贪心法和动态规划

七月算法 邹博

2015年11月7日

主要内容

- 动态规划和贪心的认识
  - 工具：马尔科夫过程
- 贪心法
  - Prim算法
  - Kruskal算法
  - Dijkstra算法
- 动态规划
  - 最长递增子序列LIS
  - 矩阵连乘的最少乘法
  - 字符串的交替连接
  - 走棋盘/格子取数问题及其应用
  - 带陷阱的走棋盘问题
  - 两次走棋盘问题
  - Catalan数简介

Palindrome Partitioning所有划分

- 给定一个字符串str，将str划分成若干子串，使得每一个子串都是回文串。计算str的所有可能的划分。
  - 单个字符构成的字符串，显然是回文串；所以，这个的划分一定是存在的。
- 如：s=“aab”，返回
  - “aa”，“b”；
  - “a”，“a”，“b”。

回文划分问题Palindrome Partitioning

- 思考：若当前计算得到了str[0...i-1]的所有划分，可否添加str[i...j]，得到更大的划分呢？
  - 显然，若str[i...j]是回文串，则可以添加。
- 剪枝：在每一步都可以判断中间结果是否为合法结果。
  - 回溯+剪枝——如果某一次发现划分不合法，立刻对该分支限界。
  - 一个长度为n的字符串，最多有n-1个位置可以截断，每个位置有两种选择，因此时间复杂度为$O(2^{n-1})=O(2^n)$。

Code

```
void FindSolution(const char* str, int size, int nStart, vector<vector<string> >& all,
                  vector<string>& solution, const vector<vector<bool> >& p)
{
    if(nStart >= size)
    {
        all.push_back(solution);
        return;
    }
    for(int i = nStart; i < size; i++)
    {
        if(p[nStart][i])
        {
            solution.push_back(string(str+nStart, str+i+1));
            FindSolution(str, size, i+1, all, solution, p);
            solution.pop_back();
        }
    }
}

void MinPalindrome3(const char* str, vector<vector<string> >& all)
{
    int size = (int)strlen(str);
    vector<vector<bool> > p(size, vector<bool>(size));
    CalcSubstringPalindrome(str, size, p);

    vector<string> solution;    //一个解
    FindSolution(str, size, 0, all, solution, p);
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    const char* str = "abacdccdaa";
    vector<vector<string> > all;
    MinPalindrome3(str, all);
    PrintAll(all);
    return 0;
}
```

```
Count: 16
1: a | b | a | c | d | c | c | d | a | a
2: a | b | a | c | d | c | c | d | aa
3: a | b | a | c | d | cc | d | a | a
4: a | b | a | c | d | cc | d | aa
5: a | b | a | c | dccd | a | a
6: a | b | a | c | dccd | aa
7: a | b | a | cdc | c | d | a | a
8: a | b | a | cdc | c | d | aa
9: aba | c | d | c | c | d | a | a
10: aba | c | d | c | c | d | aa
11: aba | c | d | cc | d | a | a
12: aba | c | d | cc | d | aa
13: aba | c | dccd | a | a
14: aba | c | dccd | aa
15: aba | cdc | c | d | a | a
16: aba | cdc | c | d | aa
```

继续思考：动态规划

- 如果已知：str[0…i-1]的所有回文划分ϕ(i)，
- 如何求str[0…i]的所有划分呢？
  - 如果子串str[j…i]是回文串，则将该子串和ϕ(j-1)共同添加到ϕ(i+1)中。
- 算法：
  - 将集合ϕ(i+1)置空；
  - 遍历j(0≤j＜i)，若str[j,j+1…i]是回文串，则将{str[j…i]，ϕ(j-1)}添加到ϕ(i+1)中；
  - i从0到n，依次调用上面两步，最终返回ϕ(n)即为所求。

Code

```
Count: 16
1: aba | c | dccd | aa
2: a | b | a | c | dccd | aa
3: aba | c | d | cc | d | aa
4: a | b | a | c | d | cc | d | aa
5: aba | cdc | c | d | aa
6: a | b | a | cdc | c | d | aa
7: aba | c | d | c | c | d | aa
8: a | b | a | c | d | c | c | d | aa
9: aba | c | dccd | a | a
10: a | b | a | c | dccd | a | a
11: aba | c | d | cc | d | a | a
12: a | b | a | c | d | cc | d | a | a
13: aba | cdc | c | d | a | a
14: a | b | a | cdc | c | d | a | a
15: aba | c | d | c | c | d | a | a
16: a | b | a | c | d | c | c | d | a | a
```

```cpp
void Add(vector<vector<string> >& to, const vector<vector<string> >& from, const string& sub)
{
    if(from.empty())
    {
        to.push_back(vector<string>(1, sub));
        return;
    }
    to.reserve(from.size());
    for(vector<vector<string> >::const_iterator it = from.begin(); it != from.end(); it++)
    {
        to.push_back(vector<string>());
        vector<string>& now = to.back();
        now.reserve(it->size()+1);
        for(vector<string>::const_iterator s = it->begin(); s != it->end(); s++)
            now.push_back(*s);
        now.push_back(sub);
    }
}

void MinPalindrome4(const char* str, vector<vector<string> >& all)
{
    int size = (int)strlen(str);
    vector<vector<bool> > p(size, vector<bool>(size));
    CalcSubstringPalindrome(str, size, p);

    //prefix[i]：长度为i的前缀串的所有划分方案
    vector<vector<string> >* prefix = new vector<vector<string> >[size];
    prefix[0].clear();  //仅为了强调长度为0的串，划分解为空
    int i,j;
    for(i = 1; i <= size; i++) //考察str[0...i-1]
    {
        for(j = 0; j < i; j++)
        {
            if(p[j][i-1])
            {
                Add((i == size) ? all : prefix[i], prefix[j], string(str+j, str+i));
            }
        }
    }
    delete[] prefix;
}
```

Code *split*

```
void MinPalindrome4(const char* str, vector<vector<string> >& all)
{
    int size = (int)strlen(str);
    vector<vector<bool> > p(size, vector<bool>(size));
    CalcSubstringPalindrome(str, size, p);

    //prefix[i]：长度为i的前缀串的所有划分方案
    vector<vector<string> >* prefix = new vector<vector<string> >[size];
    prefix[0].clear();  //仅为了强调长度为0的串，划分解为空
    int i,j;
    for(i = 1; i <= size; i++) //考察str[0...i-1]
    {
        for(j = 0; j < i; j++)
        {
            if(p[j][i-1])
            {
                Add((i == size) ? all : prefix[i], prefix[j], string(str+j, str+i));
            }
        }
    }
    delete[] prefix;
}
```

Code *split*

```
void Add(vector<vector<string> >& to, const vector<vector<string> >& from, const string& sub)
{
    if(from.empty())
    {
        to.push_back(vector<string>(1, sub));
        return;
    }
    to.reserve(from.size());
    for(vector<vector<string> >::const_iterator it = from.begin(); it != from.end(); it++)
    {
        to.push_back(vector<string>());
        vector<string>& now = to.back();
        now.reserve(it->size()+1);
        for(vector<string>::const_iterator s = it->begin(); s != it->end(); s++)
            now.push_back(*s);
        now.push_back(sub);
    }
}
```

Palindrome Partitioning思考

- 与之类似的：
  - 给定仅包含数字的字符串，返回所有可能的有效IP地址组合。如：“25525511135”，返回“255.255.11.135”，“255.255.111.35”。
  - 该问题只插入3个分割位置。
  - 只有添加了第3个分割符后，才能判断当前划分是否合法。
    - 如：2.5.5.25511135，才能判断出是非法的。
      - 当然，它可以通过“25511135”大于“255.255”等其他限界条件“事先”判断。

