A\*算法（A-Star）  
  
计算机可以通过一个“估价函数”确定一个状态的“前途”的

估价函数的定义: f^(n)=g^(n)+h^(n)

f^ 读作f-hat  
n是状态的表示，通常是状态的编号之类的。

在编程中，可以写作f\_hat()或者直接写成f()即可。  
  
g^(n)表示从初始状态到n 总共花费的代价。  
h^(n)表示从n到目标状态估计需要花费的代价，

对于一个正确的搜索，h^(n)<=h(n)，h(n)表示从n到目标状态的实际代价（没有算出来的时候，当然是求不出来h(n)的，但是可以找到一个h^(n)<=h(n)哦），在这个范围内，h^(n)越接近h(n)，搜索的启发效果越好。  
对于h^1(n)和h^2(n)，如果存在h^1(n)<h^2(n)<=h(n)，且h^1(n)<=h^2(n)，我们说h^2(n)比h^1(n)更灵通（more informed）。  
  
f^(n)的值就是对n这个状态的估价。这个估价主要体现在h^()，因为g^()是已知的；g^()体现了广度优先，当h^()>g^()时，可以省略g^()从而提高效率。  
  
搜索中，每到新的状态，计算它的f^()值，在目前所有的状态中，优先扩展f^()最小的，如果f^()最小的中有很多，则优先扩展其中g^()最大的。

IDA\*算法（迭代加深的A\*算法）

IDA\*的基本步骤：  
1、设置最大的深度maxf = f^(初始状态)  
2、IDS搜索，弃去f^()>maxf的情况，如果找到答案，则结束  
3、如果没有答案：若搜索中出现了比maxf更大的f^()，则令maxf = 其中最小的值，重复2；如果没有，则说明没有答案

# 字符串处理

KMP算法

Knuth-Morris-Pratt 字符串查找 O(n+m)

Donald Knuth、Vaughan Pratt、James H. Morris三人于1977年联合发表

匹配失败 string下标不变，回滚pattern下标到next[j]位置

Next[j]最长前缀后缀 abcXXXabc

int KmpSearch(char\* s, char\* p)

{

int i = 0;

int j = 0;

int sLen = strlen(s);

int pLen = strlen(p);

while (i < sLen && j < pLen)

{

//如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++

if (j == -1 || s[i] == p[j]) { i++; j++; }

//如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]

else j = next[j];

}

if (j == pLen)

return i - j;

else return -1;

}

void GetNext(char\* p, int next[])

{

int pLen = strlen(p);

next[0] = -1;

int j = -1;

int i = 0;

while (i < pLen - 1)

{

//p[j]表示前缀，p[i]表示后缀

if (j == -1 || p[i] == p[j]) next[++i] = ++j;

else j = next[j];

}

}

Next 数组优化

不该出现 p[j] = p[ next[j] ]。为什么呢？理由是：当 p[j] !=s[i] 时，下次匹配必然是 p[ next [j]] 跟 s[i]匹配，如果 p[j]=p[next[j]]，必然导致后一步匹配失败（因为 p[j]已经跟 s[i]失配，然后你还用跟 p[j] 等同的值 p[next[j]]去跟 s[i]匹配，很显然，必然失配），所以不能允许 p[j] = p[ next[j ]]。 但如果出现了 p[j] = p[ next[j] ] 咋办呢？如果出现了，则需要再次递归，即令 next[j] = next[ next[j] ]。 所以，咱们得修改下求 next 数组的代码。

//优化过后的next 数组求法

void GetNextval(char\* p, int next[])

{

int pLen = strlen(p);

next[0] = -1;

int j = -1;

int i = 0;

while (i < pLen - 1)

{

//p[j]表示前缀，p[i]表示后缀

if (j == -1 || p[i] == p[j])

{

++i; ++j;

//因为不能出现p[i] = p[ next[i ]]，

//所以当出现时需要继续递归，j = next[j] = next[next[j]]

//较之前next数组求法，改动在下面4行

if (p[i] != p[j]) next[i] = j; //之前只有这一行

else next[i] = next[j];

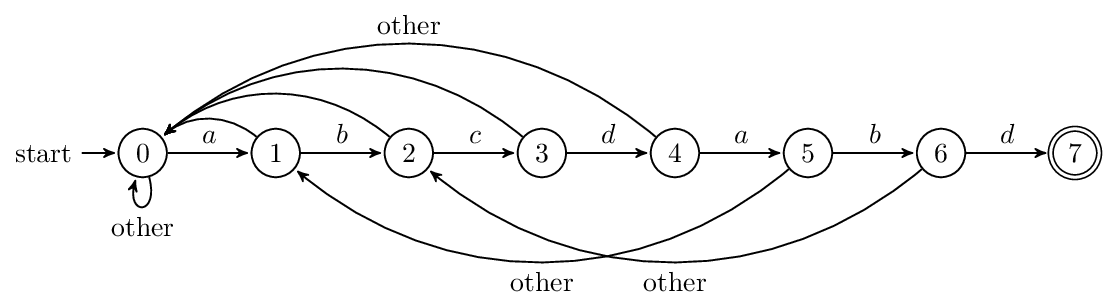
}

else j = next[j];

