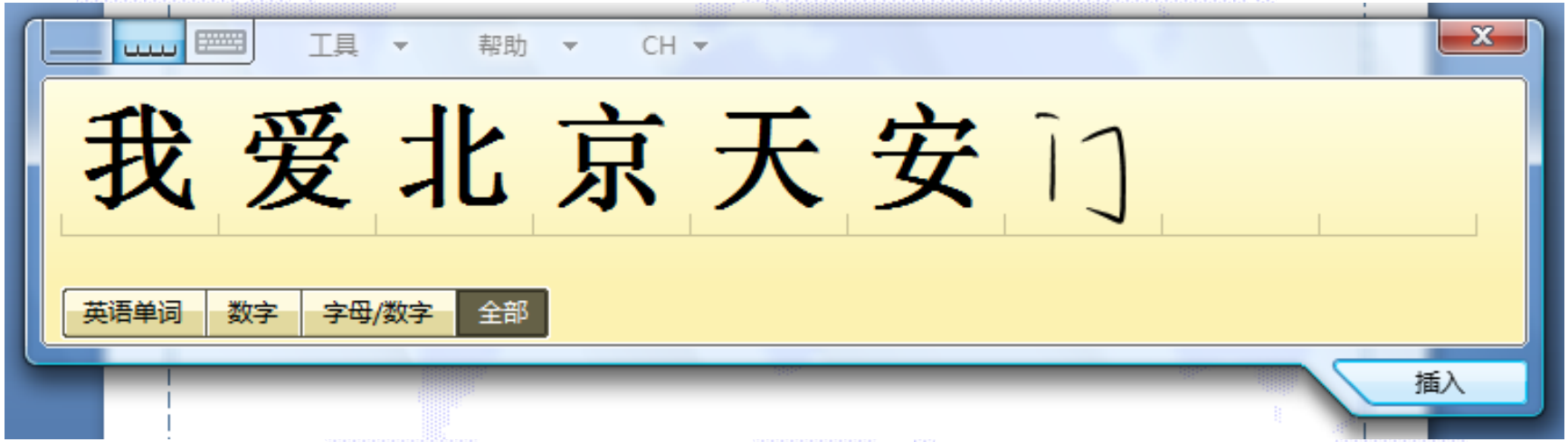
# 超越计算极限
# 人工智能搜索

JIANG, Yanyan
Nanjing University

# 计算机具有了何种“智能”？

- 模式识别。无论是人像、笔迹、指纹还是语音，都有对应的技术。



# 计算机具有了何种“智能”？(cont.)

- 形式化推理（机器证明）。
- 1977年中国科学院的平面几何机械化证明首次取得成功。
- 传说中FMSong带去的若干难题被瞬间解出。

- 还有很多……

# 强人工智能与弱人工智能

- 强人工智能观点认为有可能制造出真正能推理（Reasoning）和解决问题（Problem_solving）的智能机器，并且，这样的机器能将被认为是有知觉的，有自我意识的。

- 弱人工智能观点认为不可能制造出能真正地推理（Reasoning）和解决问题（Problem_solving）的智能机器，这些机器只不过看起来像是智能的，但是并不真正拥有智能，也不会有自主意识。

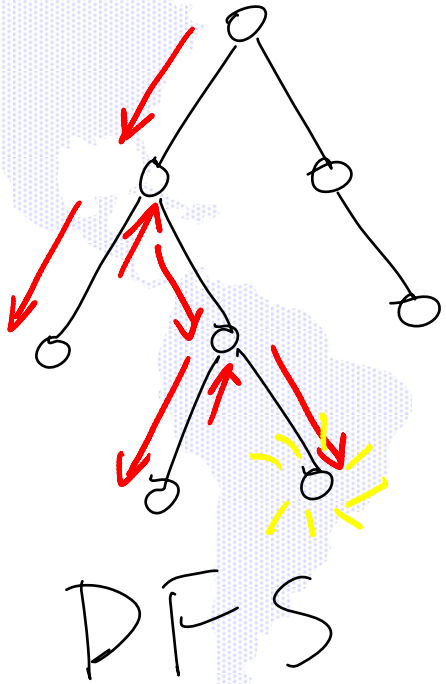
# 今天的主要内容

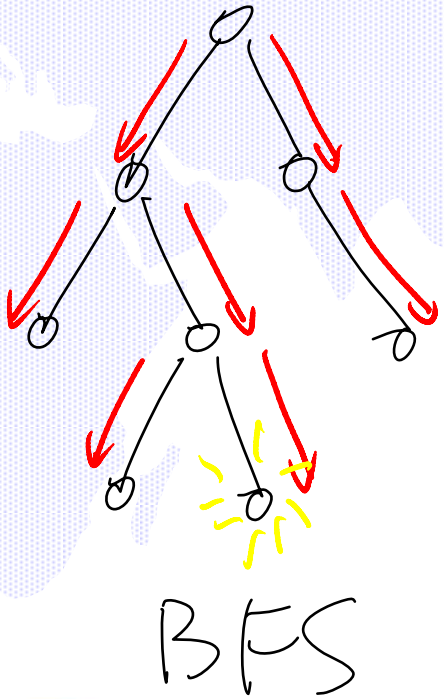
- 本来是想给大家介绍更多一些人工智能的内容，有人告诉我说，考前最好不要学新东西，于是还是把重点放在搜索上了。
- 搜索策略：DFS, BFS, DFSID
- 启发式搜索：A*, IDA*
- 优化策略：Dancing Links

