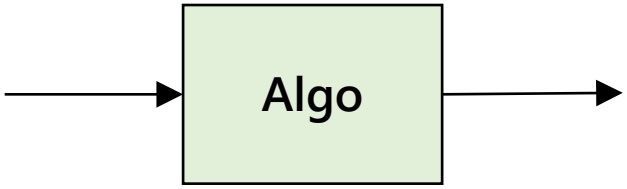
总结

已知的图算法（截至目前）

- 深度优先搜索（DFS）
  - 输入：任意图
  - 输出选项：DFS 顺序、死胡同顺序、生成树
  - 适用场景：需要访问所有元素（顶点）的问题
- 广度优先搜索（BFS）
  - 输入：任意图
  - 输出选项：BFS 顺序，生成树
  - 适用情况：问题需要遍历所有节点（顶点）
- 连通分量
  - 输入：任意图
  - 输出选项：计数或布尔值（“是否连通”）
  - 注意：已修改的 DFS/BFS
  - 适用情况：确定有多少个顶点“簇”

摘要

已知的图算法（截至目前）



- 拓扑排序
  - 输入：有向无环图（DAG）
  - 输出：顶点的线性排列
  - 注意：我们知道两种算法——修改后的深度优先搜索和分治法
  - 适用场景：需要找到顶点的顺序时
- 最小生成树（MST）
  - 输入：带权图
  - 输出：树
  - 注：我们知道两种算法——Prim、Kruskal
  - 适用场景：需要构建最便宜的连通网络时
- 单源最短路径（SSSP）
  - 输入：带权图 + 起始顶点
  - 输出选项：“长度”数组，最短路径树（即“prev”数组）
  - 注：Dijkstra 算法
  - 适用场景：查找从某一特定顶点到所有其他顶点的最短路径

第10讲

COMP 3760

动态规划

课本第8章


https://www.flickr.com/photos/monceau/52504346305/

https://www.flickr.com/photos/monceau/52504346305/

https://www.flickr.com/photos/monceau/52504346305/

https://www.flickr.com/photos/monceau/52504346305/

你好，亲爱的老朋友们：
斐波那契数列

- 0，1，1，2，3，5，8，13，21，34，55，...

- Each number is the sum of the previous two:

  fib(0) = 1

  fib(1) = 1

  fib(n) = fib(n-1) + fib(n-2)

- 我们能计算多少?

DEMO

经典的递归算法：

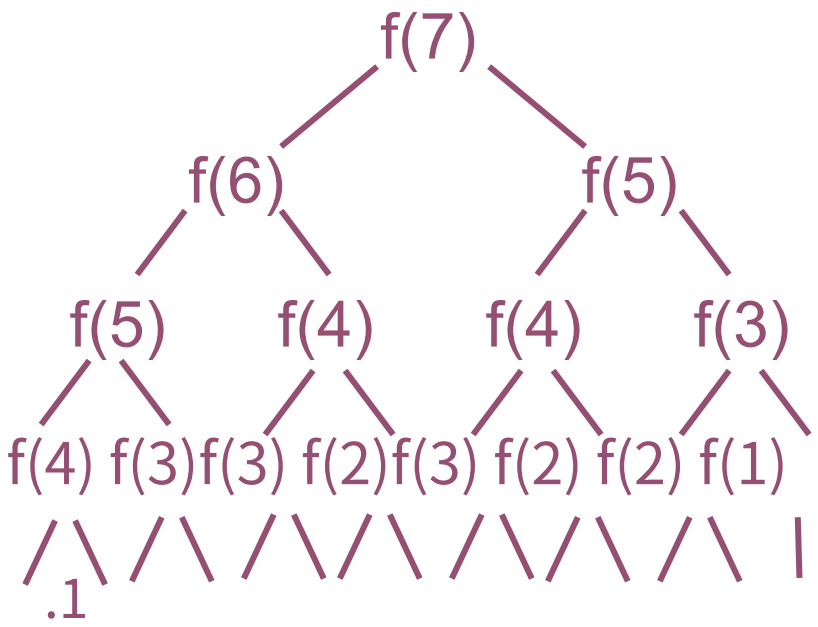
fib (n):如果 n < 2返回 n // 即 fib(0) 为 0，
fib(1) 为 1 否则 返回 fib(n-1) + fib(n-2)

斐波那契数列：

你为什么这么慢？

```
fib (n)：
    如果 n < 2
        返回 n
    else
        返回 fib(n-1) + fib(n-2)
```