}

}

KMP的next数组 为 DFA有限状态自动机



BM算法

Boyer-Moore算法 1977年提出 平均为O(n) 快于 KMP

[亚线性](http://baike.baidu.com/item/%E4%BA%9A%E7%BA%BF%E6%80%A7)串匹配算法，它在最坏情况下找到模式所有出现的时间复杂度为O(m\*n)，

在最好情况下执行匹配找到模式所有出现的时间复杂度为O(n/m)

主要思想描述如下

(1)模式字符串的匹配顺序是从右向左:

(a)首先将P和T对齐,即p[1]和t[1]对齐；

(b)然后匹配从模式字符串P的最右端字符开始，即判断 p[m]和t[m]是否匹配：

如果匹配成功,则向左移动判断p[m-1]和t[m-1]是否匹配，

如此循环下去；如果匹配不成功，则进行字符串滑移。

(2)字符串滑移启发式策略：

(a)坏字符移动启发式策略

(b)好后缀移动启发式策略

BM算法定义了两种规则：

　　注：为方面书写，S称作母串，T称作模板串

**A B C D E F G H I**

**↨**

**C D E F E F**

　　　　如上图所示，蓝色字符串“EF”就是好后缀 绿色字符“D”就是坏字符。

1）坏字符规则

在BM算法从右向左扫描的过程中，若发现某个字符x不匹配，则按如下两种情况讨论：I.如果字符x在T中没有出现，那么从字符x开始的m个文本显然不可能与S在此处的字符匹配成功，直接全部跳过该区域即可。

II.如果x在T中出现，选择最右该字符进行对齐。

2）好后缀规则

若发现某个字符不匹配的同时，已有部分字符匹配成功，则按如下两种情况讨论：

I.如果在T中其他位置存在与好后缀相同的子串，选择最边右的子串，将S左移使该子

　　　　串与好后缀对齐（相当于T右移）。

II.如果在T中任何位置不存在与好后缀相同的字串，查找是T中否存在某一前缀与好后

　　　　缀相匹配，如果有选择最长前缀与S对齐，相当于S左移或者T右移；如果不存在，那么直接跳过该后缀，T的首字符与S好后缀的下一字符对齐。

　　　　坏字符规则图文解释：

 　　　　(1)

**A  B  C  D  E  F  G H**

**↨**

**H  I   J  K**

　　　　　　不匹配，直接跳过，得到：

**A  B  C  D  E  F  G  H**

**↨**

**H  I   J  K**

　　　　(2)

**A B C D E F G**

　　　　　　　　　  ↨

**A D D F**

　　　　　　字符"D"在T中存在，那么得到：

**A  B C D E F G**

**↨**

**A D D F**

　　　　好后缀规则图文解释：

　　　　 (1)

**A B C D E F G H I J K**

**↨**

**A E F A E F**

　　　　　　字符"A"与"D"不匹配，好后缀"EF"在T中存在，那么得到：

**A B C D E F G H I J K**

　　　　　　　　　　　　　　　↨

**A E F A E F**

　　　　(2)

**A B C D E F G H I J K L**

**↨**

**F G F A E F G**

　　　　　　T中存在前缀“FG”与后缀“FG”相匹配，那么得到：

**A B C D E F G H I J K L**

**↨**

**F G F A E F G**

　　　　如果是这样：

**A B C D E F G H I J K L M N**

**↨**

**F A F A E F G**

　　　　　　不存在前缀与任一后缀匹配，那么得到：

**A B C D E F G H I J K L M N**

　　　　　　　　　　　　　　　　　　　　 ↨

**F A F A E F G**

**Sunday算法**

Daniel M.Sunday于1990年提出， 比BM更快

平均性能的为O(n)，最差情况的时间复杂度为O(n \* m)

