一文搞懂貪心算法

算法愛好者 5天前

（給算法愛好者加星標，修煉編程內功）

> 來源：獨酌逸醉
> www.cnblogs.com/chinazhangjie/archive/2010/11/23/1885330.html

顧名思義，貪心算法總是作出在當前看來最好的選擇。也就是說貪心算法並不從整體最優考慮，它所作出的選擇只是在某種意義上的局部最優選擇。當然，希望貪心算法得到的最終結果也是整體最優的。雖然貪心算法不能對所有問題都得到整體最優解，但對許多問題它能產生整體最優解。如單源最短路經問題，最小生成樹問題等。在一些情況下，即使貪心算法不能得到整體最優解，其最終結果卻是最優解的很好近似。

題一、活動安排問題

問題表述：設有n個活動的集合E = {1,2,…,n}，其中每個活動都要求使用同一資源，如演講會場等，而在同一時間內只有一個活動能使用這一資源。每個活i都有一個要求使用該資源的起始時間si和一個結束時間fi,且si < fi 。如果選擇了活動i，則它在半開時間區間[si, fi)內佔用資源。若區間[si, fi)與區間[sj, fj)不相交，則稱活動i與活動j是相容的。也就是說，當si >= fj或sj >= fi時，活動i與活動j相容。

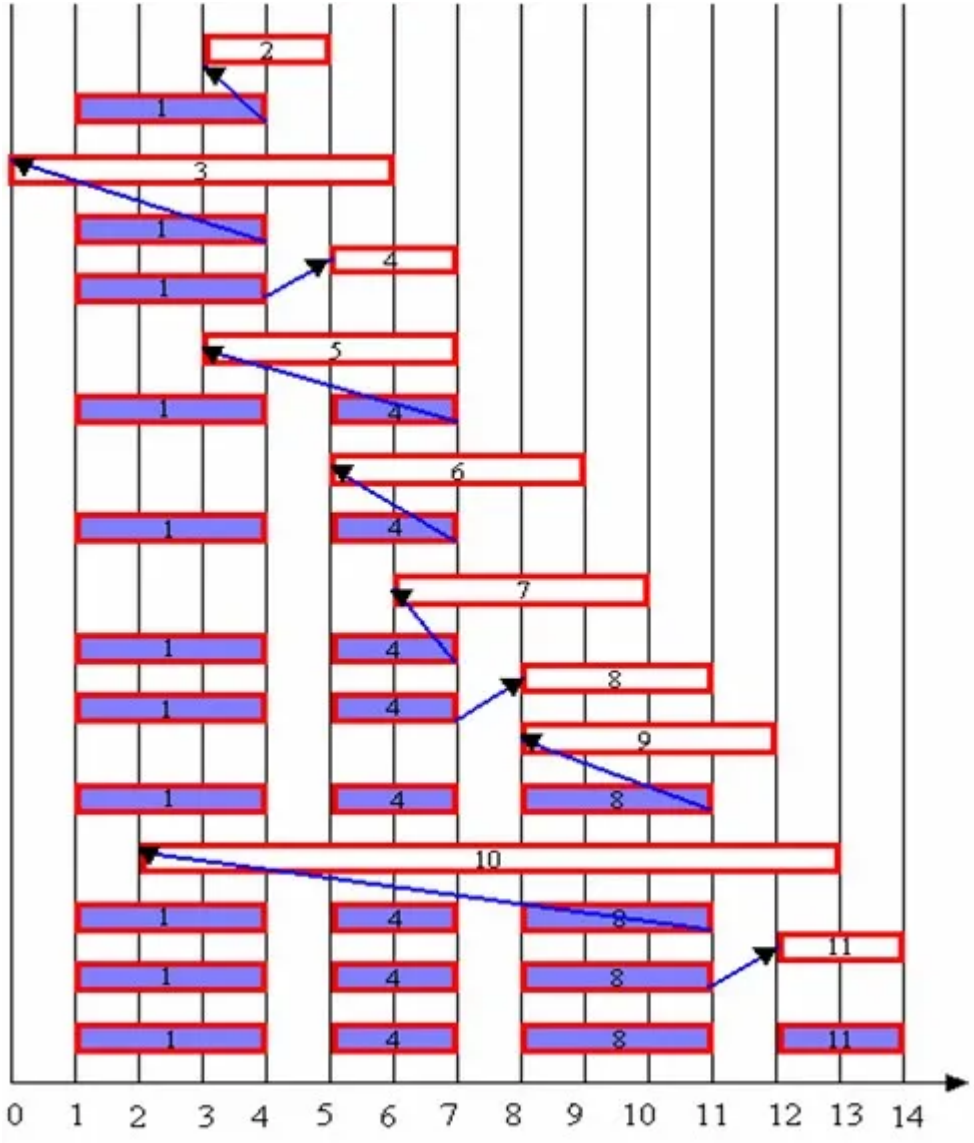
由於輸入的活動以其完成時間的非減序排列，所以算法greedySelector每次總是選擇具有最早完成時間的相容活動加入集合A中。直觀上，按這種方法選擇相容活動為未安排活動留下盡可能多的時間。也就是說，該算法的貪心選擇的意義是使剩餘的可安排時間段極大化，以便安排盡可能多的相容活動。