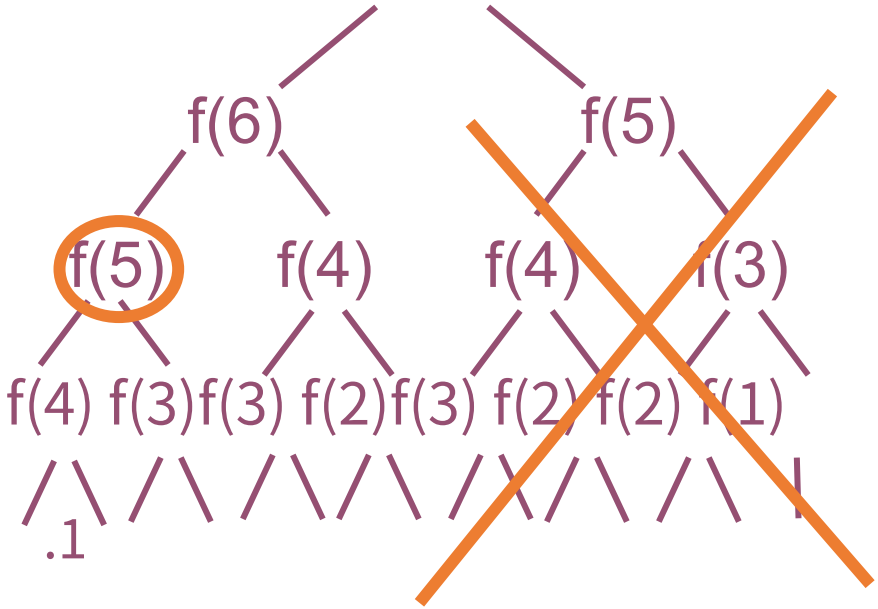
执行树：

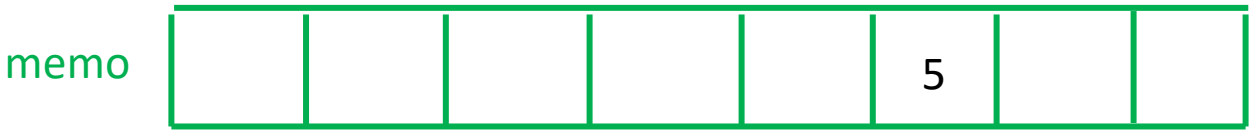


F(n) 的计算需要指数级时间。

时空权衡

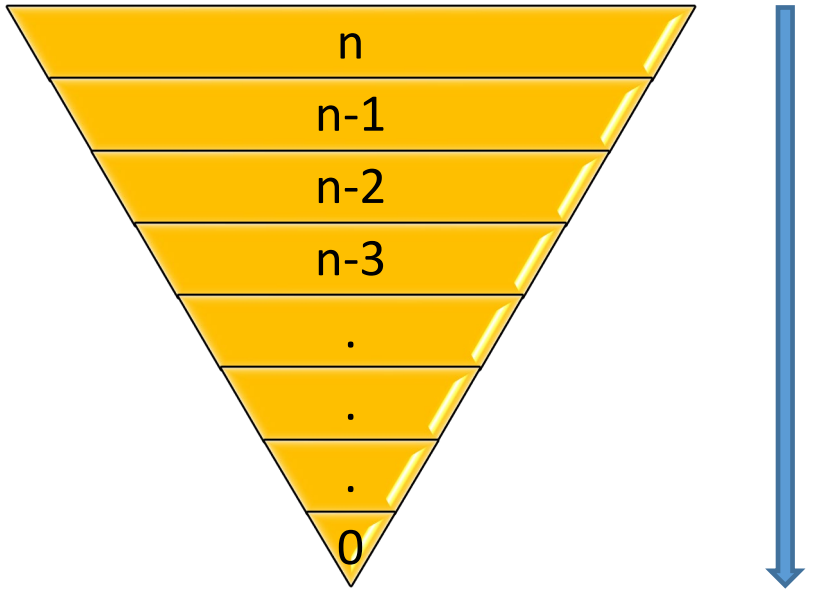
- 通过记住结果来增强算法
  你一路相随 f(7)





斐波那契，自顶向下

```
fib (n) {
    if memo[n] exists, return it
    如果 n < 2
        return n
    else
        f = fib(n-1) + fib(n-2)
        memo[n] = f
        return f
}
```



自顶向下（递归）

memo

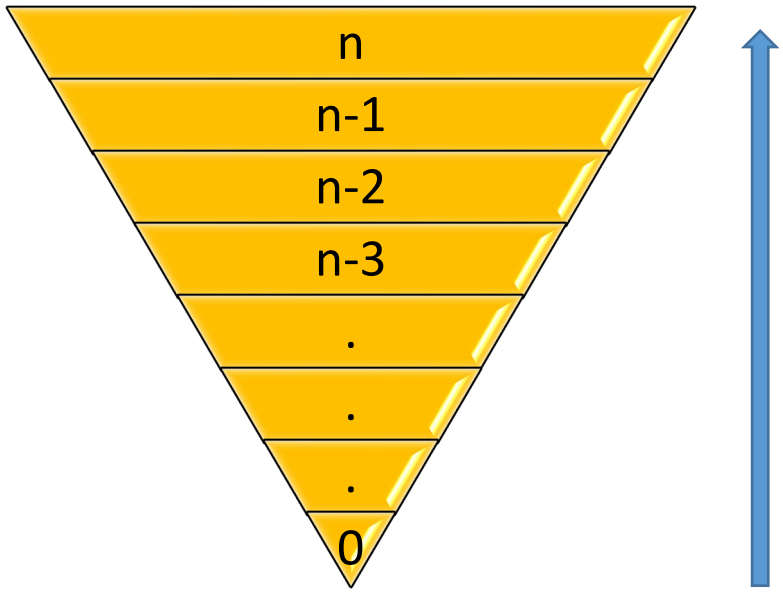
0	1	1	. . .	*fib*(*n*-2)	*fib*(*n*-1)	*fib*(*n*)

效率：

- 时间：O(n)
- 空间：需要一个大小为 O(n) 的数组

斐波那契，自底向上

```
fib (n) {
    memo[0] = 0;
    memo[1] = 1;
    for i ← 2 to n do
        memo [i] = memo[i-1] + memo[i-2]
    return memo[n]
}
```



自底向上

memo

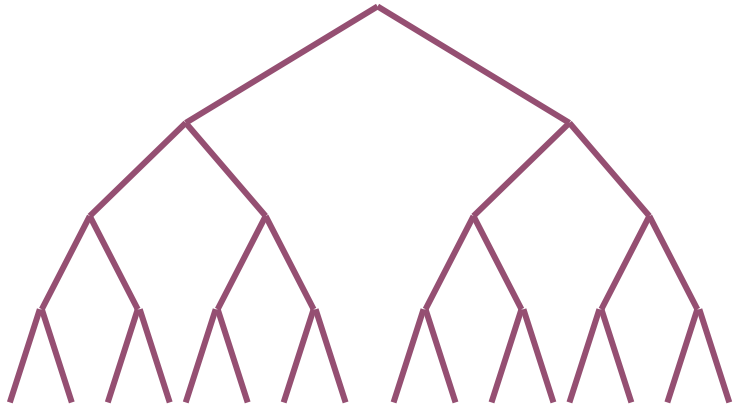
0	1	1	. . .	*fib*(*n*-2)	*fib*(*n*-1)	*fib*(*n*)

