::remove(FlightInfo f) {
  for(unsigned i=0; i< flights.size(); i++)
    if(flights[i].from == f.from &&
      flights[i].to == f.to)
        flights[i].from = "";
}
```

通过将出发城市的名称赋予长度为 0 的字符串，完成了对这个连接的删除。

为了清除 skip 字段，main()函数调用 resetAllSkip()，如下所示：

```
// Reset all skip fields.
void Search::resetAllSkip() {
  for(unsigned i=0; i< flights.size(); i++)
    flights[i].skip = false;
}
```

这个函数只是简单地将 skip 字段设置为 false。(记住将函数 resetAllSkip()和 remove()的原型加入到 Search 类的声明中)。

由于需要这么多的修改，为了清楚起见，在此显示了全部的节点删除程序。注意它也使用

了 Search 的原始深度优先版本。

```
// Search for multiple routes by use of node removal.
#include <iostream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

// Flight information.
struct FlightInfo {
  string from;     // departure city
  string to;       // destination city
  int distance;    // distance between from and to
  bool skip;       // used in backtracking

  FlightInfo() {
    from = "";
    to = "";
    distance = 0;
    skip = false;
  }

  FlightInfo(string f, string t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
};

// Find multiple solutions via node removal.
class Search {
  // This vector holds the flight information.
  vector<FlightInfo> flights;

  // This stack is used for backtracking.
  stack<FlightInfo> btStack;

  // If there is a flight between from and to,
  // store the distance of the flight in dist.
  // Return true if the flight exists and
  // false otherwise.
  bool match(string from, string to, int &dist);

  // Given from, find any connection.
  // Return true if a connection is found,
  // and false otherwise.
  bool find(string from, FlightInfo &f);

public:
```

```
  // Put flights into the database.
  void addflight(string from, string to, int dist) {
    flights.push_back(FlightInfo(from, to, dist));
  }

  // Show the route and total distance.
  void route();

  // Determine if there is a route between from and to.
  void findroute(string from, string to);

  // Return true if a route has been found.
  bool routefound() {
    return btStack.size() != 0;
  }

  // Return flight on top of stack.
  FlightInfo getTOS() {
    return btStack.top();
  }

  // Reset all skip fields.
  void resetAllSkip();

  // Remove a connection.
  void remove(FlightInfo f);
};

// Show the route and total distance.
void Search::route()
{
  stack<FlightInfo> rev;
  int dist = 0;
  FlightInfo f;

  // Reverse the stack to display route.
  while(!btStack.empty()) {
    f = btStack.top();
    rev.push(f);
    btStack.pop();
  }

  // Display the route.
  while(!rev.empty()) {
    f = rev.top();
    rev.pop();
    cout << f.from << " to ";
    dist += f.distance;
  }

  cout << f.to << endl;
```

```
  cout << "Distance is " << cist << endl;
}

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool Search::match(string from, string to, int &dist)
{
  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from &&
       flights[i].to == to && !flights[i].skip)
    {
      flights[i].skip = true; // prevent reuse
      dist = flights[i].distance;
      return true;
    }
  }

  return false; // not found
}

// Given from, find any connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from && !flights[i].skip) {
      f = flights[i];
      flights[i].skip = true; // prevent reuse

      return true;
    }
  }

  return false;
}

// Determine if there is a route between from and to.
void Search::findroute(string from, string to)
{
  int dist;
  FlightInfo f;

  // See if at destination.
  if(match(from, to, dist)) {
    btStack.push(FlightInfo(from, to, dist));
    return;
  }

  // Try another connection.
```

```
  if(find(from, f)) {
    btStack.push(FlightInfo(from, to, f.distance));
    findroute(f.to, to);
  }
  else if(!btStack.empty()) {
    // Backtrack and try another connection.
      f = btStack.top();
      btStack.pop();
      findroute(f.from, f.to);
    }
  }

  // Reset all skip fields.
  void Search::resetAllSkip() {
    for(unsigned i=0; i< flights.size(); i++)
      flights[i].skip = false;
  }

  // Remove a connection.
  void Search::remove(FlightInfo f) {
    for(unsigned i=0; i< flights.size(); i++)
      if(flights[i].from == f.from &&
        flights[i].to == f.to)
          flights[i].from = "";
}

  // Node removal version.
  int main() {
    char to[40], from[40];
    Search ob;
    FlightInfo f;