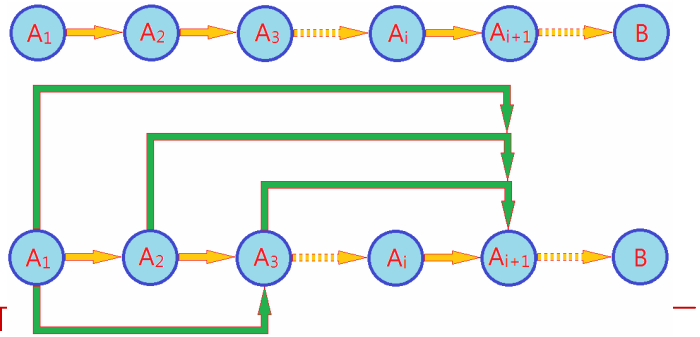
DFS与DP深刻认识

- DFS的过程，是计算完成了str[0…i]的切分，然后递归调用，继续计算str[i+1,i+2…n-1]的过程；
- 而DP中，假定得到了str[0…i-1]的所有可能切分方案，如何扩展得到str[0…i]的切分；
  - 上述两种方法都可以从后向前计算得到对偶的分析。
- 从本质上说，二者是等价的：最终都搜索了一颗隐式树。
  - DFS显然是深度优先搜索的过程，而DP更像层序遍历的过程。
  - 如果只计算回文划分的最少数目，动态规划更有优势；如果计算所有回文划分，DFS的空间复杂度比DP略优。

认识论

- 认识事物的方法：概念、判断、推理
- 推理中，又分为归纳、演绎。
- 重点考察归纳推理的具体方法。
- 形式化表述：
  - 已知：问题规模为n的前提A
  - 求解/求证：未知解B/结论B
  - 记号：用A_n表示“问题规模为n的已知条件”

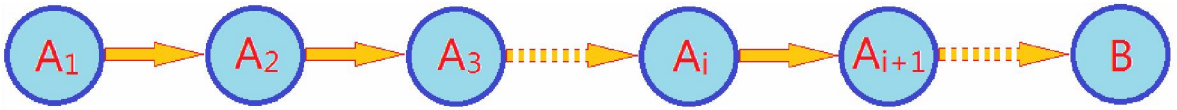
对归纳推理的理解



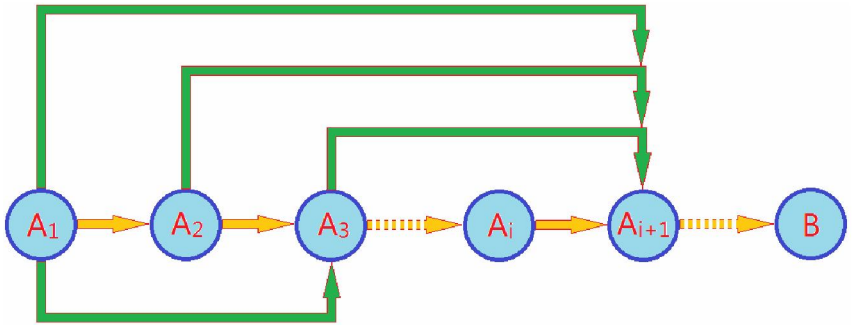
- 若将问题规模降低到0，即已知A_0，很容易计算或证明B，则有：$A_0 \rightarrow B$
- 同时，考察从A_0增加一个元素，得到A1的变化过程。即：$A_0 \rightarrow A_1$；
  - 进一步考察$A_1 \rightarrow A_2$，$A_2 \rightarrow A_3$ ……$A_i \rightarrow A_{i+1}$
  - 这种方法是(严格的)归纳推理，常常被称作**数学归纳法**。
  - 此时，由于上述推导往往不是等价推导(A_i和A_{i+1}不是互为充要条件)，导致随着i的增加，有价值的前提信息越来越少；为避免这一问题，采取如下方案：
  - $\{A_1\} \rightarrow A_2$，$\{A_1A_2\} \rightarrow A_3$ …… $\{A_1A_2 \ldots A_i\} \rightarrow A_{i+1}$
  - 相对应的，修正后的方法依然是(严格的)归纳推理，有时被称作**第二数学归纳法**。

对归纳推理的理解

- 基本归纳法：对于A_{i+1}，只需考察前一个状态A_i即可完成整个推理过程，它的特点是只要状态A_i确定，则计算A_{i+1}时不需要考察更前序的状态$A_1 \ldots A_{i-1}$，在图论中，常常称之为马尔科夫模型；

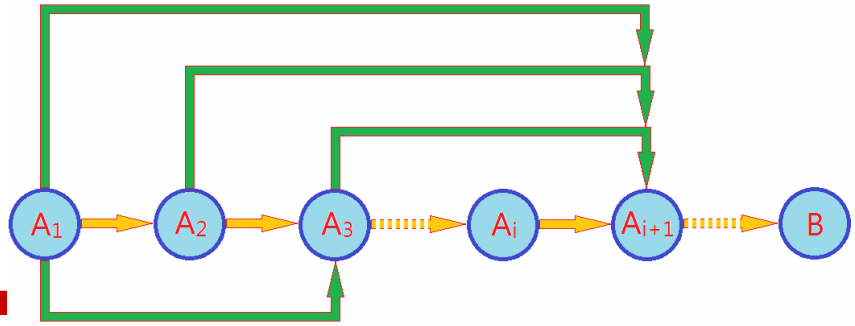


- 高阶归纳法：相应的，对于A_{i+1}，需考察前i个状态集$\{A_1A_2 \ldots A_{i-1}A_i\}$才可完成整个推理过程，往往称之为高阶马尔科夫模型；



- 在计算机算法中，高阶马尔科夫模型的推理，叫做“动态规划”，而马尔科夫模型的推理，对应“贪心法”。

以上理解的说明



- 无论动态规划还是贪心法，都是根据A[0...i]计算A[i+1]的过程
  - 计算A[i+1]不需要A[i+2]、A[i+3]……，
  - 一旦计算完成A[i+1]，再后面计算A[i+2]、A[i+3]……时，不会更改A[i+1]的值。
  - 这即无后效性。
- 亦可以如下理解动态规划：计算A[i+1]只需要知道A[0...i]的值，无需知道A[0...i]是通过何种途径计算得到的——只需知道它们当前的状态值本身即可。如果将A[0...i]的全体作为一个整体，则可以认为动态规划法是马尔科夫过程，而非高阶马尔科夫过程。

贪心法

- 根据实际问题，选取一种度量标准。然后按照这种标准对n个输入排序，并按序一次输入一个量。
- 如果输入和当前已构成在这种量度意义下的部分最优解加在一起不能产生一个可行解，则不把此输入加到这部分解中。否则，将当前输入合并到部分解中从而得到包含当前输入的新的部分解。
- 这一处理过程一直持续到n个输入都被考虑完毕，则记入最优解集合中的输入子集构成这种量度意义下的问题的最优解。
- 这种能够得到某种量度意义下的最优解的分级处理方法称为贪心方法。

最小生成树MST

- 最小生成树要求从一个带权无向连通图中选择n－1条边并使这个图仍然连通(也即得到了一棵生成树)，同时还要考虑使树的权最小。为了得到最小生成树，人们设计了很多算法，最著名的有Prim算法和Kruskal算法，这两个算法都是贪心算法。
  - Prim算法：从某个(任意一个)结点出发，选择与该结点邻接的权值最小的边；随着结点的不断加入，每次都选择这些结点发出的边中权值最小的；重复n-1次。
  - Kruskal算法：将边按照权值递增排序，每次选择权值最小并且不构成环的边，重复n-1次。

最短路径

- 将Prim算法稍做调整，就得到Dijkstra最短路径算法：
  - 结点集V初始化为源点S一个元素：V={S}，到每个点的最短路径的距离初始化为dist[u]=graph[S][u]；
  - 选择最小的dist[u]：记dist[v]是最小的，则v是当前找到的不在V中且距离S最近的结点，更新V=V∪{v}，调整dist[u]=min{dist[u],dist[v]+graph[v][u]}；
  - 重复n-1次。

贪心法的思考

- 可以看到，在从A_i到A_{i+1}的扩展过程中，上述三个算法都没有使用A[0...i-1]的值。
- 往往看名字，认为它很简单；事实上，贪心法其实并不轻松，它需要严格证明一定与更先序的值无关。
  - 思考：从1元、2元、5元的纸币，问给定总价值N元，最少需要几张纸币？

最长递增子序列LIS

- Longest Increasing Subsequence
- 给定一个长度为N的数组，找出一个最长的单调递增子序列(不一定连续，但是顺序不能乱)。例如：给定一个长度为6的数组A{5，6，7，1，2，8}，则其最长的单调递增子序列为{5，6，7，8}，长度为4。
  - 分析：其实此LIS问题可以转换成最长公子序列问题，为什么呢？