效率：

- 时间：O(n)
- 空间：需要一个大小为 O(n) 的数组

动态规划

- 要点：记住对子问题递归定义的解并用它们来解决问题
- 将子问题的解保存以备重复使用。
- 当许多子问题会重复出现时，这是个好主意
- 动态规划 不是 分治法
  - 分治法和动态规划都使用递归（至少在概念上）



动态规划概述

- **第 1 步：**
  - 将问题分解为更小的、等价的子问题
- **第 2 步：**
  - 用子问题来表示解法
- **第 3 步：**
  - 使用表格自底向上计算最优值
- **步骤 4：**
  - 根据步骤 1-3 找到最优解

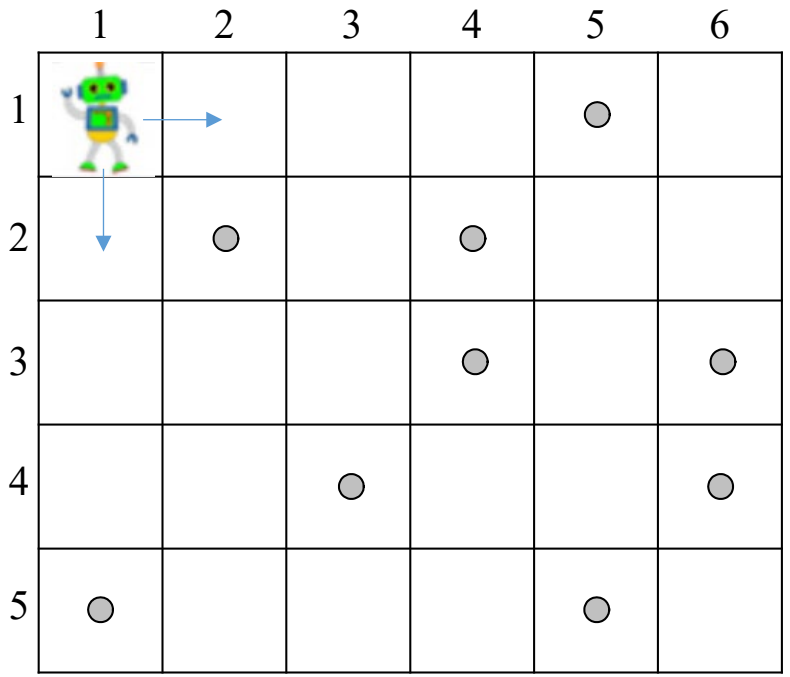
动态规划示例

- 斐波那契数
- 机器人收集硬币
- 传递闭包（Warshall 算法）
- 所有点对最短路径（Floyd 算法）

动态规划：收集硬币的机器人（第8章）

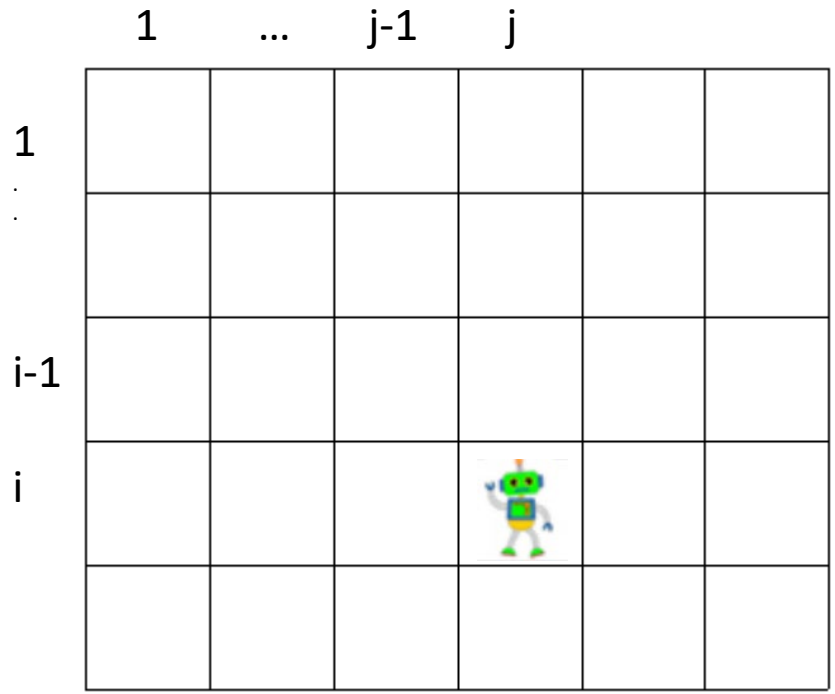