算法greedySelector的效率極高。當輸入的活動已按結束時間的非減序排列，算法只需O(n)的時間安排n個活動，使最多的活動能相容地使用公共資源。如果所給出的活動未按非減序排列，可以用O(nlogn)的時間重排。

例：設待安排的11個活動的開始時間和結束時間按結束時間的非減序排列如下：

i	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

算法greedySelector 的計算過程如下圖所示。圖中每行相應於算法的一次迭代。陰影長條表示的活動是已選入集合A的活動，而空白長條表示的活動是當前正在檢查相容性的活動。



若被檢查的活動i的開始時間Si小於最近選擇的活動j的結束時間fi，則不選擇活動i，否則選擇活動i加入集合A中。

貪心算法並不總能求得問題的整體最優解。但對於活動安排問題，貪心算法greedySelector卻總能求得的整體最優解，即它最終所確定的相容活動集合A的規模最大。這個結論可以用數學

歸納法證明。

活動安排問題實現：

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

struct ActivityTime
{
public:
    ActivityTime (int nStart, int nEnd)
        : m_nStart (nStart), m_nEnd (nEnd)
    { }
    ActivityTime ()
        : m_nStart (0), m_nEnd (0)
    { }
    friend
    bool operator < (const ActivityTime& lth, const ActivityTime& rth)
    {
        return lth.m_nEnd < lth.m_nEnd ;
    }
public:
    int m_nStart ;
    int m_nEnd ;
} ;

class ActivityArrange
{
public:
    ActivityArrange (const vector<ActivityTime>& vTimeList)
    {
        m_vTimeList = vTimeList ;
        m_nCount = vTimeList.size () ;
```

```
        m_bvSelectFlag.resize (m_nCount, false) ;
    }
    // 活动安排    void greedySelector ()
    {
        __sortTime () ;
        // 第一个活动一定入内        m_bvSelectFlag[0] = true ;
        int j = 0 ;
        for (int i = 1; i < m_nCount ; ++ i) {
            if (m_vTimeList[i].m_nStart > m_vTimeList[j].m_nEnd) {
                m_bvSelectFlag[i] = true ;
                j = i ;
            }
        }

        copy (m_bvSelectFlag.begin(), m_bvSelectFlag.end() ,ostream_iterator<bool> (cout, "
        cout << endl ;
    }

private:
    // 按照活动结束时间非递减排序    void __sortTime ()
    {
        sort (m_vTimeList.begin(), m_vTimeList.end()) ;
        for (vector<ActivityTime>::iterator ite = m_vTimeList.begin() ;
                ite != m_vTimeList.end() ;
                ++ ite) {
            cout << ite->m_nStart << ", "<< ite ->m_nEnd << endl ;
        }
    }

private:
    vector<ActivityTime>    m_vTimeList ;
    // 活动时间安排列表
    vector<bool>            m_bvSelectFlag ;
    // 是否安排活动标志
    int    m_nCount ;
```