# DFS, BFS & DFS-ID

# DFS vs. BFS

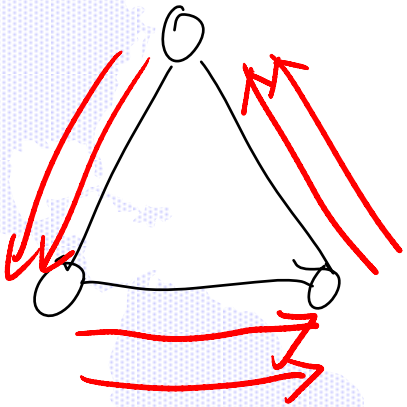
- 搜索往往是在一个图或是树中进行的。
- 考虑树的情况。

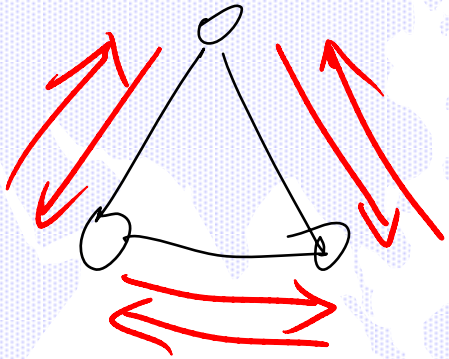


# DFS vs. BFS (cont.)

- 在树中，总能找到解。在图中就未必了。





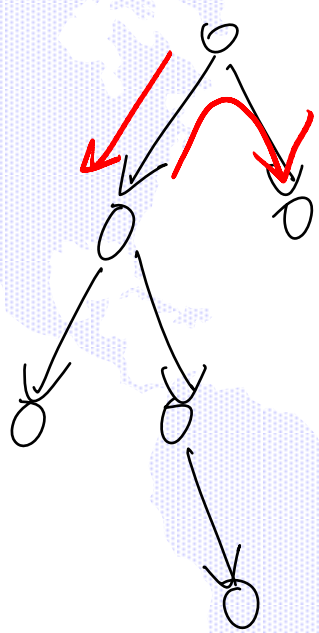
- 于是需要判重。
- 对于BFS，重复判断开销和队列开销是同一数量级的。
- 对于DFS，重复判断是一种很大的浪费。

# DFS vs. BFS (cont.)

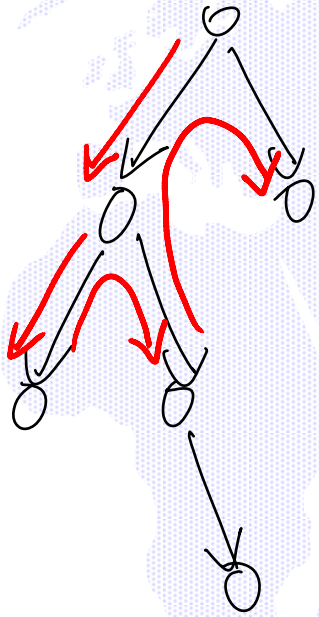
- 究竟哪一种更好？
- BFS总是能找到最优解。而实际上往往要求最优解。
- 但有时候队列开销过大。
- **试图将DFS改进为可以获得最优解的BFS！**

# DFS-ID

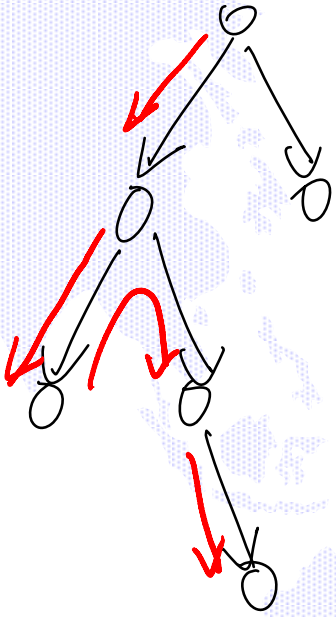
- 迭代加深 (iterative deepening)



d=1



d=2



d=3

# DFS-ID (cont.)

- 假设每个状态有k个子状态，BFS总共需要扩展

$$1 + k + k^2 + \ldots + k^d = \frac{k^{d+1} - 1}{k - 1}$$

- 个结点。DFS-ID则是

$$1 + (1 + k) + \ldots + (1 + k + \ldots + k^d) = ?$$

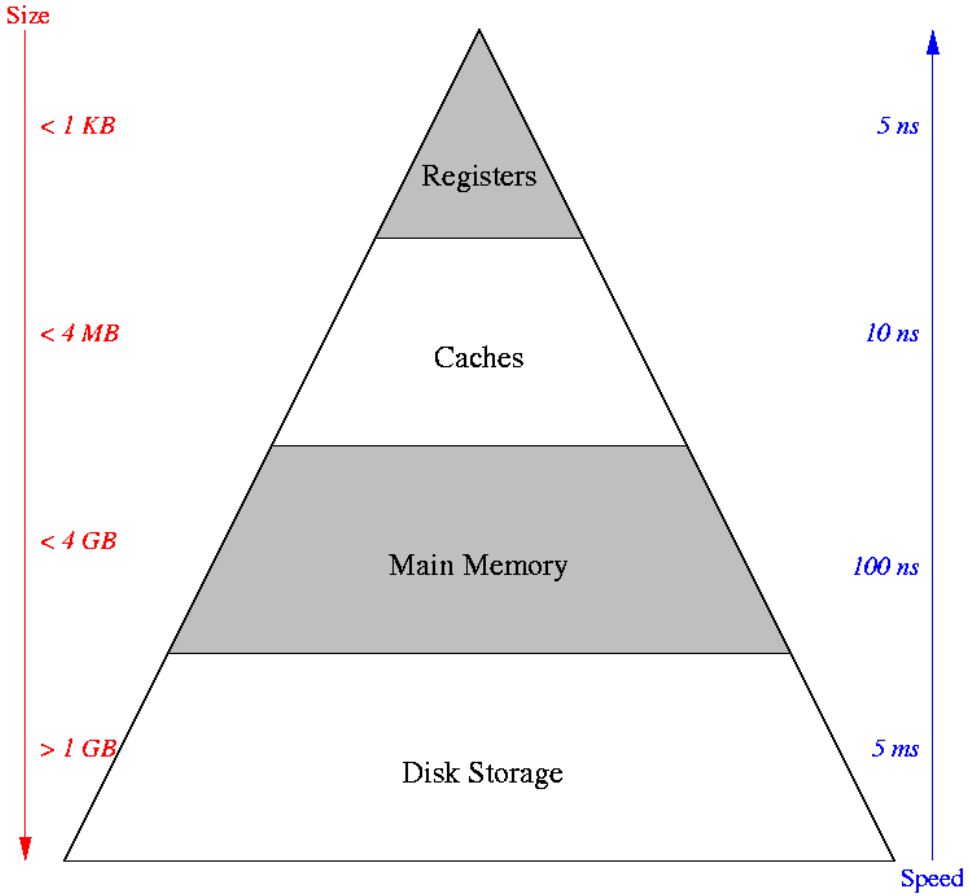
# DFS-ID (cont.)

$$
\begin{aligned}
& 1 + (1 + k) + \ldots + (1 + k + \ldots + k^d) \\
= & \sum_{i=0}^{d} \frac{k^{i+1} - 1}{k - 1} \\
= & \frac{1}{k-1}\left[k\left(\frac{k^{d+1}-1}{k-1}\right) - (d+1)\right] \\
= & \frac{k^{d+2} - 2k - kd + d + 1}{(k-1)^2} \leq \frac{k^{d+2}}{(k-1)^2} \\
= & \frac{k}{k-1}\frac{k^{d+1}}{k-1}
\end{aligned}
$$