收集硬币的机器人

若干硬币被放在一个 $n \times m$ 棋盘的格子里。机器人位于棋盘的左上角格子，需要尽可能多地收集硬币并将它们带到右下角格子。机器人只能向右或下移动。



解法

- 令 F(i,j) 为机器人能在第i行第*j*列的格子上收集并带到该格子的最多硬币数。

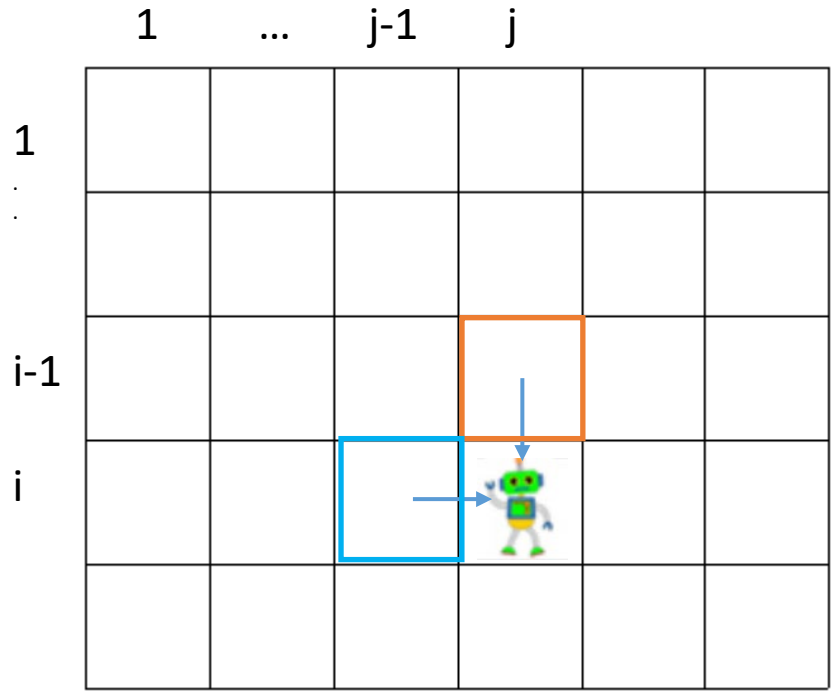


解答

机器人最多能带到格子 (i,j) 的硬币数是多少?

如果它从左侧来 → F(i, j-1)

如果它来自上方 → F(i-1, j)

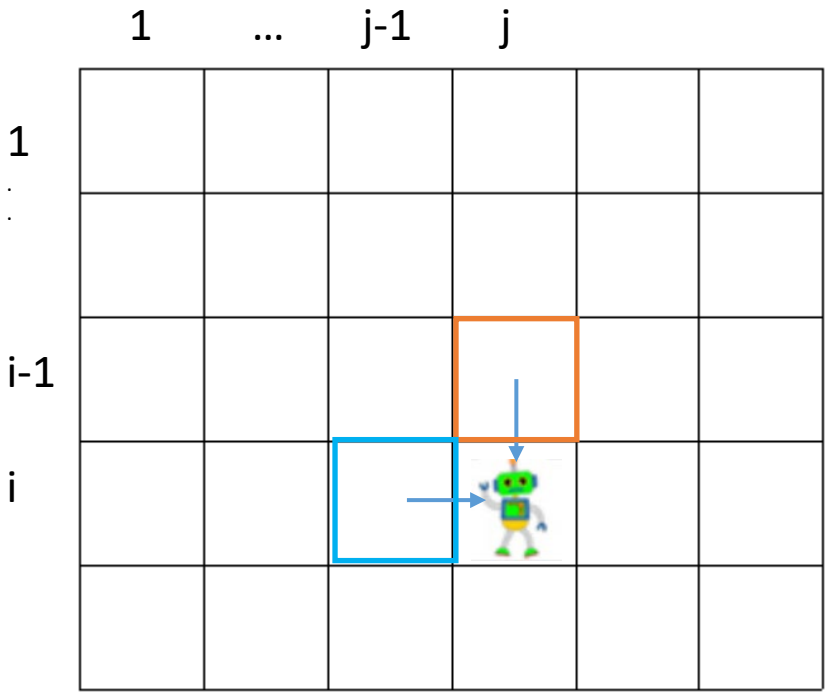


解法

递归定义：

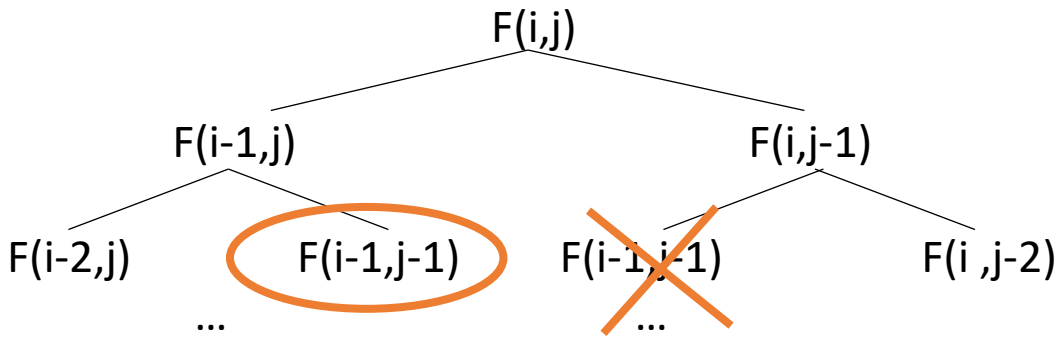
$$F(i, j) = \max\{F(i-1, j),\ F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n,\ 1 \le j \le m$$