匹配原理

从前往后匹配，在匹配失败时关注的是string中参加匹配的最末位字符的下一位字符。

如果该字符没有在pattern中出现则直接跳过，即移动位数 = 匹配串长度 + 1；

否则，其移动位数 = pattern中最右端的该字符到末尾的距离+1。

母串S： S  E  A  R  C  H  S  U  B  S  T  R  I  N  G

 模板串T：　S  U  B  S  T  R  I  N  G

 开始匹配：

**S  E  A  R  C  H  S  U  B  S  T  R  I  N  G**

**↨**

**S  U  B  S  T  R  I  N  G**

继续下一字符匹配：

**S  E  A  R  C  H  S  U  B  S  T  R  I  N  G**

**↨**

**S  U  B  S  T  R  I  N  G**

出现不匹配情况，查找母串参与匹配的最后一位字符的下一字符，上图中S中最后一位参与

　　匹配的字符是颜色为蓝色的字符’B’，其下一字符为’S’，在T中，字符’S’出现两次，按照原理，

　　选择最右位置出现的’S’进行对齐，那么可以得到：

**S  E  A  R  C  H  S  U  B  S  T  R  I  N  G**

**↨**

**S  U  B  S  T  R  I  N  G**

　　　　直接跳过大片区域。

　　　　假设母串S为：

　　　　　　S  E  A  R  C  H  S  U  B  Z  T  R  I  N  G

　　　　那么当匹配到上述情况时，字符’Z’在T中没有出现，那么就可以得到下面的情况：

**S  E  A  R  C  H  S  U  B  Z  T  R  I  N  G**

**↨**

**S  U  B  S  T  R  I  N  G**

**图基本算法**

DFS 对边（u->v ）  
color[v] == 0 白色 **树边**；v是首次被发现  
color[v] == 1 灰色 **后向边**；v已经被发现访问，但其子孙还没有被访问完(访问中)

u就是v的子孙后代，因此后向边又称返祖边（**存在环**）  
color[v] == 2 黑色 **前向边**或者**横向边**；v以及其子孙后代也已经全部访问完

无向图 只存在树边和后向边

**环判断之无向图**

方法1： DFS找后向边

方法2：如果存在回路，则环路中所有顶点的度>=2。

算法：

第一步：删除所有度<=1的顶点及相关的边，并将另外与这些边相关的其它顶点的度减一。

第二步：将度数变为1的顶点排入队列，并从该队列中取出一个顶点重复步骤一。

     如果最后还有未删除顶点，则存在环，否则没有环。

**环判断之有向图**

方法1： DFS找灰色点及后向边

方法2： 拓扑排序 存在环无法完成拓扑排序

拓扑排序 可以用改进DFS和Kahn算法

1）由AOV网构造拓扑序列 主要是循环执行以下两步，直到不存在入度为0的顶点为止。

(1) 选择一个入度为0的顶点并输出之

(2) 从网中删除此顶点及所有[出边](http://baike.baidu.com/item/%E5%87%BA%E8%BE%B9)

循环结束后，若输出的顶点数小于网中的顶点数，则输出“有[回路](http://baike.baidu.com/item/%E5%9B%9E%E8%B7%AF/35792)”信息，否则输出的顶点序列就是一种拓扑[序列](http://baike.baidu.com/item/%E5%BA%8F%E5%88%97/1302588)

2）DFS reverse post-order 可以得到 拓扑排序

* Pre-Order，在递归调用dfs之前将当前顶点添加到queue中
* Reverse Pre-Order，在递归调用dfs之前将当前顶点添加到stack中
* Post-Order，在递归调用dfs之后将当前顶点添加到queue中
* Reverse Post-Order，在递归调用dfs之后将当前顶点添加到stack中

强连通分量

无向图：for\_each(V) 做 DFS/BFS，访问到的点都为连通分量

有向图：Kosaraju 算法

STRONGLY-CONNECTED-COMPONENTS.G

1 call DFS.G(Reverse Post-Order) for each vertex u -> stored in S(stack)

2 计算 G^T(G的reverse graph)

3 call DFS.G^T for vertex u pop from S(stack)

4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

**最小生成树**

Minimum Spanning Tree（最小权重生成树）：边子集所构成的树中，不但包括了连通图里的所有顶点，且其所有边的权值之和亦为最小。

贪心算法：Kruskal, Prim

使用binary-heap O(ElgV).

使用Fibonacci heaps, Prim O(E+VlgV)。

运行在|V|远远小于|E|时，相对binary-heap改进很大。

**Kruskal算法**

由Joseph Kruskal在1956年发表。 在连接森林中两颗不同树的边里面，找到权重最小的，加入集合。

算法描述

1).记Graph(V,E)

2).新建图Graph\_new，Graph\_new中拥有原图中相同的V个顶点，但没有边

3).将原图Graph中所有E个边按权值从小到大排序（可以建造最小堆）

4).循环：从权值最小的边开始遍历每条边 直至图Graph中所有的节点都在同一个连通分量中

   if 这条边连接的两个节点于图Graph\_new中不在同一个连通分量中(并查集判断)

              添加这条边到图Graph\_new中



**Prim算法**

普里姆算法1957年由美国Robert C. Prim发现；在连接到最小生产树中的边里面，找到权重最小边，加入集合。

**算法简单描述**

1).输入：一个加权连通图，Graph(V,E).