```
    // 总活动个数
};

int main()
{
    vector<ActivityTime> vActiTimeList ;
    vActiTimeList.push_back (ActivityTime(1, 4)) ;
    vActiTimeList.push_back (ActivityTime(3, 5)) ;
    vActiTimeList.push_back (ActivityTime(0, 6)) ;
    vActiTimeList.push_back (ActivityTime(5, 7)) ;
    vActiTimeList.push_back (ActivityTime(3, 8)) ;
    vActiTimeList.push_back (ActivityTime(5, 9)) ;
    vActiTimeList.push_back (ActivityTime(6, 10)) ;
    vActiTimeList.push_back (ActivityTime(8, 11)) ;
    vActiTimeList.push_back (ActivityTime(8, 12)) ;
    vActiTimeList.push_back (ActivityTime(2, 13)) ;
    vActiTimeList.push_back (ActivityTime(12, 14)) ;

    ActivityArrange aa (vActiTimeList) ;
    aa.greedySelector () ;
    return 0 ;
}
```

貪心算法的基本要素

對於一個具體的問題，怎麼知道是否可用貪心算法解此問題，以及能否得到問題的最優解呢?這個問題很難給予肯定的回答。

但是，從許多可以用貪心算法求解的問題中看到這類問題一般具有2個重要的性質：貪心選擇性質和最優子結構性質。

1、貪心選擇性質

所謂貪心選擇性質是指所求問題的整體最優解可以通過一系列局部最優的選擇，即貪心選擇來達到。這是貪心算法可行的第一個基本要素，也是貪心算法與動態規划算法的主要區別。

動態規划算法通常以自底向上的方式解各子問題，而貪心算法則通常以自頂向下的方式進行，以迭代的方式作出相繼的貪心選擇，每作一次貪心選擇就將所求問題簡化為規模更小的子問題。

對於一個具體問題，要確定它是否具有貪心選擇性質，必須證明每一步所作的貪心選擇最終導致問題的整體最優解。

2、最優子結構性質

當一個問題的最優解包含其子問題的最優解時，稱此問題具有最優子結構性質。問題的最優子結構性質是該問題可用動態規划算法或貪心算法求解的關鍵特徵。

3、貪心算法與動態規划算法的差異

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是2类算法的一个共同点。但是，对于具有最优子结构的问题应该选用贪心算法还是动态规划算法求解?是否能用动态规划算法求解的问题也能用贪心算法求解?下面研究2个经典的组合优化问题，并以此说明贪心算法与动态规划算法的主要差别。

0-1背包问题：

给定n种物品和一个背包。物品i的重量是Wi，其价值为Vi，背包的容量为C。应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

在选择装入背包的物品时，对每种物品i只有2种选择，即装入背包或不装入背包。不能将物品i装入背包多次，也不能只装入部分的物品i。

背包问题：

与0-1背包问题类似，所不同的是在选择物品i装入背包时，可以选择物品i的一部分，而不一定要全部装入背包，1 <= i <= n。

这2类问题都具有最优子结构性质，极为相似，但背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解。

用贪心算法解背包问题的基本步骤：

首先计算每种物品单位重量的价值Vi/Wi，然后，依贪心选择策略，将尽可能多的单位重量价值最高的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未超过C，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去，直到背包装满为止。

伪代码：

```
void Knapsack(int n,float M,float v[],float w[],float x[])
{
  Sort(n,v,w);
  int i;
  for (i = 1 ; i <= n ; i++)
    x[i] = 0;
    float c=M;
    for (i=1;i<=n;i++) {
```