where $c_{ij} = 1$ 如果在单元格(i,j)有一枚硬币，则为 $c_{ij} = 0$ 否则



解（续）

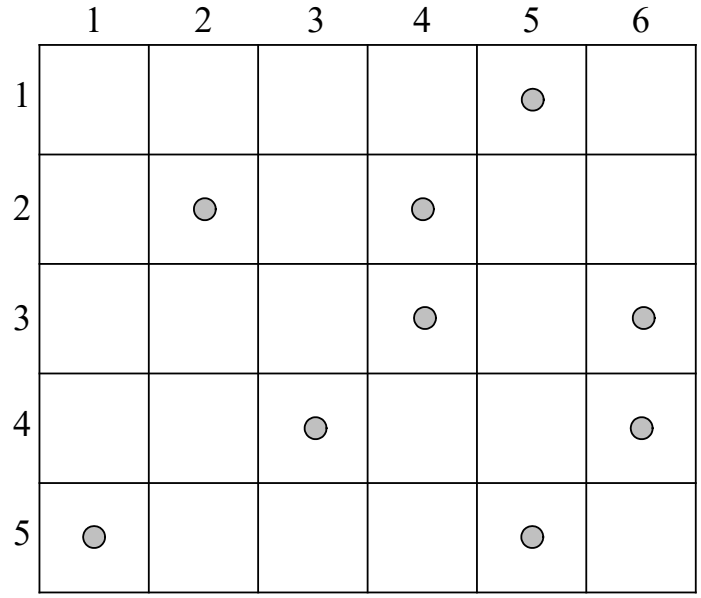
- F(*i*, *j*) = 最大值{F(*i*-1, *j*), F(*i*, *j*-1)} + c_{ij}
- F(0, *j*) = 0 当 1 ≤ *j* ≤ = m 且 F(*i*, 0) = 0 当 1 ≤ *i* ≤ = n。



解答（续）

自下而上的计算

$$F(i, j) = \max\{F(i\text{-}1, j),\ F(i, j\text{-}1)\} + c_{ij} \quad \text{for } 1 \le i \le n,\ 1 \le j \le m$$



	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

机器人硬币收集

```
ALGORITHM RobotCoinCollection(C[1..n, 1..m])
  // Robot coin collection using dynamic programming
  // Input: Matrix C[1..n, 1..m] with elements equal to 1 and 0 for
  //        cells with and without coins, respectively.
  // Output: Returns the maximum collectible number of coins
  F[1, 1] ← C[1, 1]
  for j ← 2 to m do
    F[1, j] ← F[1, j - 1] + C[1, j]
  for i ← 2 to n do
    F[i, 1] ← F[i - 1, 1] + C[i, 1]
    for j ← 2 to m do
      F[i, j] ← max(F[i - 1, j], F[i, j - 1]) + C[i, j]
  return F[n, m]
```

复杂度？ $\Theta(nm)$ 时间，$\Theta(nm)$ 空间

动态规划 要点速览

- 理解问题
- 为问题建立递归定义
  - 子问题是什么？*subproblems*
  - 子问题如何相互*related*?
- 决定如何存储子问题的结果
- 用于计算/填充数据结构的算法

动态规划：传递闭包

（第8章）

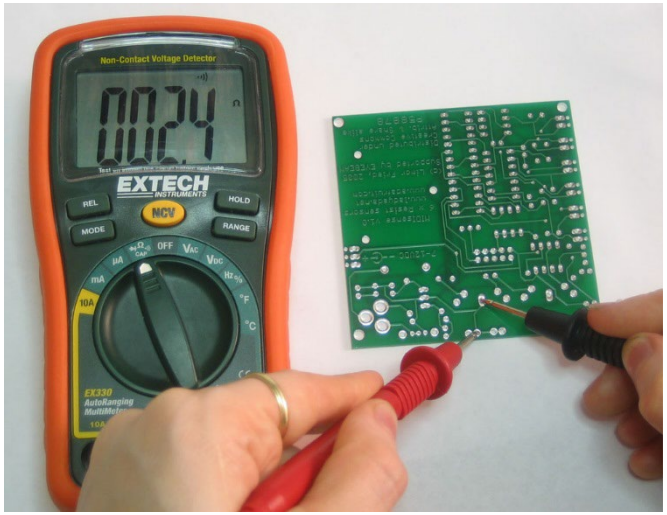
传递闭包

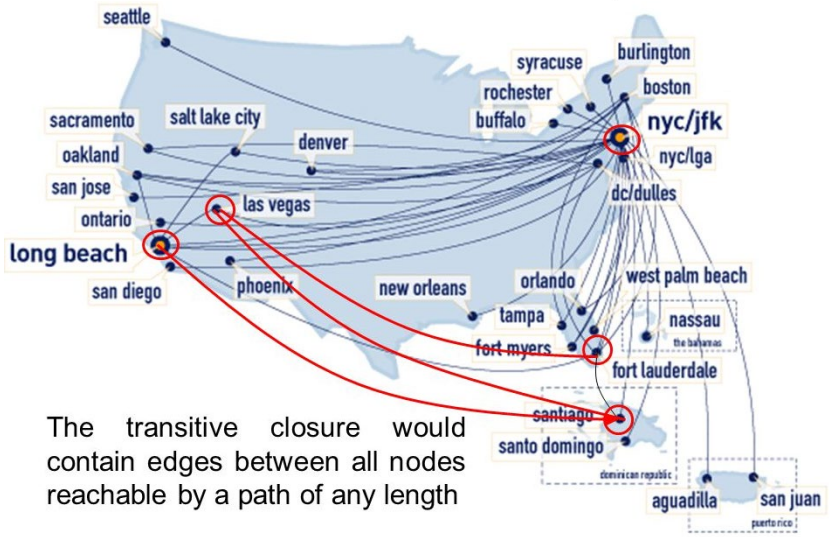
- 哪些节点可以从其他节点到达?
- 问题：
  - 给定一个有向无权图 G，具有 n 个顶点，找出从顶点 v_i 到 v_j 存在的所有路径，
    对于所有 $1 \leq (i, j) \leq n$

- 注意：该问题通常使用邻接矩阵的图表示法来解决

传递闭包

- 应用：
  - 数字电路测试、可达性测试

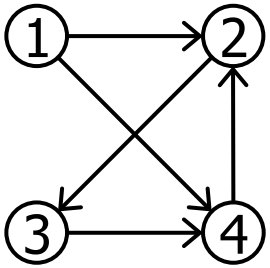




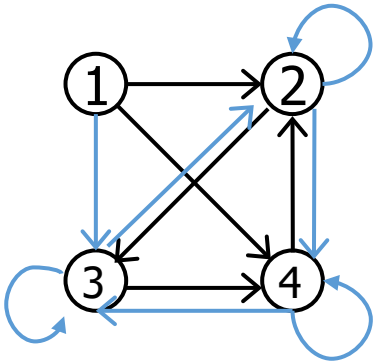
The transitive closure would contain edges between all nodes reachable by a path of any length