附：使用LCS解LIS问题

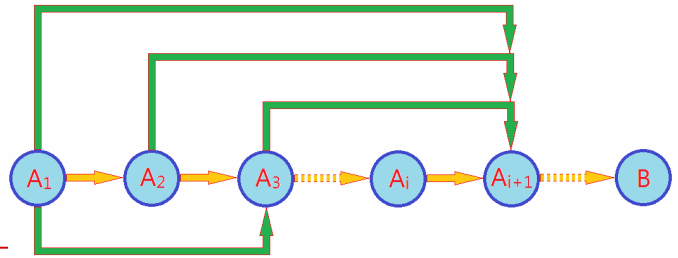
- 原数组为A {5，6，7，1，2，8}
- 排序后：A’{1，2，5，6，7，8}
- 因为，原数组A的子序列顺序保持不变，而且排序后A’本身就是递增的，这样，就保证了两序列的最长公共子序列的递增特性。如此，若想求数组A的最长递增子序列，其实就是求数组A与它的排序数组A’的最长公共子序列。

前缀分析

Array	1	4	6	2	8	9	7
LIS	1	2	3	2	4	5	4

- “1”：1
- “14”：14
- “146”：146
- “1462”：146
- “14628”：1468
- “146289”：14689
- “1462897”：14689

最长递增子序列LIS记号



- 长度为N的数组记为$A=\{a_0a_1a_2...a_{n-1}\}$；
- 记A的前i个字符构成的前缀串为$A_i=a_0a_1a_2...a_{i-1}$，以a_i结尾的最长递增子序列记做L_i，其长度记为b[i]；
- 假定已经计算得到了b[0,1…,i-1]，如何计算b[i]呢？
  - 已知$L_0\ L_1\ldots\ldots L_{i-1}$的前提下，如何求$L_i$？

求解LIS

Array	1	4	6	2	8	9	7
LIS	1	2	3	2	4	5	4

- 根据定义，L_i必须以a_i结尾；
- 如果将a_i分别缀到$L_0 L_1 \ldots\ldots L_{i-1}$后面，是否允许呢？
  - 如果$a_i \geqslant a_j$，则可以将a_i缀到L_j的后面，得到比L_j更长的字符串。
- 从而：$b[i]=\{\max(b[j])+1, 0 \leqslant j < i \text{且} a_j \leqslant a_i\}$
  - 计算b[i]：遍历在i之前的所有位置j，找出满足条件$a_j \leqslant a_i$的最大的b[j]+1；
  - 计算得到b[0…n-1]后，遍历所有的b[i]，找出最大值即为最大递增子序列的长度。
- 时间复杂度为$O(N^2)$。

LIS的思考

- 思考：如何求最大递增子序列本身？
  - 记录前驱

Array	1	4	6	2	8	9	7
LIS	1	2	3	2	4	5	4

Code1

```
int LIS1(const int* p, int length)
{
    int* longest = new int[length];
    int i,j;

    for(i = 0; i < length; i++)
        longest[i] = 1;

    int nLis = 1;
    for(i = 1; i < length; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(p[j] <= p[i])
            {
                longest[i] = max(longest[i], longest[j]+1);
            }
        }
        nLis = max(nLis, longest[i]);
    }
    delete[] longest;
    return nLis;
}
```

LIS Code split

```
int _tmain(int argc, _TCHAR* argv[])
{
    int array[] = {1,4,5,6,2,3,8,9,10,11,12,12,1};
    int size = sizeof(array)/sizeof(int);
    int* pre = new int[size];
    int nIndex;
    int max = LIS(array, size, pre, nIndex);
    vector<int> lis;
    GetLIS(array, pre, nIndex, lis);
    delete[] pre;
    cout << max << endl;
    Print(&lis.front(), (int)lis.size());
    return 0;
}
```

```
void GetLIS(const int* array, const int* pre,
            int nIndex, vector<int>& lis)
{
    while(nIndex >= 0)
    {
        lis.push_back(array[nIndex]);
        nIndex = pre[nIndex];
    }
    reverse(lis.begin(), lis.end());
}
```

```
#include <vector>
#include <algorithm>
using namespace std;

int LIS(const int* p, int length, int* pre, int& nIndex)
{
    int* longest = new int[length];
    int i,j;

    for(i = 0; i < length; i++)
    {
        longest[i] = 1;
        pre[i] = -1;
    }

    int nLis = 1;
    nIndex = 0;
    for(i = 1; i < length; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(p[j] <= p[i])
            {
                if(longest[i] < longest[j]+1)
                {
                    longest[i] = longest[j]+1;
                    pre[i] = j;
                }
            }
        }
        if(nLis < longest[i])
        {
            nLis = longest[i];
            nIndex = i;
        }
    }

    delete[] longest;

    return nLis;
}
```

```
void Print(int* p, int size)
{
    for(int i = 0; i < size; i++)
        cout << p[i] << '\t';
    cout << '\n';
}
```

O(NlogN)的最长递增子序列算法

- 对于数组A={1,4,6,2,8,9,7}
- 1
- 1,4
- 1,4,6
- 1,2,6
- 1,2,6,8
- 1,2,6,8,9
- 1,2,6,7,9

Code2

```
int LIS(const int* a, int size)
{
    int* b = new int[size];
    int s = 0;
    int i;
    for(i = 0; i < size; i++)
        Insert(b, s, a[i]);
    delete[] b;
    return s;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {5,6,7,1,2,8};
    int size = sizeof(a) / sizeof(int);
    int len = LIS(a, size);
    cout << len << endl;
    return 0;
}
```

```
void Insert(int* a, int& size, int x)
{
    if(size <= 0)
    {
        a[0] = x;
        size++;
        return;
    }
    int low = 0;
    int high = size-1;
    int mid;
    while(low <= high)
    {
        mid = (low+high)/2;
        if(x < a[mid])
            high = mid-1;
        else if(x >= a[mid])
            low = mid+1;
    }

    if(low >= size)
    {
        a[size] = x;
        size++;
    }
    else
    {
        if(a[low] < x)
            a[low+1] = x;
        else
            a[low] = x;
    }
}
```

矩阵乘积

- 根据矩阵相乘的定义来计算 C=A×B，需要m*n*s次乘法。
- 三个矩阵A、B、C的阶分别是$a_0 \times a_1$, $a_1 \times a_2$, $a_2 \times a_3$，从而(A×B)×C和A×(B×C)的乘法次数是$a_0a_1a_2+a_0a_2a_3$、$a_1a_2a_3+a_0a_1a_3$，二者一般情况是不相等的。
  - 问：给定n个矩阵的连乘积：$A_1 \times A_2 \times A_3 \ldots\ldots \times A_n$，如何添加括号来改变计算次序，使得乘法的计算量最小？
- 此外：若A、B都是n阶方阵，C的计算时间复杂度为$O(n^3)$
  - 问：可否设计更快的算法？
  - 答：分治法：Strassen分块——理论意义大于实践意义。

矩阵连乘的提法

- 给定n个矩阵$\{A_1,A_2,\ldots,A_n\}$，其中A_i与A_{i+1}是可乘的，i=1，2...，n-1。考察该n个矩阵的连乘积：$A_1 \times A_2 \times A_3 \ldots\ldots \times A_n$，确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的乘法次数最少。
  - 即：利用结合律，通过加括号的方式，改变计算过程，使得数乘的次数最少。

分析

- 将矩阵连乘积$A_iA_{i+1}...A_j$记为A[i:j]，这里i≤j
  - 显然，若i==j，则A[i:j]即A[i]本身。
- 考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，i≤k<j，则其相应的完全加括号方式为

$$(A_iA_{i+1}...A_k)(A_{k+1}A_{k+2}...A_j)$$

- 计算量：A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量

最优子结构

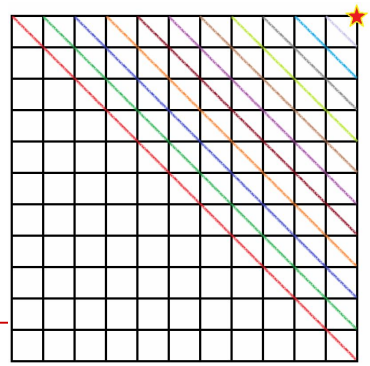
- 特征：计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。
- 最优子结构性质是可以使用动态规划算法求解的显著特征。