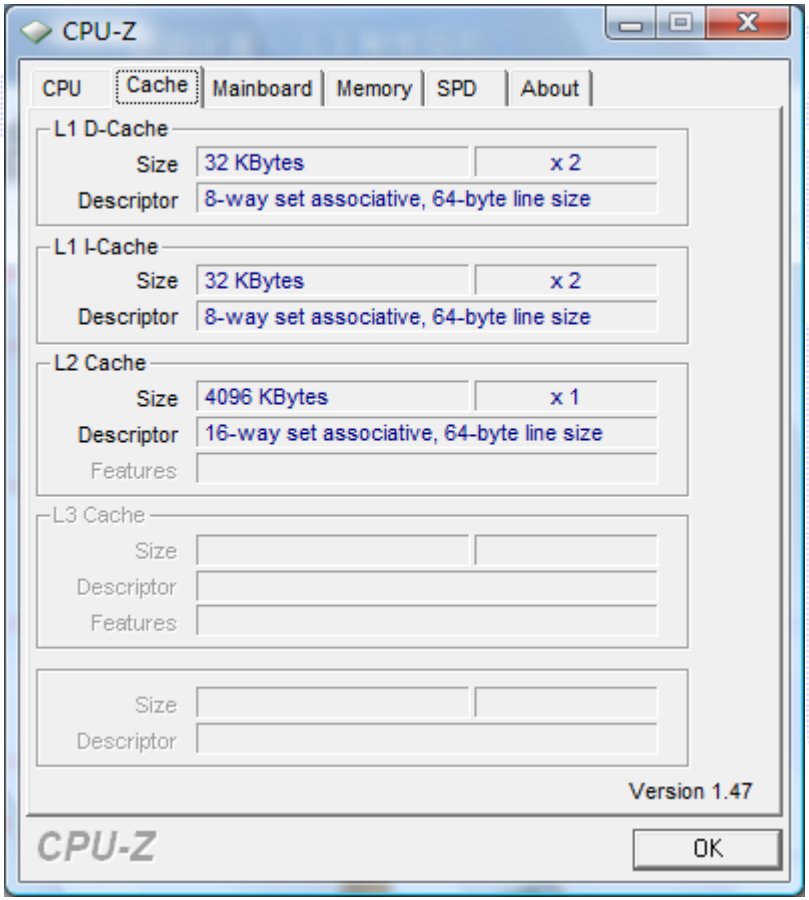
# DFS-ID (cont.)

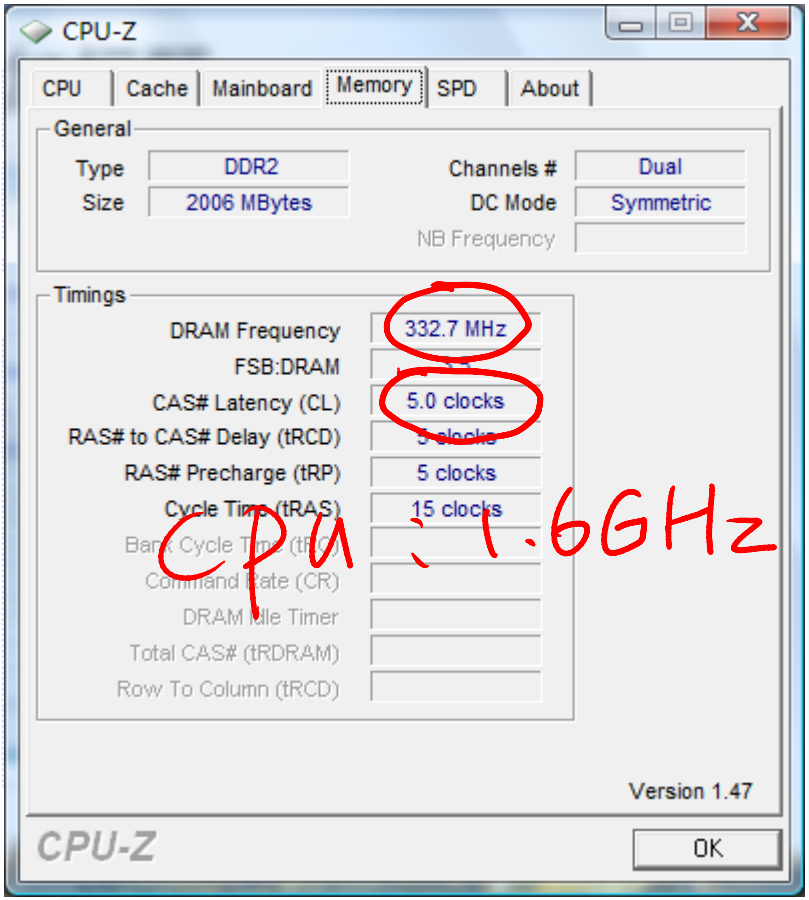
- 比BFS多产生了 $\frac{1}{k-1}$ 倍的结点。
- 实际这个数字很小。而且DFS比BFS有着不可比拟的优势：**内存**！
- 下面给大家补充存储器相关的内容。

# 存储器结构


Size
< 1 KB
< 4 MB
< 4 GB
> 1 GB
Registers
Caches
Main Memory
Disk Storage
5 ns
10 ns
100 ns
5 ms
Speed

# 存储器结构


CPU-Z
CPU
Cache
Mainboard
Memory
SPD
About
L1 D-Cache
Size
32 KBytes
x 2
Descriptor
8-way set associative, 64-byte line size
L1 I-Cache
Size
32 KBytes
x 2
Descriptor
8-way set associative, 64-byte line size
L2 Cache
Size
4096 KBytes
x 1
Descriptor
16-way set associative, 64-byte line size
Features
L3 Cache
Size
Descriptor
Features
Size
Descriptor
Version 1.47
CPU-Z
OK


CPU-Z
CPU
Cache
Mainboard
Memory
SPD
About
General
Type
DDR2
Size
2006 MBytes
Channels #
Dual
DC Mode
Symmetric
NB Frequency
Timings
DRAM Frequency
332.7 MHz
FSB:DRAM
CAS# Latency (CL)
5.0 clocks
RAS# to CAS# Delay (tRCD)
RAS# Precharge (tRP)
5 clocks
Cycle Time (tRAS)
15 clocks
Command Rate (CR)
DRAM Idle Timer
Total CAS# (tRDRAM)
Row To Column (tRCD)
CPU: 1.6GHz
Version 1.47
CPU-Z
OK

# 存储器结构

- L1-Cache的访问周期与CPU相同。
- Cache缺失时，需访问主存。往往损失数百个时钟周期。
- 主存缺失时，需访问磁盘。损失的时间以ms计算，非常多。

```
int a[N][N];
for (i=0;i<N;i++)
for (j=0;j<N;j++)
    a[i][j]=0;
```

```
int a[N][N];
for (i=0;i<N;i++)
for (j=0;j<N;j++)
    a[j][i]=0;
```