传递闭包

- 算法思路：
  - 创建一个新图，其中每条边表示原图中的一条路径在原图中



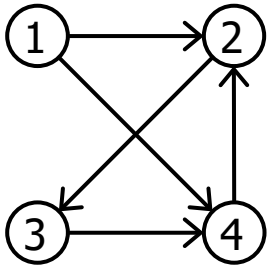
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0



	1	2	3	4
1	0	1	1	1
2	0	1	1	1
3	0	1	1	1
4	0	1	1	1

传递闭包示例

- 考虑下图及其对应的邻接矩阵……



	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

- 我们称这个初始矩阵为 R^0。
  - 为方便起见，这里我们使用从 1 开始的数组：A[1..n][1..n]

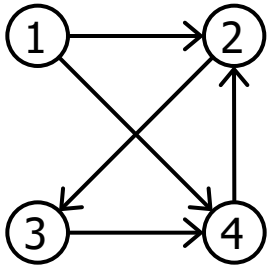
传递闭包

步骤 1：

- 选择第 1 行和第 1 列
- 对所有 i,j

if (i,1) = 1 and (1,j) = 1 then set (i,j) ← 1

在这种情况下没有变化。



i \ j	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

在此步骤结束时，该矩阵称为 R^1。

传递闭包

步骤 2：

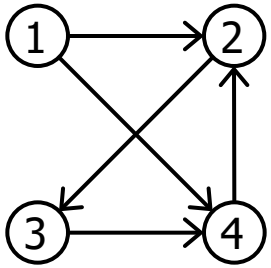
- 选择第 2 行和第 2 列
- 对所有 i,j

if (i,2) = 1 and (2,j) = 1 then set (i,j) ← 1

注意：

(1,2) == (2,3) == 1 → 设置 (1,3) ← 1

(4,2) == (2,3) == 1 → 设置 (4,3) ← 1



j

i	1	2	3	4
1	0	1	**1**	1
2	0	0	1	0
3	0	0	0	1
4	0	1	**1**	0

在此步骤结束时，该矩阵称为 R^2。

传递闭包

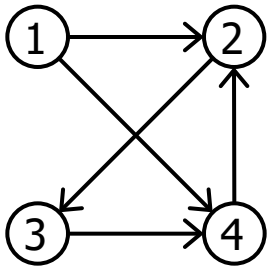
步骤 3：

- 选择第 3 行和第 3 列
  - 对所有 i, j

if (i,3) = 1 and (3,j) = 1 then set (i,j) ← 1

注意：

(1,3) == (3,4) == 1 → 设置 (1,4) ← 1
(2,3) == (3,4) == 1 → 设置 (2,4) ← 1
(4,3) == (3,4) == 1 → 设置 (4,4) ← 1



j

i	1	2	3	4
1	0	1	1	**1**
2	0	0	1	**1**
3	0	0	0	1
4	0	1	1	**1**

在此步骤结束时，该矩阵记为 R^3。

传递闭包

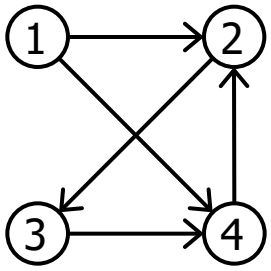
步骤 4：

- 选择第 4 行和第 4 列
- 对所有 i,j

if (i,4) = 1 and (4,j) = 1 then set (i,j) ← 1

注意：

(2,4) == (4,2) == 1 → 设置 (2,2) ← 1
(3,4) == (4,2) == 1 → 设置 (3,2) ← 1
(3,4) == (4,3) == 1 → 设置 (3,3) ← 1



j

i	1	2	3	4
1	0	1	1	1
2	0	**1**	1	1
3	0	**1**	**1**	1
4	0	1	1	1

在这一步结束时，该矩阵称为 R^4。它是图 G 的“传递闭包”。在单元格 (i,j) 中出现 1 表示在 G 中存在从 i 到 j 的路径。

沃舍尔算法

- 也许关于这个算法最棒的就是它的简单性

```
Warshall(R[1..n, 1..n]) 对 k ← 1 到 n { 对 i ← 1 到 n {
对 j ← 1 到 n { 如果 ( R[i,k] == R[k,j] == 1 ) { 设为
R                                          [   ] ← 1i,j } } } }
```