状态转移方程 $(A_iA_{i+1}\ldots A_k)(A_{k+1}A_{k+2}\ldots A_j)$

- 设计算A[i:j](1≤i≤j≤n)所需要的最少数乘次数为m[i,j]，则原问题的最优值为m[1,n]；
- 记A_i的维度为 $p_{i-1}\times p_i$
- 当i==j时，A[i:j]即A_i本身，因此，m[i,i]=0；(i=1,2,…,n)
- 当i<j时，$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$
- 从而：

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

矩阵连乘问题从算法到实现

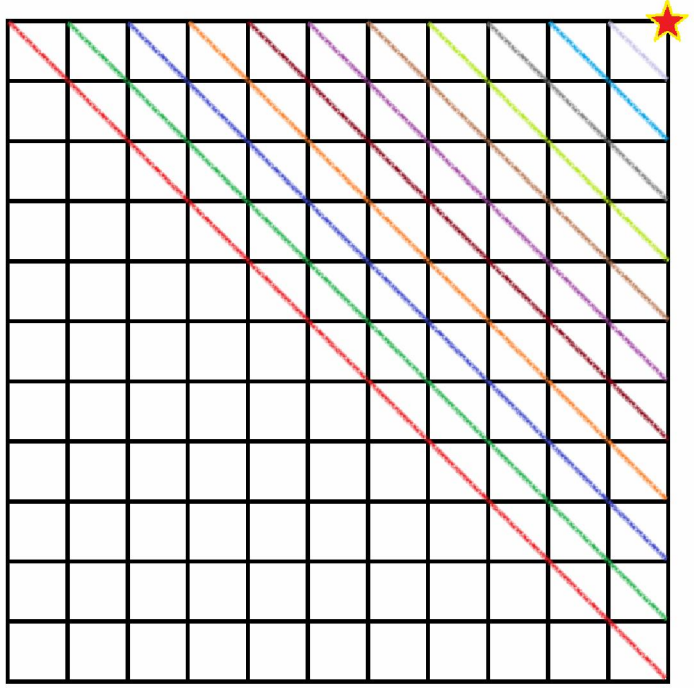


- 由m[i,j]的递推关系式可以看出，在计算m[i,j]时，需要用到m[i+1,j]，m[i+2,j]…m[j-1,j]；

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

- 因此，求m[i,j]的前提，不是m[0…i-1;0…j-1]，而是沿着主对角线开始，依次求取到右上角元素。
- 因为m[i,j]一个元素的计算，最多需要遍历n-1次，共$O(n^2)$个元素，故算法的时间复杂度是$O(n^3)$，空间复杂度是$O(n^2)$。

Code



```
//p[0...n]存储了n+1个数，其中，(p[i-1],p[i])是矩阵i的阶；
//s[i][j]记录A[i...j]从什么位置断开；m[i][j]记录数乘最小值
void MatrixMultiply(int* p, int n, int** m, int** s)
{
    int r,i,j,k,t;
    for(i = 1; i <= n; i++)
        m[i][i] = 0;

    //r个连续矩阵的连乘：上面的初始化，相当于r=1
    for(r = 2; r <= n; r++)
    {
        for(i = 1; i <= n-r+1; i++)
        {
            j=i+r-1;
            m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
            s[i][j] = i;
            for(k = i+1; k < j; k++)
            {
                t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if(t < m[i][j])
                {
                    m[i][j] = t;
                    s[i][j] = k;
                }
            }
        }
    }
}
```

找零钱

- 给定某不超过100万元的现金总额，兑换成数量不限的100、50、10、5、2、1的组合，共有多少种组合呢？

该问题的思考过程

- 此问题涉及两个类别：面额和总额。
  - 如果面额都是1元的，则无论总额多少，可行的组合数显然都为1。
  - 如果面额多一种，则组合数有什么变化呢？
- 定义dp[i][j]：使用面额小于等于i的钱币，凑成j元钱，共有多少种组合方法。
  - dp[100][500] = dp[50][500] + dp[100][400]
  - $dp[i][j] = dp[i_{small}][j] + dp[i][j-i]$
    - 不考虑j-i下溢出等边界问题

递推公式 dp[i][j] = dp[i_{small}][j] + dp[i][j-i]

- 使用dom[]={1,2,5,10,20,50,100}表示基本面额，i的意义从面额变成面额下标，则：
  - dp[i][j] = dp[i-1][j] + dp[i][j-dom[i]]
- 从而：

$$dp[i][j]=\begin{cases} dp[i-1][j]+dp[i][j-dom[i]], & j \geq dom[i] \\ dp[i-1][j], & j < dom[i] \end{cases}$$

- 初始条件：

$$\begin{cases} dp[0][j]=1 \\ dp[i][0]=1 \end{cases}$$

Code

```
int Charge(int value, const int* denomination, int size)
{
    int i;
    int** dp = new int*[size]; //dp[i][j]：用i面额以下的组合成j元
    for(i = 0; i < size; i++)
        dp[i] = new int[value+1];

    int j;
    for(j = 0; j <= value; j++)//只用面额1元的
        dp[0][j] = 1;

    for(i = 1; i < size; i++)  //先用面额小的，再用面额大的
    {
        dp[i][0] = 1;   //原因：添加任何一个面额，就是一个有效组合
        for(j = 1; j <= value; j++)
        {
            if(j >= denomination[i])
                dp[i][j] = dp[i-1][j] + dp[i][j-denomination[i]];
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    int time = dp[size-1][value];

    for(i = 0; i < size; i++)  //清理内存
        delete[] dp[i];
    delete[] dp;
    return time;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int denomination[] = {1,2,5,10,20,50,100}; //面额
    int size = sizeof(denomination) / sizeof(int);
    int value = 200;
    int c = Charge(value, denomination, size);
    cout << c << endl;
    return 0;
}
```

滚动数组

- 将状态转移方程去掉第一维，很容易使用滚动数组，降低空间使用量。
- 原状态转移方程：

$$dp[i][j]=\begin{cases} dp[i-1][j]+dp[i][j-dom[i]], & j \geq dom[i] \\ dp[i-1][j], & j < dom[i] \end{cases}$$

- 滚动数组版本的状态转移方程：

$$dp[j]=last[j]+dp[j-dom[i]], \quad (j \geq dom[i])$$

Code2

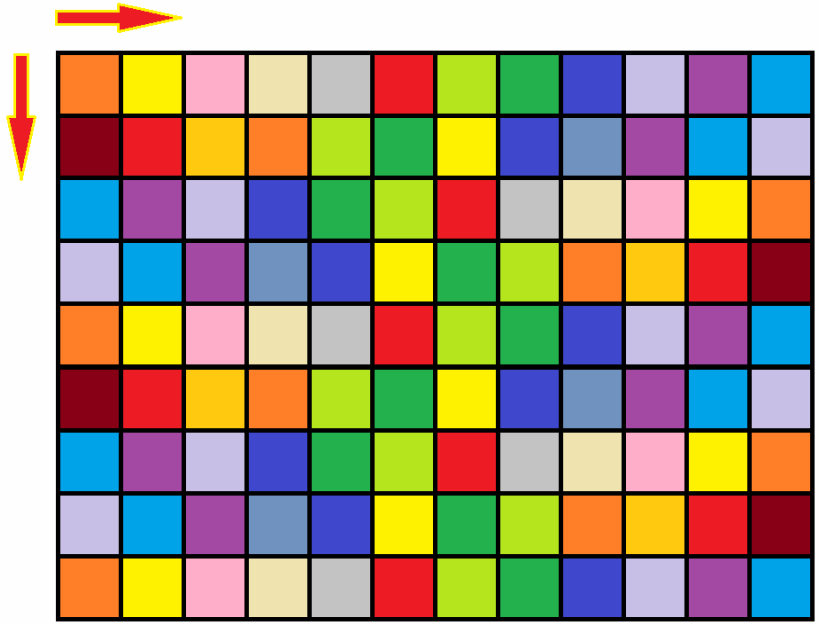
```
int Charge2(int value, const int* denomination, int size)
{
    int i;
    int* dp = new int[value+1]; //dp[j]：凑成j元的组合数
    int* last = new int[value+1];

    int j;
    for(j = 0; j <= value; j++)//只用面额1元的
    {
        dp[j] = 1;
        last[j] = 1;
    }

    for(i = 1; i < size; i++)  //先用面额小的，再用面额大的
    {
        for(j = 1; j <= value; j++)
        {
            if(j >= denomination[i])
                dp[j] = last[j] + dp[j-denomination[i]];
        }
        memcpy(last, dp, sizeof(int)*(value+1));
    }
    int chargeTimes = dp[value];

    delete[] last;
    delete[] dp;
    return chargeTimes;
}
```