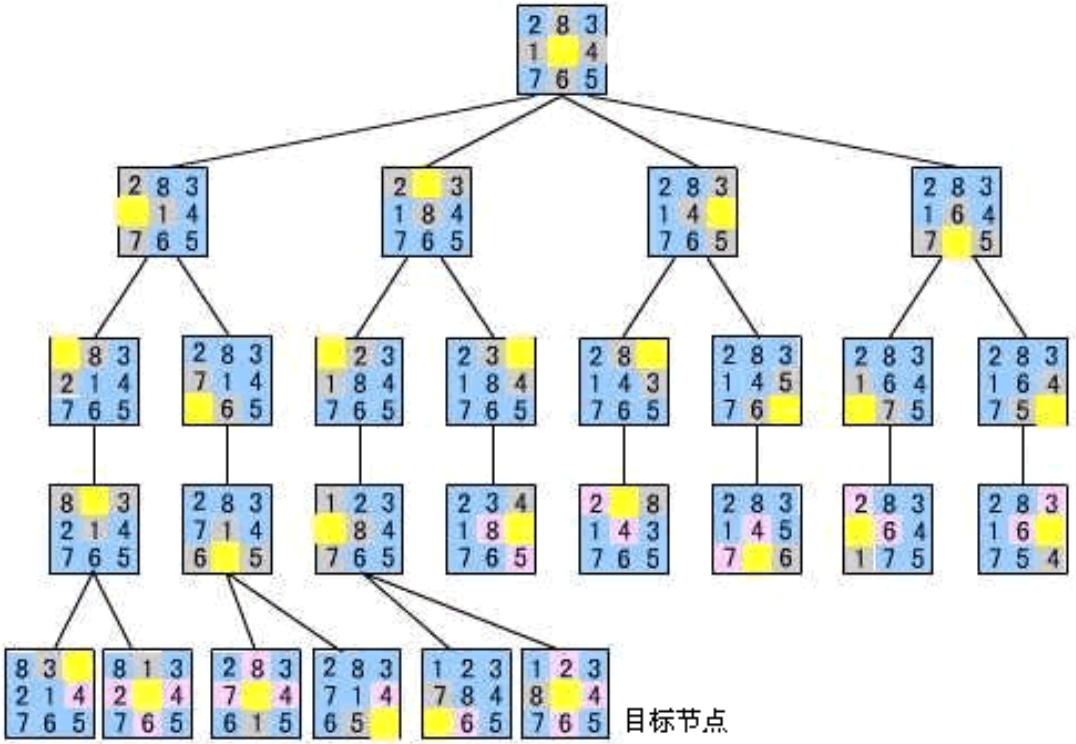
两个程序效率相差很大。 N越大，效率相差越大。

# DFS-ID vs. BFS

- DFS-ID比BFS多产生了 $\frac{1}{k-1}$ 倍的结点。
- 换来的是程序的局部访问。DFS中深度不会非常大，L2-Cache完全可容纳，局部频繁访问的总是可以在L1中找到。
- 当BFS仅仅需要几MB内存的时候，这个差异并不明显，甚至BFS可能会略快一些。但除非找到非常好的启发途径，现实中的问题是没有那么便宜的。采用DFS-ID至少不会引起溢出和换页所导致的时间损失。

# Example

- 连火星人都知道的八数码问题。



# 同时不要忘记

- **双向搜索也许成为解决的关键。**将反向BFS得到的结果保存在Hash表中备查，同时在正向做DFS-ID，对于确定目标状态的搜索问题是很有效的。
- 另外可做一些**可行性剪枝**，截去明显不可能的状态。但这一点应用比较灵活，每个问题的分析方式都不相同。
- 注意以上两点均可以在随后介绍的A*算法中应用。

# A* & IDA*

# 启发式搜索

- 我们可以人为估计一个状态的好坏。
- 用启发函数 $\hat{f}$ 代表结点的好坏，约定 $\hat{f}$ 比较小时，结果比较理想。
- 往往，

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

- 其中 $\hat{g}(n)$ 是从开始结点到n的路径长度，$\hat{h}(n)$ 表示对结点n的启发，即从当前结点到目标结点距离的估计。数值越小表明离目的越近。
- A*算法总是选择 $\hat{f}$ 最小的那个结点进行扩展。

# 启发式搜索

- 两个特殊情况：

- 当$\hat{h}(n) = h(n)$时，A*算法演变为贪心法，直接求得最优解。
- 当$\hat{h}(n) = 0$时，A*算法演变为Dijkstra算法，特别的，当边权为1时，实际就是原始的BFS算法。

- A*算法实际上是盲目搜索的扩展。

# 可接纳性定理

- 若搜索图中权值非负，且对搜索图中的所有结点有

$$\hat{h}(n) \leq h(n)$$

- 则第一个被选中的目标结点确定了到达目标的最优路径。

- 与Dijkstra算法非常类似。

# 可接纳性定理(cont.)

- 引理：在A*终止前的每一步，总有一个结点n*，满足：
- (1) n*在到达目标的一条最佳路径上
- (2) A*已经发现了到达n*的最佳路径
- (3) $\hat{f}(n^*) \leq h(n_0)$

- 证明采用数学归纳法。

# 可接纳性定理(cont.)

- 接下来证明可接纳性定理。反证法：
- 假设算法终止时并非最优解g2。对于最优解g1，有

$$f(n_{g1}) = f(n_0)$$

- 在g2终止时，$\hat{f}(n_{g2}) \geq f(n_{g2}) > f(n_0)$。
- 根据引理，此时队列中一定存在

$$\hat{f}(n^*) \leq f(n_0)$$

- A*算法总是寻找最小权值的结点扩展，于是得出矛盾，可接纳性定理得证。

# 可接纳性定理(cont.)

- 可见，启发函数的数值不能大于从当前状态到目标状态的最优值，在此限制下，启发函数越大越好。越大表明与正确的贪心算法越接近。

# IDA*

- A*算法需要维护一个优先队列。通常用堆实现，取最小操作将耗费很多的时间，还需要判重以减少队列中的元素个数。这个算法与Dijkstra算法非常的类似。

- 同样，可以把DFS-ID中的思路放到A*算法中。
- 每次选择启发函数最小的点进行DFS，当深度到达某个限制时回溯。但IDA*的最大优势在于，无需维护全局最小值和判重！
- **它成为求解启发式搜索的理想工具。**