效率：？

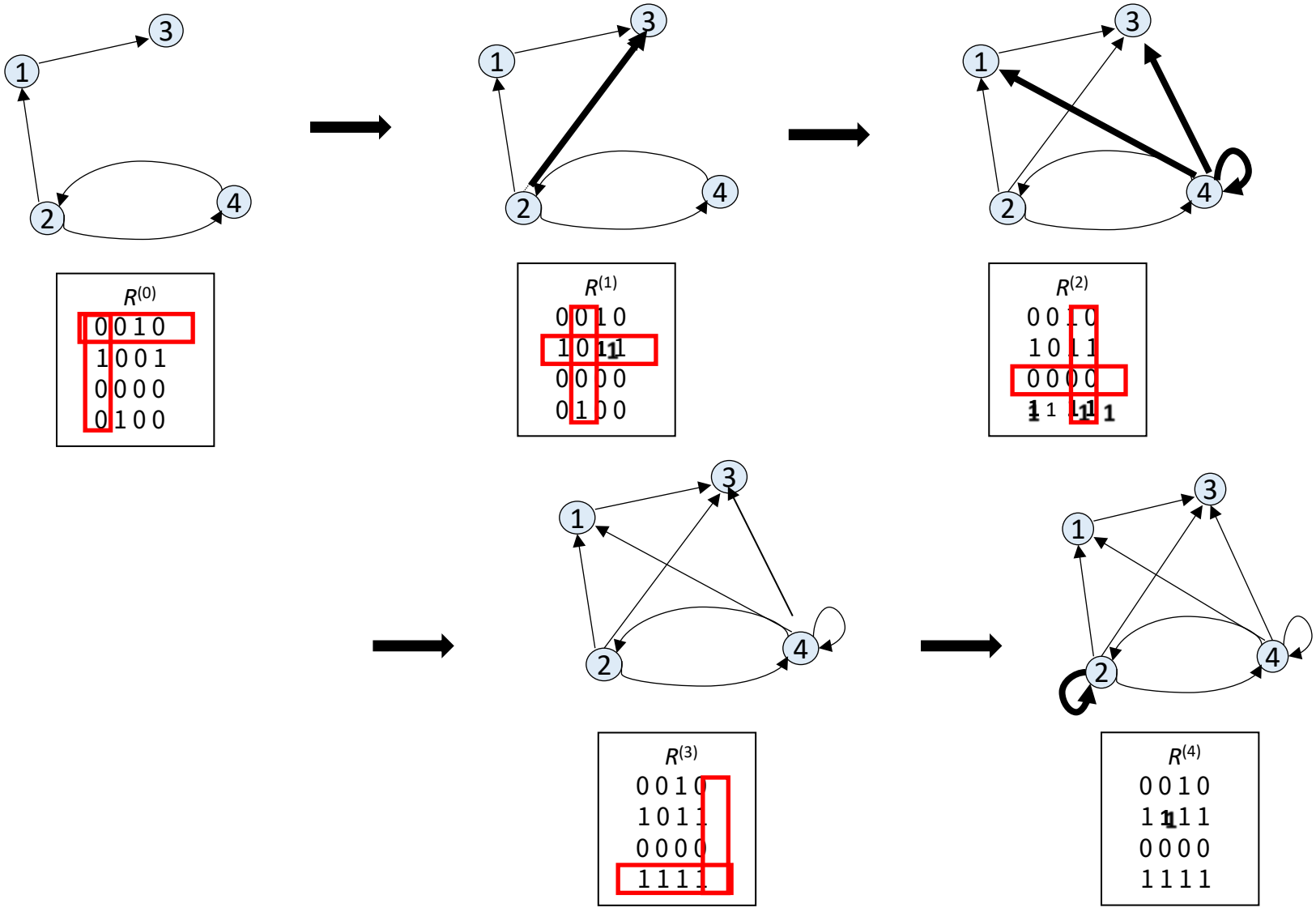
为什么这是动态规划?

- 在第 *k*-次迭代:
  - 该算法对每一对顶点*i, j*判断是否存在一条仅允许顶点 1,...,*k* 作为中间顶点的从*i* 到 *j* 的路径 作为 中间点

- 所以：它从更简单的子问题中找到路径

- 还通过在进行过程中记录到的矩阵自底向上生成结果

另一个示例


1
2
3
4
$R^{(0)}$
0 0 1 0
1 0 0 1
0 0 0 0
0 1 0 0
$R^{(1)}$
0 0 1 0
1 0 1 1
0 0 0 0
0 1 0 0
$R^{(2)}$
0 0 1 0
1 0 1 1
0 0 0 0
1 1 1 1
$R^{(3)}$
0 0 1 0
1 0 1 1
0 0 0 0
1 1 1 1
$R^{(4)}$
0 0 1 0
1 1 1 1
0 0 0 0
1 1 1 1

动态规划：所有点对最短路径（第8章）

所有顶点对最短路径

- 问题：
  - 给定一个有向加权图 G，含有 n 个顶点，求从任意顶点 v_i 到任意其他顶点 v_j 的最短路径，针对所有 $1 \le (i,j) \le n$

- 注意：该问题通常用邻接矩阵来表示图
  图的邻接矩阵表示法

- 应用：该问题在许多应用中出现
  ——尤其在电脑游戏中，很有用以在规划移动之前找到最短路径。

Floyd 算法

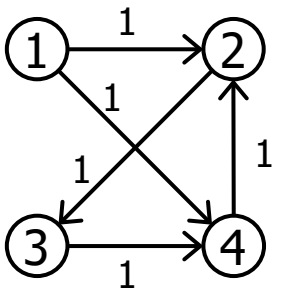
- 类似 Warshall 算法，但有所不同：
  - 在初始图的每条边上添加权重（或代价）
  - 当不存在边时，权重为 ∞
    - “从这里还到不了那里”（暂时）
  - 将对角线上的权重设为 0
    - 从一个顶点到其自身的最短路径应为0

Floyd 算法

- 以及真正的关键变动：
  - Warshall 算法是这样说的：
    - 如果 (i,k) == (k,j) == 1 那么设置 (i,j) ← 1
    - 也就是说：如果你能从 i 到 k，并且从 k 到 j，那么现在你就能从 i 到 j
  - ...但对于 Floyd，我们会这样说：
    - 如果 (i,k) + (k,j) < (i,j) 则设置 (i,j) ← (i,k) + (k,j)
    - 即 如果 i-k-j 的代价小于目前已知的从 i 到 j 的代价，则更新已知的最短路径"

弗洛伊德算法

- 图的初始表示



j

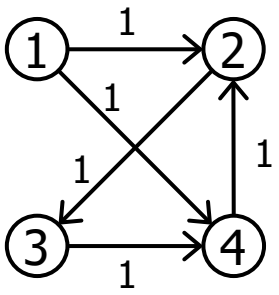
i	1	2	3	4
1	0	1	∞	1
2	∞	0	1	∞
3	∞	∞	0	1
4	∞	1	∞	0