    // Add flight connections to database.
    ob.addflight("New York", "Chicago", 900);
    ob.addflight("Chicago", "Denver", 1000);
    ob.addflight("New York", "Toronto", 500);
    ob.addflight("New York", "Denver", 1800);
    ob.addflight("Toronto", "Calgary", 1700);
    ob.addflight("Toronto", "Los Angeles", 2500);
    ob.addflight("Toronto", "Chicago", 500);
    ob.addflight("Denver", "Urbana", 1000);
    ob.addflight("Denver", "Houston", 1000);
    ob.addflight("Houston", "Los Angeles", 1500);
    ob.addflight("Denver", "Los Angeles", 1000);

  // Get departure and destination cities.
  cout << "From? ";

    cin.getline(from, 40);
    cout << "To? ";

    cin.getline(to, 40);
```

```
    // Find multiple solutions.
    for(;;) {
    // See if there is a connection.
    ob.findroute(from, to);

    // If no new route was found, then end.
    if(!ob.routefound()) break;

    // Save the flight on top-of-stack.
    f = ob.getTOS();

    ob.route(); // display the current route.

    ob.resetAllSkip(); // reset the skip fields

    // Remove last flight in previous solution
    // from the flight database.
    ob.remove(f);
  }

  return 0;
}
```

这个程序找到了如下的路线：

```
From? New York
To? Los Angeles
New York to Chicago to Denver to Los Angeles
Distance is 2900
New York to Chicago to Denver to Houston to Los Angeles
Distance is 4400
New York to Toronto to Los Angeles
Distance is 3000
```

在此情况下，第二个解决方案可能是最差的路线，但是找到了两个相当好的解决方案。注意节点删除法找到的这组解决方案与路径删除法找到的方案不同。不同的生成多解的方法通常会产生不同的结果。

7.11 寻找“最优”解决方案

所有前面的搜索技术都只关心寻找一个解决方案——任何解决方案都可以。当您观察启发式搜索时，会努力增加找到好的解决方案的可能性。但是却没有试图确保寻找最优解决方案。然而，有时您只想要最优解决方案。记住，这里所讲的“最优”，只是意味着使用一个生成多解的技术所能找到的最好的路线——实际上它可能不是最好的解决方案(当然，寻找最好的解决方案需要花费大量时间的穷举搜索)

在抛开很好使用的航班安排示例之前，考虑一个寻找给定约束下最优航线的程序，这个约束要求距离最小化。为此，这个程序使用了路径删除方法来生成多解，并使用最低成本法搜索

最小距离。寻找最短航线的关键是使得某个解决方案比前面生成的解决方案的航线短。当不再产生解决方案时，最优解决方案就确定了。

全部的“最优解决方案”程序如下所示。注意这个程序创建了一个附加的堆栈，名为 optimal，这个堆栈保存最优解，还有一个名为 minDist 的实例变量，用这个变量来跟踪距离。route()和 main()也有少量的修改。

```
// Find an "optimal" solution using least-cost with path removal.
#include <iostream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

// Flight information.
struct FlightInfo {
  string from; // departure city
  string to; // destination city
  int distance; // distance between from and to
  bool skip; // used in backtracking

  FlightInfo() {
    from = "";
    to = "";
    distance = 0;
    skip = false;
  }

  FlightInfo(string f, string t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
};

const int MAXDIST = 100000;

// Find connections using least cost.
class Optimal {
  // This vector holds the flight information.
  vector<FlightInfo> flights;

  // This stack is used for backtracking.
  stack<FlightInfo> btStack;

  // This stack holds the optimal solution.
  stack<FlightInfo> optimal;

  int minDist;
```

```
  // If there is a flight between from and to,
  // store the distance of the flight in dist.
  // Return true if the flight exists and
  // false otherwise.
  bool match(string from, string to, int &dist);

  // Least-cost version.
  // Given from, find the closest connection.
  // Return true if a connection is found,
  // and false otherwise.
  bool find(string from, FlightInfo &f);

  public:

  // Constructor
  Optimal() {
    minDist = MAXDIST;
  }

  // Put flights into the database.
  void addflight(string from, string to, int dist) {
    flights.push_back(FlightInfo(from, to, dist));
  }

  // Show the route and total distance.
  void route();