# IDA* (cont.)

- 在Heap-A*中采用优先队列，IDA*中则是将估价函数作为**剪枝条件**。当

$$\hat{g}(n) + \hat{h}(n) > L$$

- 时，由于

$$\hat{g}(n) + \hat{h}(n) \leq g(n) + h(n)$$

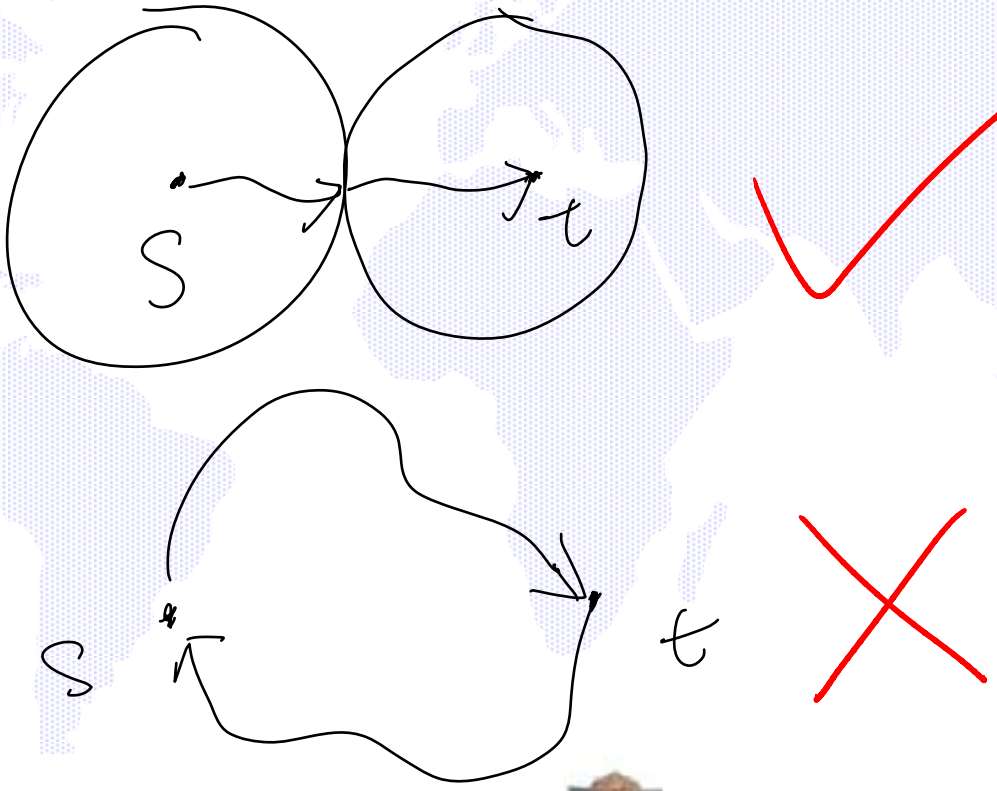
- 有

$$g(n) + h(n) > L$$

- 所以无论如何也无法得到预期的解，则剪枝。

# 双向搜索

- 双向的启发式搜索可能引起不正确的结果：



# 双向搜索(cont.)

- 观察启发函数的形式

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

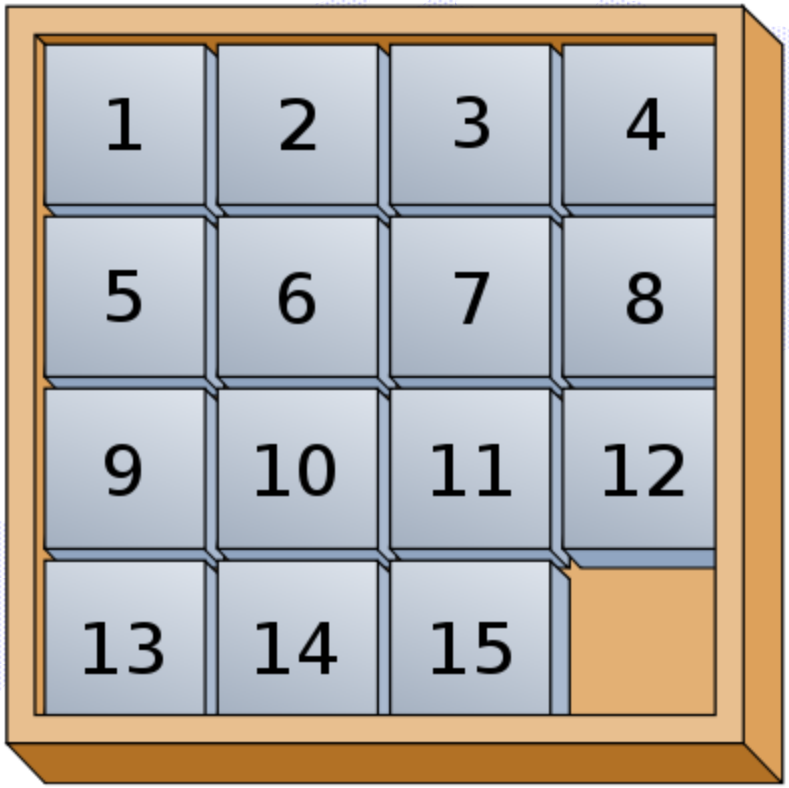
- 以及需要满足的条件

$$\hat{h}(n) \leq h(n)$$

- 当$h(n)$可知时，用$h(n)$代替$\hat{h}(n)$会收到较好的效果。
- 这一点同样可以应用在Heap-A*和IDA*上。

# 启发式搜索例题

# 15-puzzle

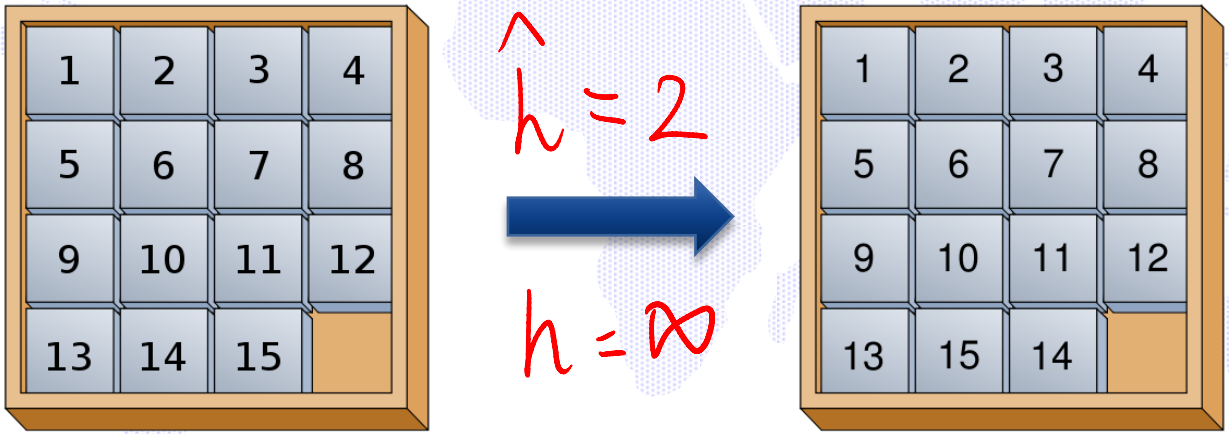


# 启发函数

$$\hat{h}(n) \le h(n)$$

$$\hat{h}(n) = \max\{\hat{h}_1(n), \hat{h}_2(n), \ldots\}$$

- 曼哈顿距离和。易见它满足条件，但不够精确：



# 启发函数(cont.)

- 交换两个数字的位置，无论如何，2步都是无法达到的。（不考虑其他数字的情况）