总结与思考

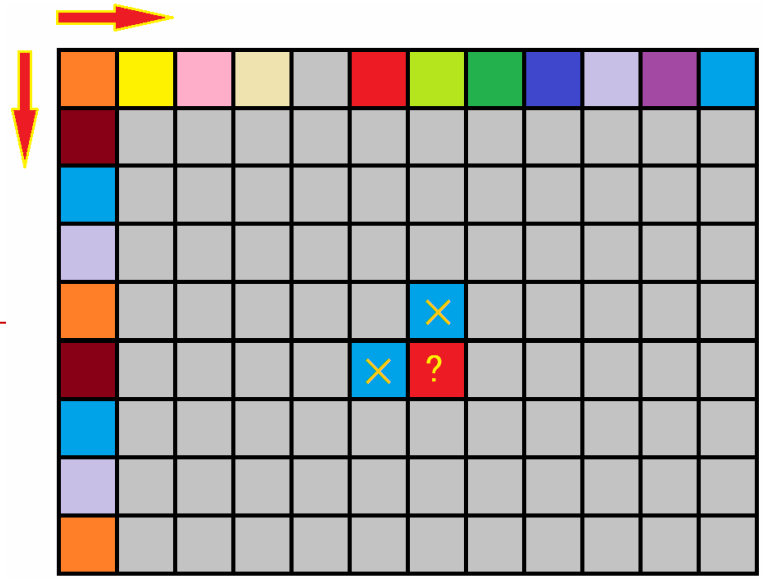
- 请问：本问题的时间复杂度是多少？
- 在动态规划的问题中，如果不求具体解的内容，而只是求解的数目，往往可以使用滚动数组的方式降低空间使用量（甚至空间复杂度）
  - 由于滚动数组减少了维度，甚至代码会更简单
- 思考0-1背包问题和格子取数问题。

走棋盘/格子取数

- 给定m×n的矩阵，每个位置是一个非负整数，从左上角开始，每次只能朝右和下走，走到右下角，求总和最小的路径。

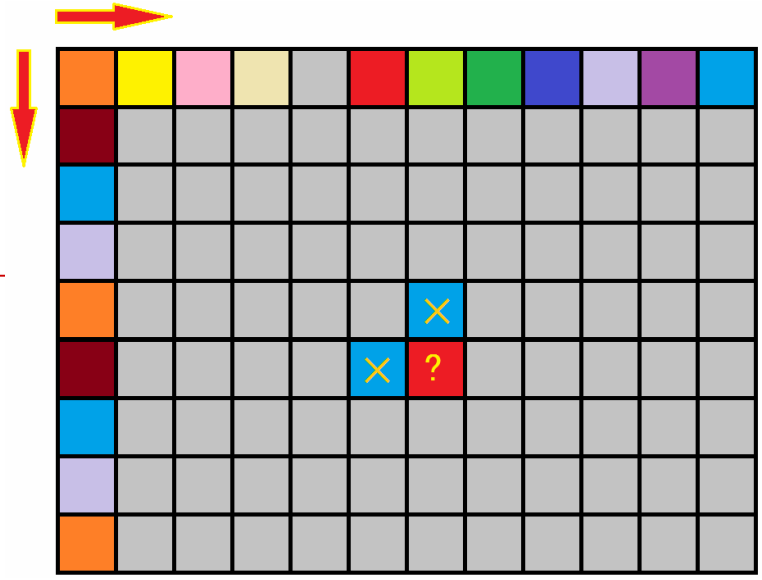


状态转移方程



- 走的方向决定了同一个格子不会经过两次。
  - 若当前位于(x,y)处，它来自于哪些格子呢？
  - dp[0,0]=a[0,0] / 第一行(列)累积
  - dp[x,y] = min(dp[x-1,y]+a[x,y],dp[x,y-1]+a[x,y])
  - 即：dp[x,y] = min(dp[x-1,y],dp[x,y-1]) +a[x,y]
- 思考：若将上述问题改成“求从左上到右下的最大路径”呢？

状态转移方程



- 状态转移方程：

$$\begin{cases} dp(i,0)=\sum_{k=0}^{i} chess[k][0] \\ dp(0,j)=\sum_{k=0}^{j} chess[0][k] \\ dp(i,j)=\min(dp(i-1,j),dp(i,j-1))+chess[i][j] \end{cases}$$

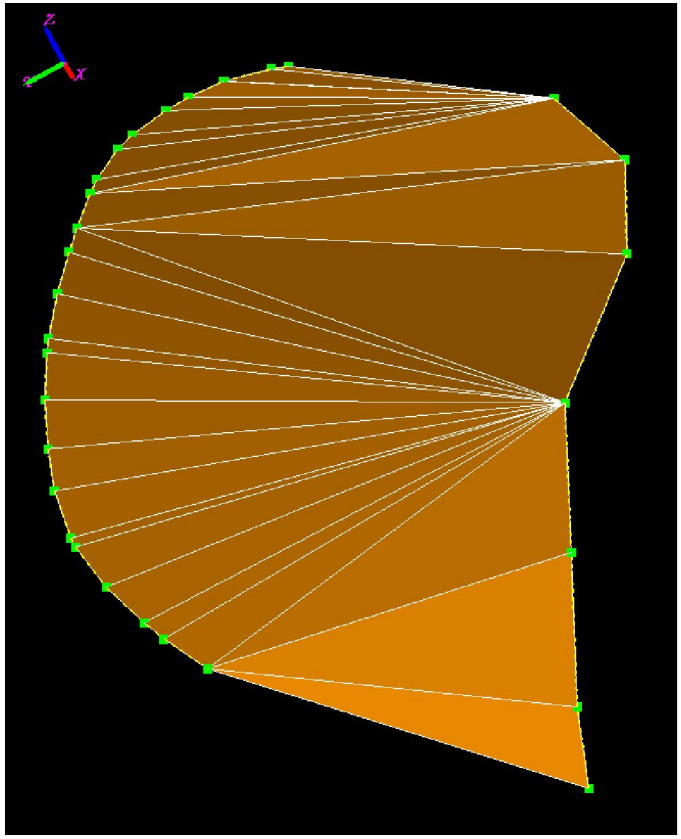
- 滚动数组：

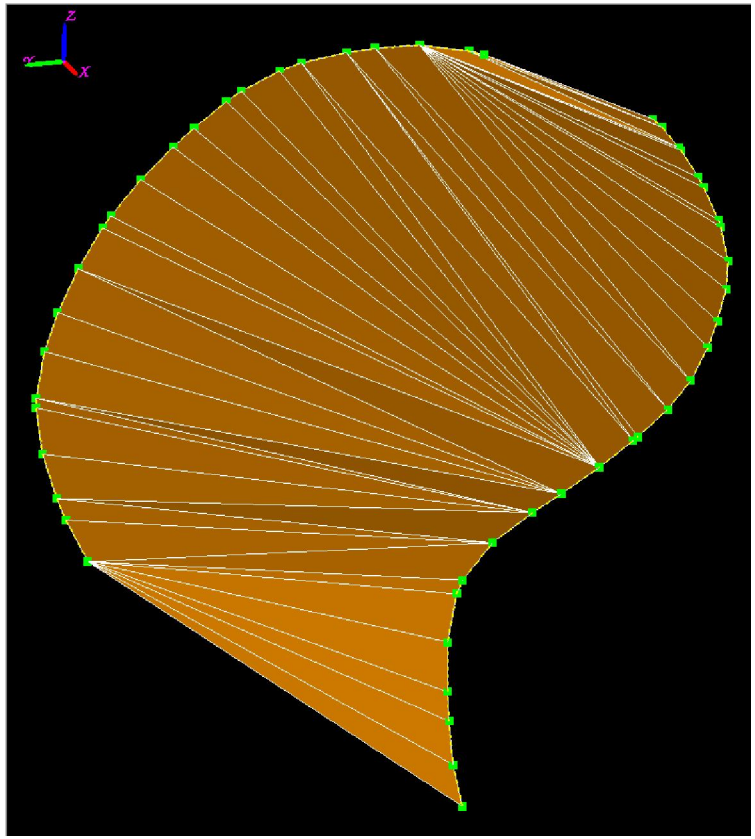
$$\begin{cases} dp(j)=\sum_{k=0}^{j} chess[0][k] \\ dp(j)=\min(dp(j),dp(j-1))+chess[i][j] \end{cases}$$