  // Display the optimal route.
  void Optimal::showOpt();

  // Determine if there is a route between from and to.
  void findroute(string from, string to);

  // Return true if a route has been found.
  bool routefound() {
    return btStack.size() != 0;
  }
};

// Show the route and total distance.
void Optimal::route()
{
  stack<FlightInfo> optTemp;
  int dist = 0;
  FlightInfo f;

  // Reverse the stack to display route.
  while(!btStack.empty()) {
    f = btStack.top();
    optTemp.push(f);
    btStack.pop();
    dist += f.distance;
```

```
  }

  // If shorter, keep this route.
  if(minDist > dist) {
    optimal = optTemp;
    minDist = dist;
  }
}

// Display the optimal route.
void Optimal::showOpt()
{
  FlightInfo f;
  int dist = 0;

  cout <<"Optimal solution is:\n";
  // Display the optimal route.
  while(!optimal.empty()) {
    f = optimal.top();
    optimal.pop();
    cout << f.from << " to ";
    dist += f.distance;
  }

  cout << f.to << endl;
  cout << "Distance is " << dist << endl;
}

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool Optimal::match(string from, string to, int &dist)
{
  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from &&
       flights[i].to == to && !flights[i].skip)
    {
      flights[i].skip = true; // prevent reuse
      dist = flights[i].distance;
      return true;
    }
  }

  return false; // not found
}

// Least-cost version.
// Given from, find the closest connection.
// Return true if a connection is found,
// and false otherwise.
bool Optimal::find(string from, FlightInfo &f)
```

```
{
  int pos = -1;
  int dist = MAXDIST; // longer than longest flight

  for(unsigned i=0; i < flights.size(); i++) {
    if(flights[i].from == from && !flights[i].skip)
    {
      // Use the shortest flight.
      if(flights[i].distance < dist) {
        pos = i;
        dist = flights[i].distance;
      }
    }
  }

  if(pos != -1) {
    f = flights[pos];
    flights[pos].skip = true; // prevent reuse

    return true;
  }

  return false;
}

// Determine if there is a route between from and to.
void Optimal::findroute(string from, string to)
{
  int dist;
  FlightInfo f;

  // See if at destination.
  if(match(from, to, dist)) {
    btStack.push(FlightInfo(from, to, dist));
    return;
  }

  // Try another connection.
  if(find(from, f)) {
    btStack.push(FlightInfo(from, to, f.distance));
    findroute(f.to, to);
  }
  else if(!btStack.empty()) {
    // Backtrack and try another connection.
    f = btStack.top();
    btStack.pop();
    findroute(f.from, f.to);
  }
}

// Find "optimal" solution by using least-cost with path removal.
int main() {
```

```
  char to[40], from[40];
  Optimal ob;

  // Add flight connections to database.
  ob.addflight("New York", "Chicago", 900);
  ob.addflight("Chicago", "Denver", 1000);
  ob.addflight("New York", "Toronto", 500);
  ob.addflight("New York", "Denver", 1800);
  ob.addflight("Toronto", "Calgary", 1700);
  ob.addflight("Toronto", "Los Angeles", 2500);
  ob.addflight("Toronto", "Chicago", 500);
  ob.addflight("Denver", "Urbana", 1000);
  ob.addflight("Denver", "Houston", 1000);
  ob.addflight("Houston", "Los Angeles", 1500);
  ob.addflight("Denver", "Los Angeles", 1000);

  // Get departure and destination cities.
  cout << "From? ";

  cin.getline(from, 40);
  cout << "To? ";

  cin.getline(to, 40);

  // Find multiple solutions.
  for(;;) {
    // See if there is a connection.
    ob.findroute(from, to);

    // If no route found, then end.
    if(!ob.routefound()) break;

    ob.route();
  }

  // Display optimal solution.
  ob.showOpt();

  return 0;

}
```

这个程序的输出如下所示：

```
From? New York
To? Los Angeles
Optimal solution is:
New York to Chicago to Denver to Los Angeles
Distance is 2900
```

在此情况下，这个“最优”解决方案并不是最好的，但是它仍然是非常好的一个解决方案。如前所述，当使用基于 AI 的搜索时，某种搜索技术所能找到的最佳解决方案并不总是可能存

在的那个最佳方案。在前面的程序中，您或许想使用其他的搜索技术，来观察会找到什么样的“最优”解决方案。

前面的方法中有一个不足，即所有的路径都通向终点。改进的方法在长度等于或者大于当前最小长度时，就会停止跟踪这条路径。您可能需要修改这个程序来适应这种增强。

7.12 回到丢失钥匙的问题

为了总结本章关于问题的解决方案，提供一个搜索本章前面提到的丢失钥匙的 C++程序比较合适。下面的代码使用了与寻找两个城市间航线问题中相同的方法，因此没有对这个程序进行更多的解释：