```
      if (w[i] > c) break;
    x[i]=1;
    c-=w[i];
  }
  if (i <= n)
    x[i]=c / w[i];
}
```

算法knapsack的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此，算法的计算时间上界为 O（nlogn）。

为了证明算法的正确性，还必须证明背包问题具有贪心选择性质。

对于0-1背包问题，贪心选择之所以不能得到最优解是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。事实上，在考虑0-1背包问题时，应比较选择该物品和不选择该物品所导致的最终方案，然后再作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用动态规划算法求解的另一重要特征。实际上也是如此，动态规划算法的确可以有效地解0-1背包问题。

问题二、 哈夫曼编码

哈夫曼编码是广泛地用于数据文件压缩的十分有效的编码方法。其压缩率通常在20%～90%之间。哈夫曼编码算法用字符在文件中出现的频率表来建立一个用0，1串表示各字符的最优表示方式。

给出现频率高的字符较短的编码，出现频率较低的字符以较长的编码，可以大大缩短总码长。

	a	b	c	d	e	f
频率（千次）	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

定长码：

3*(45+13+12+16+9+5) = 300 千位

变长码：

1*45*+*3*13+3*12*+*3*16+4*9*+*45* = 224 千位

1、前缀码

对每一个字符规定一个0,1串作为其代码，并要求任一字符的代码都不是其它字符代码的前缀。这种编码称为前缀码。

编码的前缀性质可以使译码方法非常简单。

表示最优前缀码的二叉树总是一棵完全二叉树，即树中任一结点都有2个儿子结点。

f(c)表示字符c出现的概率，dt(c)表示c的码长

平均码长定义为：

$$B(T)=\sum_{c\in C} f(c)d_T(c)$$

使平均码长达到最小的前缀码编码方案称为给定编码字符集C的最优前缀码。

2、构造哈夫曼编码

哈夫曼提出构造最优前缀码的贪心算法，由此产生的编码方案称为哈夫曼编码。

哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树T。

算法以|C|个叶结点开始，执行|C| - 1次的“合并”运算后产生最终所要求的树T。

以f为键值的优先队列Q用在贪心选择时有效地确定算法当前要合并的2棵具有最小频率的树。一旦2棵具有最小频率的树合并后，产生一棵新的树，其频率为合并的2棵树的频率之和，并将新树插入优先队列Q。经过n - 1次的合并后，优先队列中只剩下一棵树，即所要求的树T。

算法huffmanTree用最小堆实现优先队列Q。初始化优先队列需要O(n)计算时间，由于最小堆的removeMin和put运算均需O(logn)时间，n - 1次的合并总共需要O(nlogn)计算时间。因此，关于n个字符的哈夫曼算法的计算时间为O(nlogn) 。

3、哈夫曼算法的正确性

要证明哈夫曼算法的正确性，只要证明最优前缀码问题具有贪心选择性质和最优子结构性质。

(1)贪心选择性质

(2)最优子结构性质

实现：

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std ;


class HaffmanNode
{
public:
    HaffmanNode (int nKeyValue,
                 HaffmanNode* pLeft = NULL,
                 HaffmanNode* pRight = NULL)
    {
        m_nKeyValue = nKeyValue ;
        m_pLeft = pLeft ;
        m_pRight = pRight ;
    }

    friend
    bool operator < (const HaffmanNode& lth, const HaffmanNode& rth)
    {
        return lth.m_nKeyValue < rth.m_nKeyValue ;
    }

public:
    int        m_nKeyValue ;
```

```
    HaffmanNode*    m_pLeft ;
    HaffmanNode*    m_pRight ;
} ;

class HaffmanCoding
{
public:
    typedef priority_queue<HaffmanNode*> MinHeap ;
    typedef HaffmanNode*    HaffmanTree ;

public:
    HaffmanCoding (const vector<int>& weight)
        : m_pTree(NULL)
    {
        m_stCount = weight.size () ;
        for (size_t i = 0; i < weight.size() ; ++ i) {
            m_minheap.push (new HaffmanNode(weight[i], NULL, NULL)) ;
        }
    }
    ~ HaffmanCoding()
    {
        __destroy (m_pTree) ;
    }

    // 按照左1右0编码     void doHaffmanCoding ()
    {
        vector<int> vnCode(m_stCount-1) ;
        __constructTree () ;
        __traverse (m_pTree, 0, vnCode) ;
    }

private:
    void __destroy(HaffmanTree& ht)
    {
        if (ht->m_pLeft != NULL) {
            __destroy (ht->m_pLeft) ;
        }
```

```
        if (ht->m_pRight != NULL) {
            __destroy (ht->m_pRight) ;
        }
        if (ht->m_pLeft == NULL && ht->m_pRight == NULL) {
            // cout << "delete" << endl ;            delete ht ;
            ht = NULL ;
        }
    }
    void __traverse (HaffmanTree ht,int layers, vector<int>& vnCode)
    {
        if (ht->m_pLeft != NULL) {
            vnCode[layers] = 1 ;
            __traverse (ht->m_pLeft, ++ layers, vnCode) ;
            -- layers ;
        }
        if (ht->m_pRight != NULL) {
            vnCode[layers] = 0 ;
            __traverse (ht->m_pRight, ++ layers, vnCode) ;
            -- layers ;
        }
        if (ht->m_pLeft == NULL && ht->m_pRight == NULL) {
            cout << ht->m_nKeyValue << " coding:  " ;
            for (int i = 0; i < layers; ++ i) {
                cout << vnCode[i] << " " ;
            }
            cout << endl ;
        }
    }

    void __constructTree ()
    {
        size_t i = 1 ;
        while (i < m_stCount) {
            HaffmanNode* lchild = m_minheap.top () ;
```

```
            m_minheap.pop () ;
            HaffmanNode* rchild = m_minheap.top () ;
            m_minheap.pop () ;

            // 确保左子树的键值大于有子树的键值            if (lchild->m_nKeyValue < rchild->m_n
                HaffmanNode* temp = lchild ;
                lchild = rchild ;
                rchild = temp ;
            }
            // 构造新结点            HaffmanNode* pNewNode =
                new HaffmanNode (lchild->m_nKeyValue + rchild->m_nKeyValue,
                lchild, rchild ) ;
            m_minheap.push (pNewNode) ;
            ++ i ;
        }
        m_pTree = m_minheap.top () ;
        m_minheap.pop () ;
    }

private:
    vector<int> m_vnWeight ;    // 权值    HaffmanTree m_pTree ;
    MinHeap        m_minheap ;
    size_t        m_stCount ;        // 叶结点个数
} ;



int main()
{
    vector<int> vnWeight ;
    vnWeight.push_back (45) ;
    vnWeight.push_back (13) ;
    vnWeight.push_back (12) ;
    vnWeight.push_back (16) ;
    vnWeight.push_back (9) ;
    vnWeight.push_back (5) ;

    HaffmanCoding hc (vnWeight) ;
    hc.doHaffmanCoding () ;
```

```
    return 0 ;
}
```