Code

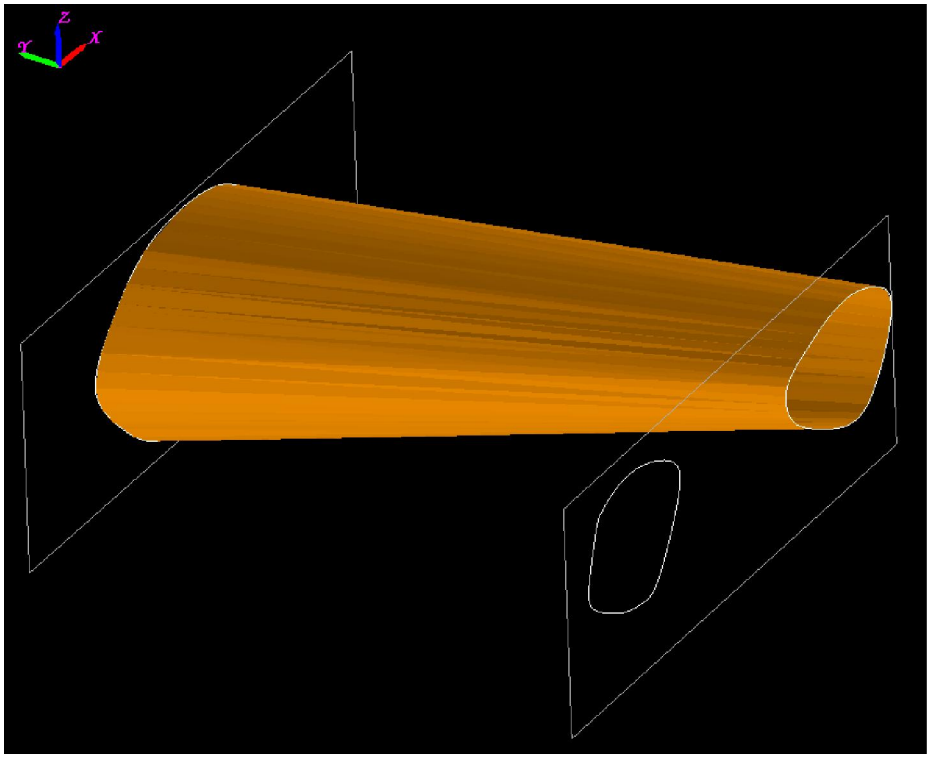
```
int MinPath(vector<vector<int> >& chess, int M, int N)
{
    vector<int> pathLength(N);
    int i,j;

    //初始化
    pathLength[0] = chess[0][0];
    for(j = 1; j < N; j++)
        pathLength[j] = pathLength[j-1] + chess[0][j];

    //依次计算每行
    for(i = 1; i < M; i++)
    {
        pathLength[0] += chess[i][0];
        for(j = 1; j < N; j++)
        {
            if(pathLength[j-1] < pathLength[j])
                pathLength[j] = pathLength[j-1] + chess[i][j];
            else
                pathLength[j] += chess[i][j];
        }
    }
    return pathLength[N-1];
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int M = 10;
    const int N = 8;
    vector<vector<int> > chess(M, vector<int>(N));

    //初始化棋盘：(随机给定)
    int i,j;
    for(i = 0; i < M; i++)
    {
        for(j = 0; j < N; j++)
            chess[i][j] = rand() % 100;
    }
    cout << MinPath(chess, M, N) << endl;
    return 0;
}
```

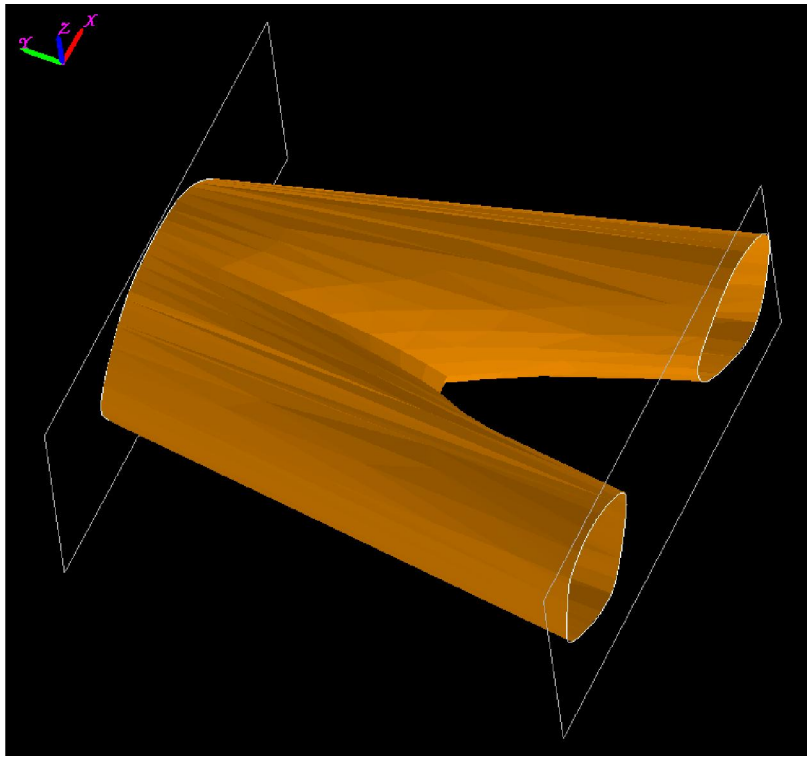
GIS中的应用





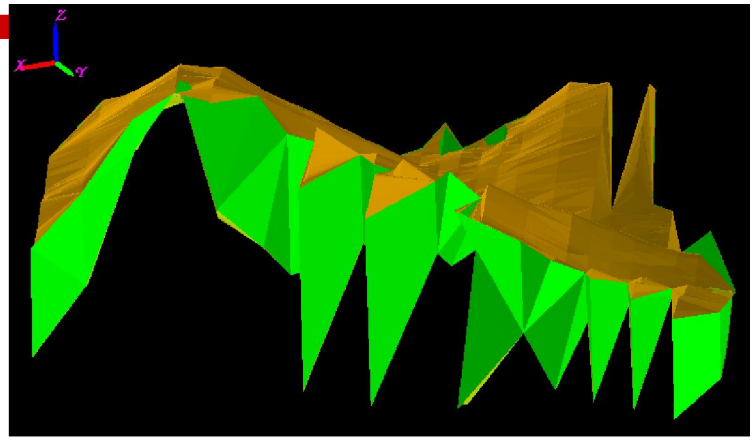
如果三维曲线是封闭线...