- 考虑两个数字已经在目的地所在的那一行了，则至少需要增加两次的移动（让出位置）。这个性质对列也同样。
- 可以用曼哈顿距离加上这个数值作为较好的$\hat{h}(n)$。

- 还可以考虑任何其他的可行的启发方式，包括双向搜索。

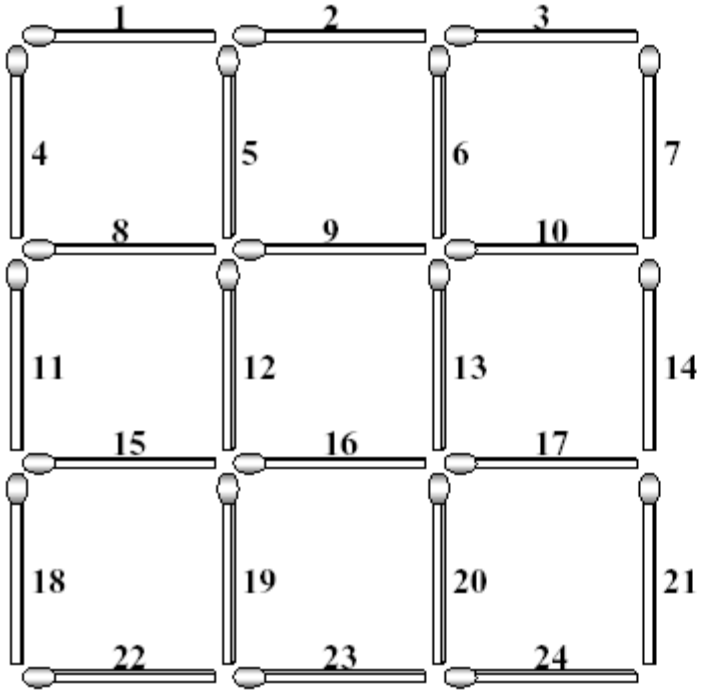
# 15数码问题

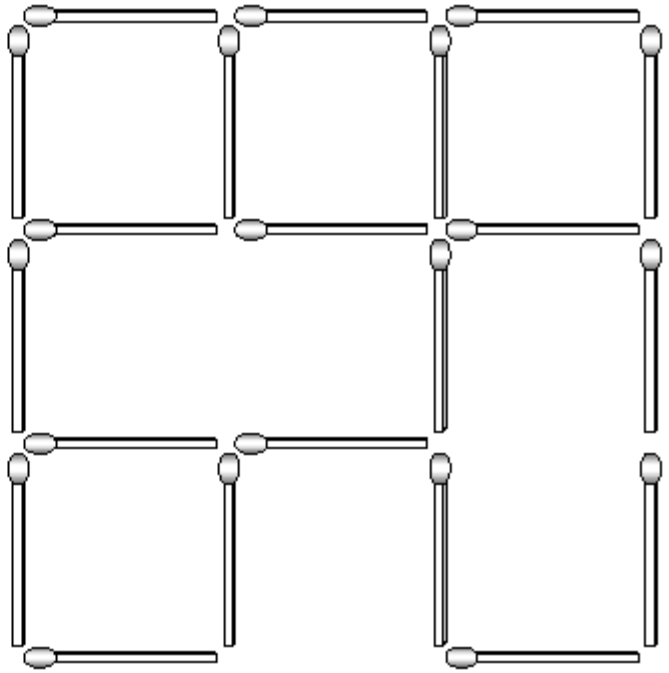
| | | | |
|---|---|---|---|
| 15 | 14 | 13 | 12 |
| 11 | 10 | 9 | 8 |
| 7 | 6 | 5 | 4 |
| 3 | 1 | 2 | |

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | |

# Destroying Squares

- 消灭方块, n<=10。以下是一个3x3的例子。



# 试图做一种特殊的探索

- **给定一个二分图，用最少数量的左边结点支配右边结点！**
- 可以将它归约到一个NPC问题。

- 那就试图设计一个满足 $\hat{h}(n) \leq h(n)$ 的启发函数。
- 启发函往往贪心算法将得到一个上界。这个上界可以用于剪枝（明显无法达到），但不能用作启发函数。
- 数需要估计一个下界。

# 一个实际效果很好的启发函数

- 对于当前状态，已经选择了部分左边点。
- (1) ans = 0
- (2) 如果所有右边点都被覆盖，结束
- (3) ans = ans + 1
- (4) 任选一个未被覆盖的右边点，将关联它的左边点全部选中，这将引起其他右边点被覆盖
- (5) 转(2)