2).初始化：V\_new = {x}，其中x为集合V中的任一节点（起始点），E\_new= {},

3).重复下列操作，直到V\_new = V：

a.在集合E中选取权值最小的边<u, v>，其中u为集合V\_new中的元素，而v不在V\_new集合当中。

b.将v加入集合V\_new中，将<u, v>边加入集合E\_new中；

4).输出：使用集合V\_new和E\_new来描述所得到的最小生成树。



**最短路径**

**单源最短路径**

松弛操作





**Bellman-Ford算法**

可以存在权重为负的边，不能包含权重为负的环路。

O(VE)



**Dijkstra算法**

所有边的权重都为非负

简单实现 O(V^2) 普通二叉堆O((V+E)lgV) 斐波那契堆 O(E+VlgV)



**所有节点对的最短路径**

**Floyd-Warshall算法**

可以存在权重为负的边，不能包含权重为负的环路。时间复杂度V^3 空间复杂度V^2

动态规划，二维矩阵递推。

分别以k = 1 to n 假设做中间结点





**RMQ算法**

ST（Sparse Table）算法是一个非常有名的在线处理RMQ问题的算法。

它可以在O(nlogn)时间内进行预处理，然后在O(1)时间内回答每个查询。

F[i,j]表示从i开始连续2^j 个数 即[i, i+2^j-1]这个区间内的极值，

F[]的求法可以用[动态规划](http://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92/529408)，**F[i, j] = min(F[i, j-1], F[i+2^(j-1), j-1])**

查询**RMQ(i,j) = min(F[i,k], F[j-2^k+1,k])**，k=log2(j-i+1) = ln(j-i+1)/ln(2)

其中k是满足2^k<=b-a+1(即长度)的最大的k,即。

INIT\_RMQ

for i : 1 to n

max[i][0] = num[i]

for j : 1 to int(log(n)/log(2))

for i : 1 to (n+1-2^i) // for(int i=1; i+(1<<i)-1)<=n; i++)

max[i][j] = MAX（max[i][j-1], max[i+2^(i-1)][j-1]）

RMQ(i, j)

k = log(j-i+1) / log(2)

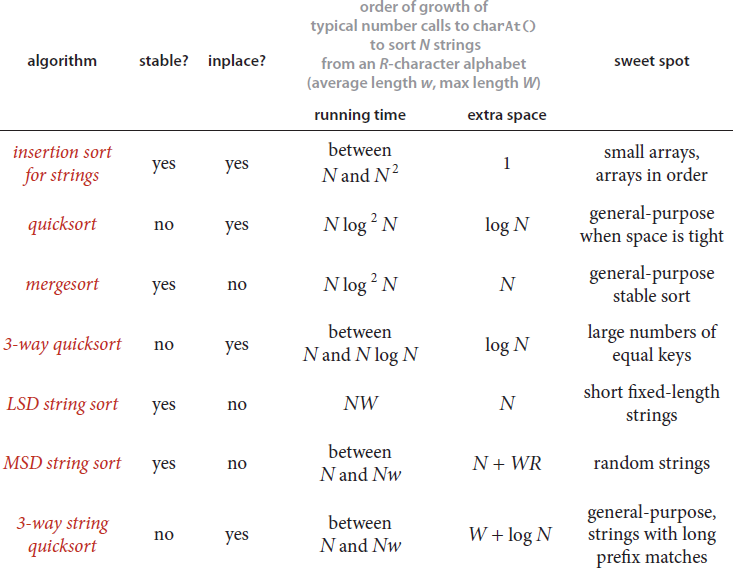
return MAX(max[i][k], max[j-2^k+1][k])

**P vs NPC**

Shortest vs. longest simple paths

Euler tour(each edge once) vs. hamiltonian cycle

2-CNF satisfiability vs. 3-CNF satisfiability



**Theorem**：定理。是文章中重要的数学化的论述，一般有严格的数学证明。

**Proposition**：可以翻译为命题，经过证明且interesting，但没有Theorem重要，比较常用。

**Lemma**：一种比较小的定理，通常lemma的提出是为了来逐步辅助证明Theorem，有时候可以将Theorem拆分成多个小的Lemma来逐步证明，以使得证明的思路更加清晰。

**Corollary**：推论，由Theorem推出来的结论，通常我们会直接说this is a corollary of Theorem A。

**Property**：性质，结果值得一记，但是没有Theorem深刻。

**Claim**：陈述，先论述然后会在后面进行论证，可以看作非正式的lemma。

**Note**：就是注解。

**Remark**：涉及到一些结论，相对而言，Note像是说明，而Remark则是非正式的定理。

**Conjecture**：猜测。一个未经证明的论述，但是被认为是真。

**Axiom/Postulate**：公理。不需要证明的论述，是所有其他Theorem的基础。