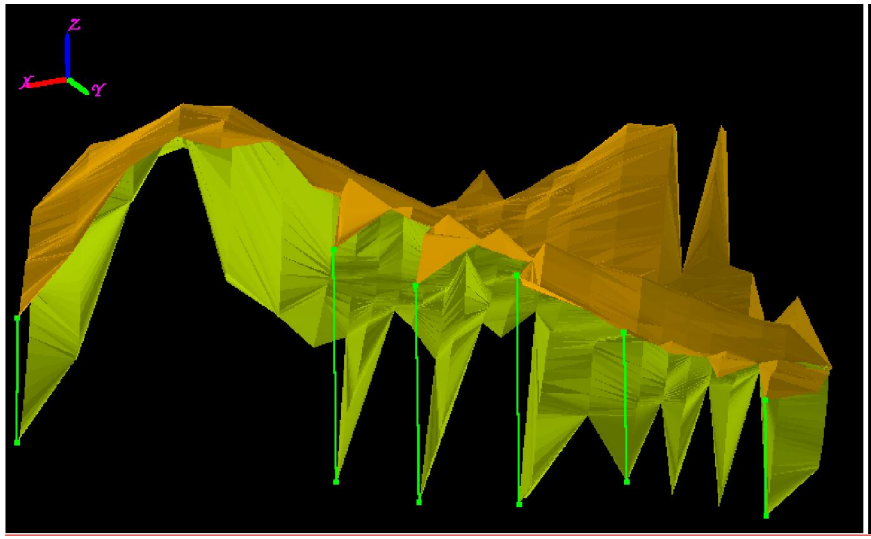


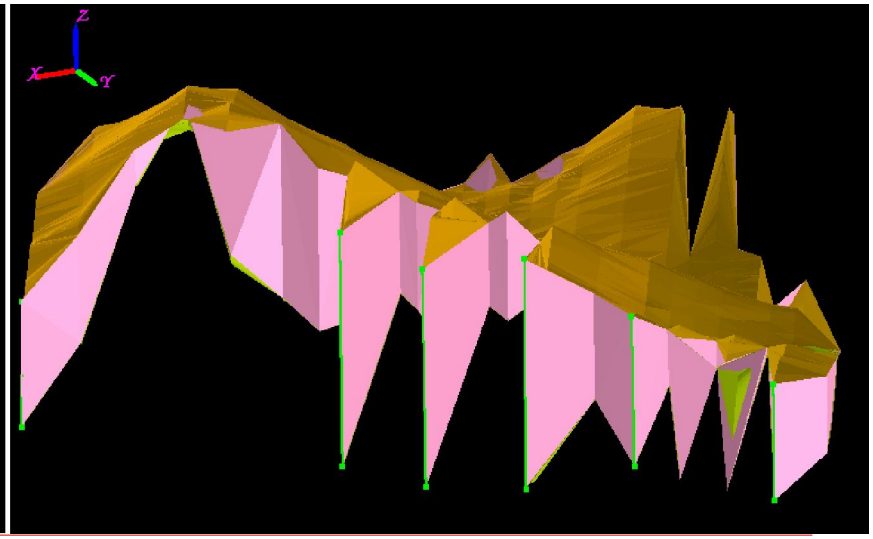


GIS中三维建模的实际应用

- 右上：未使用引导线
- 左下：输入的引导线
- 右下：过引导线的曲面







带陷阱的走棋盘

- 有一个n*m的棋盘网格，机器人最开始在左上角，机器人每一步只能往右或者往下移动。棋盘中有些格子是禁止机器人踏入的，该信息存放在二维数组blocked中，如果blocked[i][j]为true，那么机器人不能踏入格子(i,j)。请计算有多少条路径能让机器人从左上角移动到右下角。

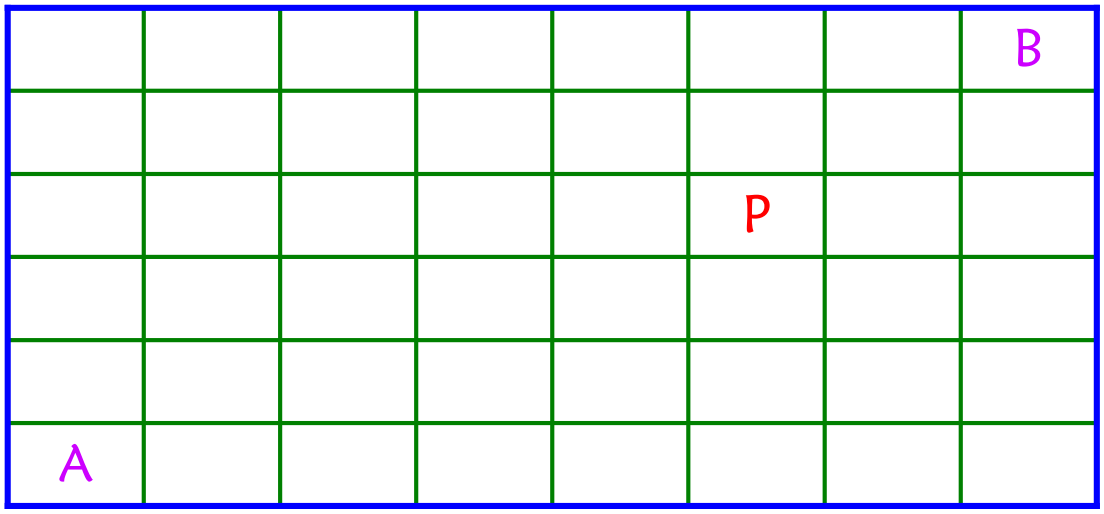
状态转移方程

- dp[i][j]表示从起点到(i,j)的路径条数。
- 只能从左边或者上边进入一个格子
- 如果(i,j)被占用
  - dp[i][j]=0
- 如果(i,j)不被占用
  - dp[i][j]=dp[i-1][j]+dp[i][j–1]

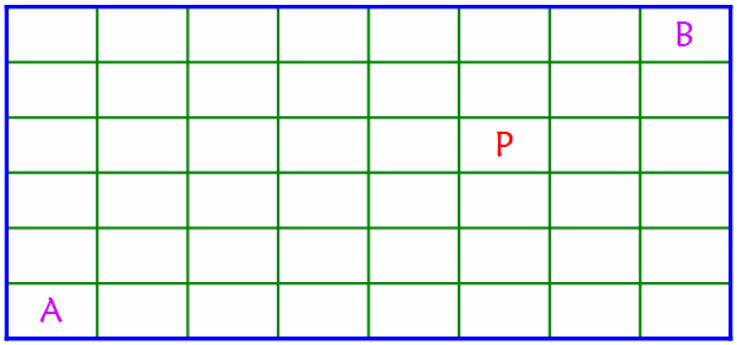
- 思考：如果没有占用的格子呢?
- 一共要走m+n–2步，其中(m–1)步向右，(n-1)步向下。组合数C(m+n–2,m-1)=C(m+n-2,n-1)

陷阱走棋盘

- 在8×6的矩阵中，每次只能向上或向右移动一格，并且不能经过P。试计算从A移动到B一共有多少种走法。

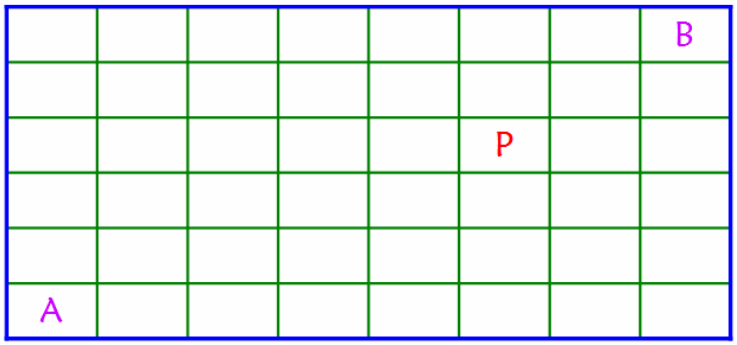


解题过程



- 从A到B共需要移动12步，其中7步向右，5步向上，可行走法数目为 $C_{12}^{5}=792$
- 从A到P共需要8步，其中5步向右，3步向上，可行走法数目为 $C_{8}^{5}=56$
- 从P到B共需要4步，其中2步向右，2步向上，可行走法数目为 $C_{4}^{2}=6$
- 则，从A到B经过P的路线有56*6=336种；
- 从A到B不经过P的路线有792-336=456种。

方格的可行路径数目



1	6	21	56	126	196	294	456
1	5	15	35	70	70	98	162
1	4	10	20	35	0	28	64
1	3	6	10	15	21	28	36
1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

思考题：两次走棋盘

- 给定m*n的矩阵，每个位置是一个非负的权值，从左上角开始，每次只能朝右和下走，走到右下角；然后，从右下角开始，每次只能朝左和朝上走，走到左上角。求权值总和最大的路径。若相同格子走过两次，则该位置的权值只算一次。