- 在右边点的选取上，可以优先取关联的左边点较少的那些，增加估计函数的值。用这个启发可以很轻松地解决这个问题。

# 更强的算法

- 是否能回忆起，很多最小值问题的近似算法都存在

$$A^* \leq \rho A$$

- 这样形式的界。变形后得到

$$A \geq \frac{1}{\rho} A^*$$

- 作为估计函数，利用

$$\hat{h}(n) = \max\{\hat{h}_1(n), \hat{h}_2(n), \ldots\}$$

- 加强优化。

# 解决这个问题的意义

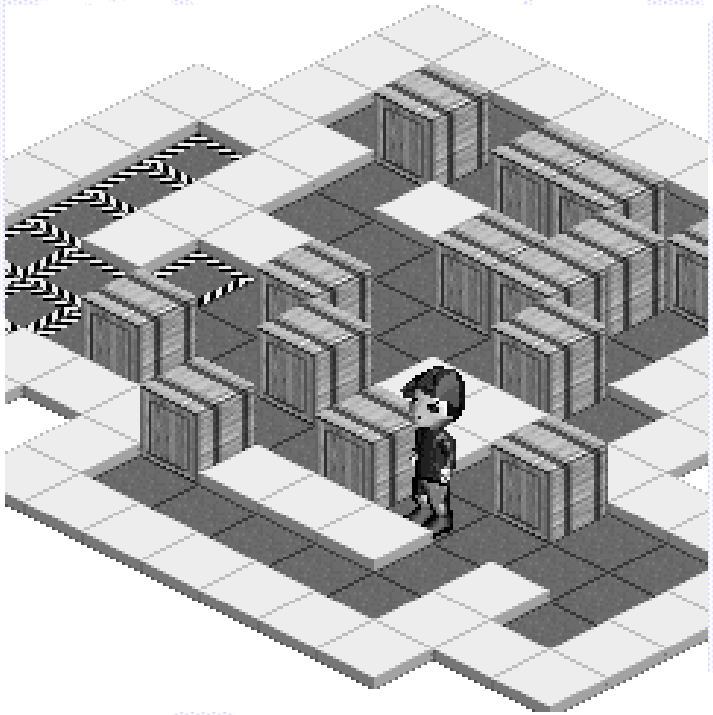
- 这个模型是个很广泛的模型，甚至很多存在P算法的问题，都可以避免建模过程直接暴力构造这样的模型进行求解。
- **IDA*的效果有多好，请同学们自己动手实践！**
- 配合接下来的常数优化，效果将更好。

# 讨论题：15-puzzle II

- 1-16这些数字填入了一个4x4的方格
- 每次允许将一行或一列进行旋转操作：
- 1 2 3 4 => 2 3 4 1
- 1 2 3 4 => 4 1 2 3
- 问初始状态到目标状态至少需要多少步。

# 讨论题：Sokuban

- 推箱子, n,m<=8(含边界)



```
6 7
######.
#+##+@#.
#Bb..##
#.#.b.#
#.....#
#######
```

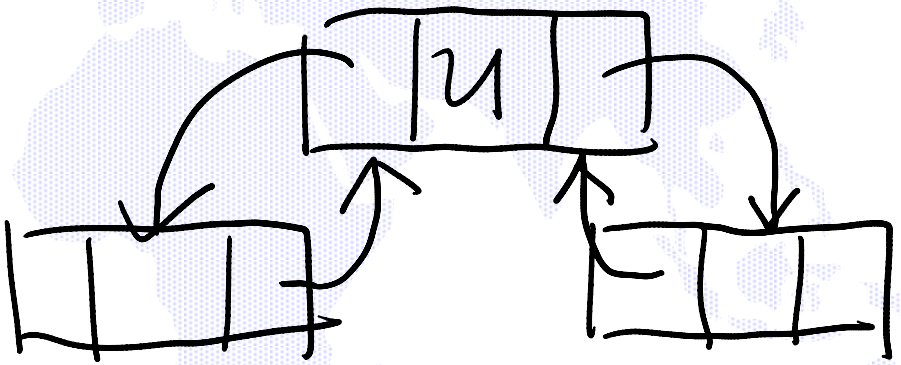
# Dancing Links & DFS Speedup

# 加速动机

- 全排列：
- void dfs(int l) {
- for (i = 0; i < n; i ++)
- if (!used[i]) {
- number[l] = i;
- used[i] = false;
- dfs(l + 1);
- }
- }
- **循环处当层数增加时，空转严重！**

# 双向链表实现

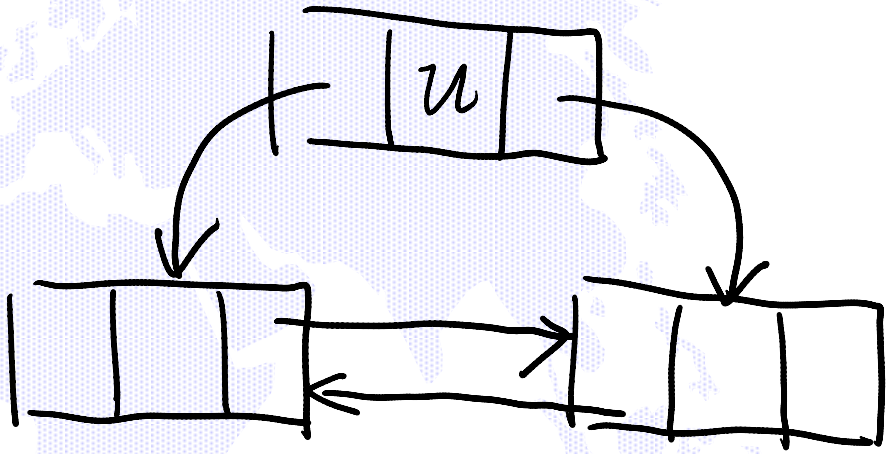
- 用A[n]表示一个长度为n的链表
- L[u] 表示u左边的结点
- R[u] 表示u右边的结点
- 删除：
- L[R[u]] = L[u];
- R[L[u]] = R[u];




- 注意u并未被删除

# 双向链表实现

- 用A[n]表示一个长度为n的链表
- L[u] 表示u左边的结点
- R[u] 表示u右边的结点
- 恢复：
- L[R[u]] = u;
- R[L[u]] = u;