问题三、单源最大路径

给定带权有向图G =(V,E)，其中每条边的权是非负实数。另外，还给定V中的一个顶点，称为源。现在要计算从源到所有其它各顶点的最短路长度。这里路的长度是指路上各边权之和。这个问题通常称为单源最短路径问题。

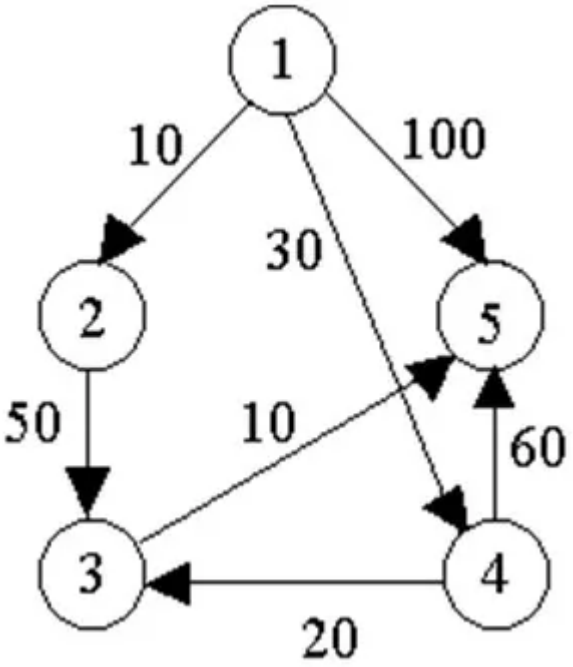
1、算法基本思想

Dijkstra算法是解单源最短路径问题的贪心算法。

其基本思想是，设置顶点集合S并不断地作贪心选择来扩充这个集合。一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知。

初始时，S中仅含有源。设u是G的某一个顶点，把从源到u且中间只经过S中顶点的路称为从源到u的特殊路径，并用数组dist记录当前每个顶点所对应的最短特殊路径长度。Dijkstra算法每次从V-S中取出具有最短特殊路长度的顶点u，将u添加到S中，同时对数组dist作必要的修改。一旦S包含了所有V中顶点，dist就记录了从源到所有其它顶点之间的最短路径长度。

例如，对下图中的有向图，应用Dijkstra算法计算从源顶点1到其它顶点间最短路径的过程列在下表中。



Dijkstra算法的迭代过程：

迭代	s	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	maxint	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