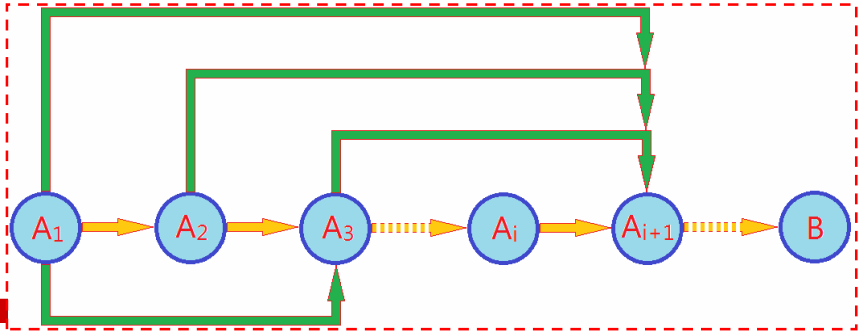
Code

```
const int N = 202;
const int inf = 1000000000;  //无穷大
int dp[N * 2][N][N];
bool IsValid(int step, int x1, int x2, int n)
{
    int y1 = step - x1, y2 = step - x2;
    return ((x1 >= 0) && (x1 < n) && (x2 >= 0) && (x2 < n) && (y1 >= 0) && (y1 < n) && (y2 >= 0) && (y2 < n));
}

int GetValue(int step, int x1, int x2, int n)
{
    return IsValid(step, x1, x2, n) ? dp[step][x1][x2] : (-inf);
}

//dp[step][i][j]表示在第step步两次分别在第i行和第j行的最大得分
int MinPathSum(int a[N][N], int n)
{
    int P = n * 2 - 2; //最终的步数
    int i, j, step;

    //不能到达的位置 设置为负无穷大
    for(i = 0; i < n; ++i)
    {
        for(j = i; j < n; ++j)
        {
            dp[0][i][j] = -inf;
        }
    }
    dp[0][0][0] = a[0][0];

    for(step = 1; step <= P; ++step)
    {
        for(i = 0; i < n; ++i)
        {
            for(j = i; j < n; ++j)
            {
                dp[step][i][j] = -inf;
                if(!IsValid(step, i, j, n))    //非法位置
                    continue;
                //对于合法的位置进行dp
                if(i != j)
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));
                    dp[step][i][j] += a[i][step - i] + a[j][step - j];  //不在同一个格子，加两个数
                }
                else
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j,   n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j,   n));
                    dp[step][i][j] += a[i][step - i]; // 在同一个格子里，只能加一次
                }
            }
        }
    }
    return dp[P][n - 1][n - 1];
}
```

Code split

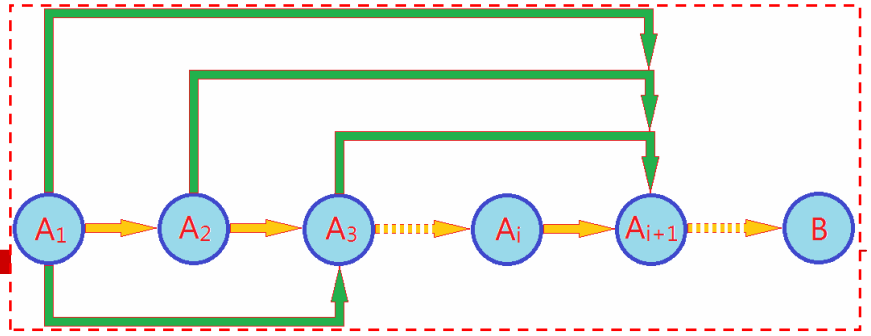
```
//dp[step][i][j]表示在第step步两次分别在第i行和第j行的最大得分
int MinPathSum(int a[N][N], int n)
{
    int P = n * 2 - 2; //最终的步数
    int i, j, step;

    //不能到达的位置 设置为负无穷大
    for(i = 0; i < n; ++i)
    {
        for(j = i; j < n; ++j)
        {
            dp[0][i][j] = -inf;
        }
    }
    dp[0][0][0] = a[0][0];

    for(step = 1; step <= P; ++step)
    {
        for(i = 0; i < n; ++i)
        {
            for(j = i; j < n; ++j)
            {
                dp[step][i][j] = -inf;
                if(!IsValid(step, i, j, n))    //非法位置
                    continue;
                //对于合法的位置进行dp
                if(i != j)
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));
                    dp[step][i][j] += a[i][step - i] + a[j][step - j];
                }
                else
                {
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j,   n));
                    dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j,   n));
                    dp[step][i][j] += a[i][step - i]; // 在同一个格子里，只能加一次
                }
            }
        }
    }
    return dp[P][n - 1][n - 1];
}
```

动态规划总结



- 动态规划是方法论，是解决一大类问题的通用思路。
  事实上，很多内容都可以归结为动态规划的思想。
  - KMP中求next数组：已知next[0…i-1]，求next[i]；
  - 最长回文子串Manacher算法中，已知P[0…i-1]求P[i]
- 何时可以考虑使用动态规划：
  - 初始规模下能够方便的得出结论
    - 空串、长度为0的数组、自身等
  - 能够得到问题规模增大导致的变化
    - 递推式——状态转移方程

动态规划总结



- 无后效性
  - 计算A[i]时只读取A[0…i-1]，不修改——历史
  - 计算A[i]时不需要A[i+1…n-1]的值——未来
- 在实践中往往忽略无后效性：
  - 问题本身决定了它是成立的：格子取数问题
  - 通过更改计算次序可以达到该要求：矩阵连乘问题
- 哪些题目不适合用动态规划？
  - 状态转移方程的推导，往往陷入局部而忽略全局。
    在重视动态规划的同时，别忘了从总体把握问题。

思考题：字符串的交替连接

- 输入三个字符串s1、s2和s3，判断第三个字符串s3是否由前两个字符串s1和s2交错而成，即不改变s1和s2中各个字符原有的相对顺序，例如当s1=“aabcc”，s2=“dbbca”，s3=“aadbbcbcac”时，则输出true，但如果s3=“accabdbbca”，则输出false。
- 换个表述：
  - s1和s2是s3的子序列，且s1∪s2=s3

Code

```
bool IsInterlace(const char* str1, const char* str2, const char* str)
{
    int M = (int)strlen(str1);
    int N = (int)strlen(str2);
    int S = (int)strlen(str);
    if(M+N != S)
        return false;
    vector<vector<bool> > p(M+1, vector<bool>(N+1));
    int i,j;
    p[0][0] = true;
    for(i = 1; i <= M; i++)      //首列
        p[i][0] = (p[i-1][0] && (str1[i-1] == str[i-1]));
    for(j = 1; j <= N; j++)      //首行
        p[0][j] = (p[0][j-1] && (str2[j-1] == str[j-1]));

    for(i = 1; i <= M; i++)
    {
        for(j = 1; j <= N; j++)
        {
            p[i][j] = (p[i-1][j] && (str[i+j-1] == str1[i-1]))
                || (p[i][j-1] && (str[i+j-1] == str2[j-1]));
        }
    }
    return p[M][N];
}
```

Code2

```
bool IsInterlace2(const char* str1, const char* str2, const char* str)
{
    int M = (int)strlen(str1);
    int N = (int)strlen(str2);
    int S = (int)strlen(str);
    if(M+N != S)
        return false;
    if(M < N)
        return IsInterlace2(str2, str1, str);
    vector<bool> p(N+1);
    int i,j;
    p[0] = true;
    for(j = 1; j <= N; j++)      //首行
        p[j] = (p[j-1] && (str2[j-1] == str[j-1]));

    for(i = 1; i <= M; i++)
    {
        p[0] = (p[0] && (str1[i-1] == str[i-1]));
        for(j = 1; j <= N; j++)
        {
            p[j] = (p[j] && (str[i+j-1] == str1[i-1]))
                || (p[j-1] && (str[i+j-1] == str2[j-1]));
        }
    }
    return p[N];
}
```

我们在这里

- 七月算法 http://www.julyedu.com/
  - 视频/课程/社区
- 七月题库APP：Android/iOS
  - http://www.julyapp.com/
- 微博
  - @研究者July
  - @七月题库
  - @邹博_机器学习
- 微信公众号
  - julyedu

感谢大家

恳请大家批评指正！