- **只要按栈的规则进行删除、恢复，链表被原样保持！**

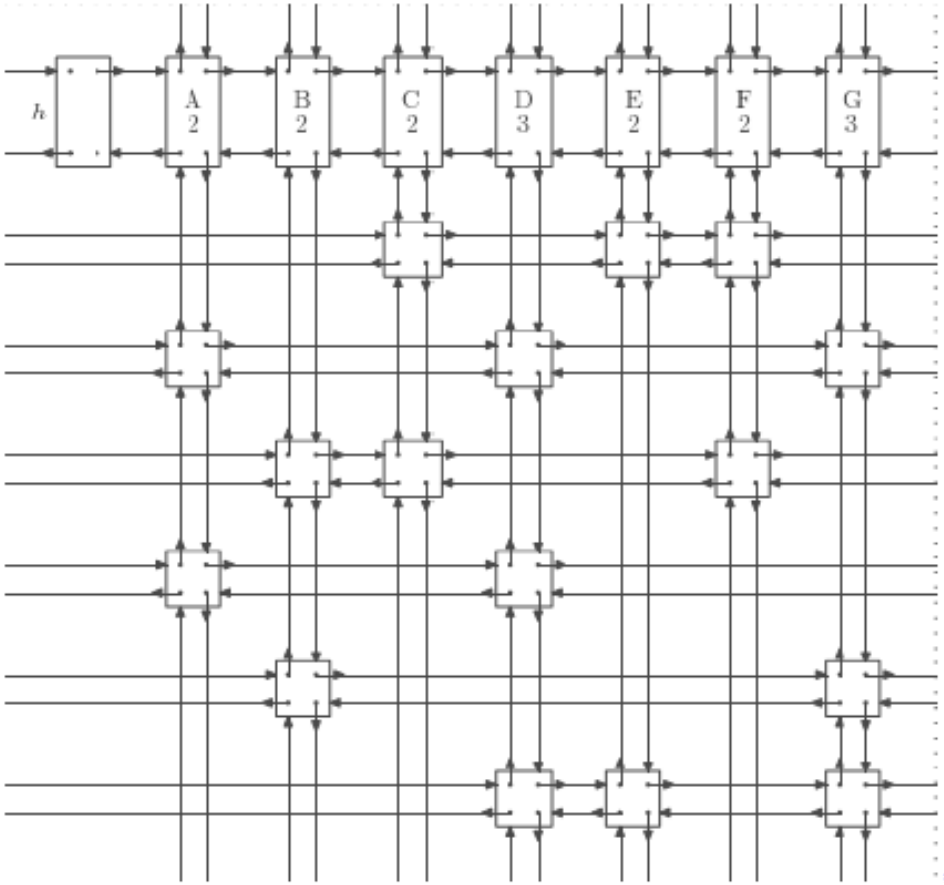
# 全排列

```
void dfs(int l) {
    for (i = R[0]; i != -1; i = R[i]) {
        remove(i);
        number[l] = i;
        dfs(l + 1);
        restore(i);
    }
}
```

- remove, restore 均为O(1)操作，提高了效率

# 推广到二维

- 如果用十字链表存储二维矩阵，则同样可以用Dancing Links进行删除与恢复。
- 做法是在行上做一次，再在列上做一次。



# Exact Cover Problem

- 给定01矩阵，选择一些行，使得选出那些行中，每一列仅有一个1。

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

# Exact Cover Problem (cont.)

- 算法：
- [1] if (A空) 返回一组解;
- [2] 选择一个未被覆盖的列c
- [3] 将所有与c相关的行删除
- [4] 对每一个与c相关的行,执行DFS
- [5] 恢复那些行

- **可以体会一下Dancing Links的应用！**
- **要注意恢复的顺序与删除的顺序应当保持栈的关系。**

# Exact Cover Problem (cont.)

- 更多的优化：
- [1] if (A空) 返回一组解;
- [2] 选择一个未被覆盖的列c
- **在这里可以选择包含行数最少的列，以减少搜索分支数。**
- [3] 将所有与c相关的行删除
- [4] 对每一个与c相关的行,执行DFS
- **这里选择哪些行比较好呢？**
- [5] 恢复那些行

# Sudoku

- 加强版，保证唯一解

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A | | | | | C | | | | | | O | | I |
| | J | | | A | | B | | P | | C | G | F | | H | |
| | | D | | | F | | I | | E | | | | | P | |
| | G | | E | L | | H | | | | | M | | J | | |
| | | | | E | | | | | C | | | G | | | |
| | I | | | K | | G | A | | B | | | | E | | J |
| D | | G | P | | | J | | F | | | | | A | | |
| | E | | | | C | | B | | | D | P | | | O | |
| E | | | F | | M | | | D | | | L | | K | | A |
| | C | | | | | | | | | O | | I | | L | |
| H | | P | | C | | | F | | A | | | B | | | |
| | | | G | | O | D | | | | J | | | | | H |
| K | | | | J | | | | | H | | A | | P | | L |
| | | B | | | P | | | E | | | K | | | A | |
| | H | | | B | | | K | | | F | I | | C | | |
| | | F | | | | C | | | D | | | H | | N | |

a) Sudoku grid

Figure 1. Sudoku

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| F | P | A | H | M | J | E | C | N | L | B | D | K | O | G | I |
| O | J | M | I | A | N | B | D | P | K | C | G | F | L | H | E |
| L | N | D | K | G | F | O | I | J | E | A | H | M | B | P | C |
| B | G | C | E | L | K | H | P | O | F | I | M | A | J | D | N |
| M | F | H | B | E | L | P | O | A | C | K | J | G | N | I | D |
| C | I | L | N | K | D | G | A | H | B | M | O | P | E | F | J |
| D | O | G | P | I | H | J | M | F | N | L | E | C | A | K | B |
| J | E | K | A | F | C | N | B | G | I | D | P | L | H | O | M |
| E | B | O | F | P | M | I | J | D | G | H | L | N | K | C | A |
| N | C | J | D | H | B | A | E | K | M | O | F | I | G | L | P |
| H | M | P | L | C | G | K | F | I | A | E | N | B | D | J | O |
| A | K | I | G | N | O | D | L | B | P | J | C | E | F | M | H |
| K | D | E | M | J | I | F | N | C | H | G | A | O | P | B | L |
| G | L | B | C | D | P | M | H | E | O | N | K | J | I | A | F |
| P | H | N | O | B | A | L | K | M | J | F | I | D | C | E | G |
| I | A | F | J | O | E | C | G | L | D | P | B | H | M | N | K |

b) Solution

# Sudoku ⇔ Exact Cover Problem

- 行：
- 16 x 16 x 16 = 4096
- 一共16 x 16小格，每小格有16种可能(A..P)
- 列：
- (16 + 16 + 16) x 16：每个字母都需要占据一个行、列或大块
- 16 x 16：每个小块只能放在一个小格中(这一个条件的加入是为了填满整个棋盘)

- A[i][j] = 1 当且仅当 行i 可以放在 列j 处。

- **你认为不可能出解吗？试验一下你就知道了！**

# 还可以做什么呢？

- **Dancing Links还可以用在IDA*上。**
- Destroying Squares就可以采用这个方法减少搜索常数，进一步提高效率。
- 几乎任何DFS模型的搜索都可以套用Dancing Links进行优化，而且获得很大的效率提升。作为一个附加性的通用算法，加上代码简洁，这个优化途径很值得采纳。

# 总结

- 有两种算法：
- [1] 保证正确，但为了得到正确的结果，可能需要很多的时间，如IDA*
- [2] 不保证正确，很快可以求出一个不错的解，可以求很多次，如随机调整、遗传算法等
- 应当如何选择？

# 推荐的练习

- [1] Dancing Links优化的实现。可以直接尝试Exact Cover，也可以从全排列开始，比较各种算法的效率。
- [2] 用Dancing Links和IDA*实现二分图特殊支配问题。
- [3] 用这两个模型解决一些其他问题。
- [4] 尝试构造不同评估函数解决遇到的搜索问题。