2、算法的正确性和计算复杂性

(1)贪心选择性质

(2)最优子结构性质

(3)计算复杂性

对于具有n个顶点和e条边的带权有向图，如果用带权邻接矩阵表示这个图，那么Dijkstra算法的主循环体需要O(n)时间。这个循环需要执行n-1次，所以完成循环需要O(n)时间。算法的其余部分所需要时间不超过O(n^2)。

实现：

```
#include <iostream>
#include <vector>
#include <limits>using namespace std ;

class BBShortestDijkstra
{
public:
    BBShortestDijkstra (const vector<vector<int> >& vnGraph)
        :m_cnMaxInt (numeric_limits<int>::max())
    {
        m_vnGraph = vnGraph ;
        m_stCount = vnGraph.size () ;
        m_vnDist.resize (m_stCount) ;
        for (size_t i = 0; i < m_stCount; ++ i) {
            m_vnDist[i].resize (m_stCount) ;
        }
    }

    void doDijkatra ()
{
        int nMinIndex = 0 ;
        int nMinValue = m_cnMaxInt ;
        vector<bool> vbFlag (m_stCount, false) ;
        for (size_t i = 0; i < m_stCount; ++ i) {
            m_vnDist[0][i] = m_vnGraph[0][i] ;
            if (nMinValue > m_vnGraph[0][i]) {
                nMinValue = m_vnGraph[0][i] ;
                nMinIndex = i ;
            }
        }
```

```
        vbFlag[0] = true ;
        size_t k = 1 ;
        while (k < m_stCount) {
            vbFlag[nMinIndex] = true ;
            for (size_t j = 0; j < m_stCount ; ++ j) {
                // 没有被选择                if (!vbFlag[j] && m_vnGraph[nMinIndex][j] != m_c
                    if (m_vnGraph[nMinIndex][j] + nMinValue
                        < m_vnDist[k-1][j]) {
                        m_vnDist[k][j] = m_vnGraph[nMinIndex][j] + nMinValue ;
                    }
                    else {
                        m_vnDist[k][j] = m_vnDist[k-1][j] ;
                    }
                }
                else {
                    m_vnDist[k][j] = m_vnDist[k-1][j] ;
                }
            }
            nMinValue = m_cnMaxInt ;
            for (size_t j = 0; j < m_stCount; ++ j) {
                if (!vbFlag[j] && (nMinValue > m_vnDist[k][j])) {
                    nMinValue = m_vnDist[k][j] ;
                    nMinIndex = j ;
                }
            }
            ++ k ;
        }

        for (int i = 0; i < m_stCount; ++ i) {
            for (int j = 0; j < m_stCount; ++ j) {
                if (m_vnDist[i][j] == m_cnMaxInt) {
                    cout << "maxint " ;
                }
```

```
                else {
                    cout << m_vnDist[i][j] << " " ;
                }
            }
            cout << endl ;
        }
    }
private:
    vector<vector<int> >    m_vnGraph ;
    vector<vector<int> >    m_vnDist ;
    size_t m_stCount ;
    const int m_cnMaxInt ;
} ;

int main()
{
    const int cnCount = 5 ;
    vector<vector<int> > vnGraph (cnCount) ;
    for (int i = 0; i < cnCount; ++ i) {
        vnGraph[i].resize (cnCount, numeric_limits<int>::max()) ;
    }
    vnGraph[0][1] = 10 ;
    vnGraph[0][3] = 30 ;
    vnGraph[0][4] = 100 ;
    vnGraph[1][2] = 50 ;
    vnGraph[2][4] = 10 ;
    vnGraph[3][2] = 20 ;
    vnGraph[3][4] = 60 ;

    BBShortestDijkstra bbs (vnGraph) ;
    bbs.doDijkatra () ;
}
```

- EOF -

推荐阅读 点击标题可跳转

1、人民日报：数学到底有多重要？网友：道理都懂，实力不允许啊...

2、三种洗牌算法简介

3、运用贪心算法来做时间管理

觉得本文有帮助？请分享给更多人

推荐关注「算法爱好者」，修炼编程内功

点赞和在看就是最大的支持♡

喜歡此內容的人還喜歡

面試官：緩存一致性問題怎麼解決？| 文末送書

艾小仙

史上最便捷搭建Zookeeper服務器的方法

ImportNew

10 天5 千Star！21 歲本科生給程序員開發的十六進制編輯器

開源前哨