```
// Search for the lost keys.
#include <iostream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

// Room information.
struct RoomInfo {
  string from;
  string to;
  bool skip;

  RoomInfo() {
    from = "";
    to = "";
    skip = false;
  }

  RoomInfo(string f, string t) {
    from = f;
    to = t;
    skip = false;
  }
};

// Find the keys using a depth-first search.
class Search {
  // This vector holds the room information.
  vector<RoomInfo> rooms;

  // This stack is used for backtracking.
  stack<RoomInfo> btStack;

  // Return true if a path exists between
  // from and to. Return false otherwise.
  bool match(string from, string to);
```

```
  // Given from, find any path.
  // Return true if a path is found,
  // and false otherwise.
  bool find(string from, RoomInfo &f);

public:

  // Put rooms into the database.
  void addroom(string from, string to)
  {
    rooms.push_back(RoomInfo(from, to));
  }

  // Show the route taken.
  void route();

  // Determine if there is a path between from and to.
  void findkeys(string from, string to);

  // Return true if the keys have been found.
  bool keysfound() {
    return !btStack.empty();
  }
};

// Show the route.
void Search::route()
{
  stack<RoomInfo> rev;
  RoomInfo f;

  // Reverse the stack to display route.
  while(!btStack.empty()) {
    f = btStack.top();
    rev.push(f);
    btStack.pop();
  }

  // Display the route.
  while(!rev.empty()) {
    f = rev.top();
    rev.pop();
    cout << f.from << " to ";
  }

  cout << f.to << endl;
}

// Return true if a path exists between
// from and to. Return false otherwise.
bool Search::match(string from, string to)
```

```
{
  for(unsigned i=0; i < rooms.size(); i++) {
    if(rooms[i].from == from &&
       rooms[i].to == to && !rooms[i].skip)
    {
      rooms[i].skip = true; // prevent reuse
      return true;
    }
  }

  return false; // not found
}

// Given from, find any path.
// Return true if a path is found,
// and false otherwise.
bool Search::find(string from, RoomInfo &f)
{
  for(unsigned i=0; i < rooms.size(); i++) {
    if(rooms[i].from == from && !rooms[i].skip) {
      f = rooms[i];
      rooms[i].skip = true; // prevent reuse

      return true;
    }
  }

  return false;
}

// Find the keys.
void Search::findkeys(string from, string to)
{
  RoomInfo f;

  // See if keys are found.
  if(match(from, to)) {
    btStack.push(RoomInfo(from, to));
    return;
  }

  // Try another room.
  if(find(from, f)) {
    btStack.push(RoomInfo(from, to));
    findkeys(f.to, to);
  }
  else if(!btStack.empty()) {
    // Backtrack and try another path.
    f = btStack.top();
    btStack.pop();
    findkeys(f.from, f.to);
  }
```

```
}

int main() {
  Search ob;

  // Add rooms to database.
  ob.addroom("front_door", "lr");
  ob.addroom("lr", "bath");
  ob.addroom("lr", "hall");
  ob.addroom("hall", "bd1");
  ob.addroom("hall", "bd2");
  ob.addroom("hall", "mb");
  ob.addroom("lr", "kitchen");
  ob.addroom("kitchen", "keys");

  // Find the keys.
  ob.findkeys("front_door", "keys");

  // If keys are found, show the path.
  if(ob.keysfound())
     ob.route();

  return 0;
}
```

7.13 尝试完成以下任务

由于基于 AI 的搜索是一个挑战，并且仍然是计算机科学中具有挑战性的一个领域，因此对它的体验充满了乐趣。在此有一些您可以尝试完成的任务。首先，用试探搜索方法替代广度优先方法并观察结果。其次，在寻找多解时，或者在寻找“最优”解决方案时，替代广度优先搜索方法。最后，试着处理其他现实生活中的搜索问题。