弗洛伊德算法

第 1 步：

- 选择第 1 行和第 1 列
- 对所有 i,j

如果 (i,1) + (1,j) < (i,j) 则将 (i,j) 设为 ← (i,1) + (1,j)



i \ j	1	2	3	4
1	0	1	∞	1
2	∞	0	1	∞
3	∞	∞	0	1
4	∞	1	∞	0

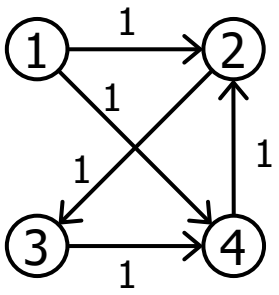
在这种情况下没有变化。

Floyd 算法

步骤 2:

- 选择第 2 行和第 2 列
  - 对所有 i, j

如果 (i,2) + (2,j) < (i,j) 则将 (i,j) 设为 ← (i,2) + (2,j)



i \ j	1	2	3	4
1	0	1	**2**	1
2	∞	0	1	∞
3	∞	∞	0	1
4	∞	1	**2**	0

注意:

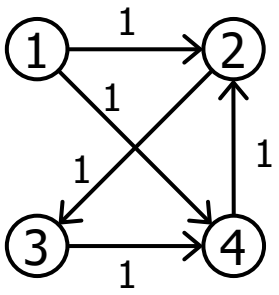
(1,2) + (2,3) < ∞ → set (1,3) ← 2
(4,2) + (2,3) < ∞ → set (4,3) ← 2

Floyd 算法

步骤 3:

- 选择第 3 行和第 3 列
  - 对所有 i,j

if (i,3) + (3,j) < (i,j) then set (i,j) ← (i,3) + (3,j)



j

i	1	2	3	4
1	0	1	2	1
2	∞	0	1	**2**
3	∞	∞	0	1
4	∞	1	2	0

这次只有一个变化……

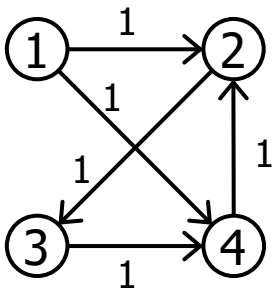
(2,3) + (3,4) < ∞ → 设置 (2,4) ← 2

Floyd 算法

第 4 步：

- 选择第 4 行和第 4 列
  - 对所有 i,j

if (i,4) + (4,j) < (i,j) then set (i,j) ← (i,4) + (4,j)



j

i	1	2	3	4
1	0	1	2	1
2	∞	0	1	2
3	∞	**2**	0	1
4	∞	1	2	0

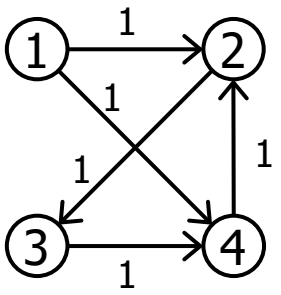
再次说明，只有一次更改……

(3,4) + (4,2) < ∞ → 设置 (3,2) ← 2

弗洛伊德算法

这次我们的解法给出任意 i 到任意 j 的最短路径。

我们可以看到 2、3 或 4 都没有到 1 的路径，算法已为 1→3、 2→4、 3→2 和 4→3 发现了两跳路径，



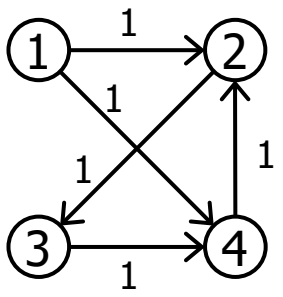
j

i	1	2	3	4
1	0	1	2	1
2	∞	0	1	2
3	∞	2	0	1
4	∞	1	2	0

Floyd 算法

- 最终的矩阵给出任意 i 到任意 j 的最短路径。

- 观察：
  - 你无法从任何顶点到达顶点 1
  - 该算法发现了以下顶点的两跳路径
    $1 \rightarrow 3$、 $2 \rightarrow 4$、 $3 \rightarrow 2$ 和 $4 \rightarrow 3$



i \ j	1	2	3	4
1	0	1	2	1
2	∞	0	1	2
3	∞	2	0	1
4	∞	1	2	0

Floyd 算法（伪代码）

```
Floyd(G[1..n, 1..n])
    for k ← 1 to n {
        for i ← 1 to n {
            for j ← 1 to n {
                cost_thru_k ← G[i,k] + G[k,j]
                if ( cost_thru_k < G[i,j] ) {
                    set G[i,j] ← thru_k
                }
            }
        }
    }
```

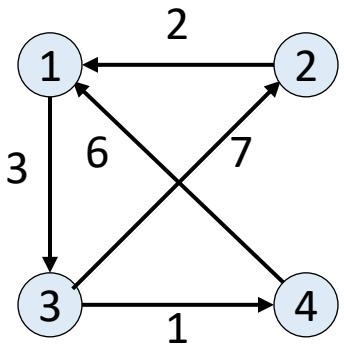
这段中间部分被称为“Warshall 参数”。我们可以调整它以解决各种问题。

效率：？

这个动态规划怎么样?

- (类似 Warshall 的) “子问题” 是找到仅使用顶点 1..k 作为中转点的最短路径
- 每一步都会引入一个新的顶点 (k)
- 在每一步之后，你会得到一个矩阵 D_k，表示通过这些顶点的当前最佳距离

另一个示例



D^0 =

0	∞	3	∞
2	0	∞	∞
∞	7	0	1
6	∞	∞	0

D^1 =

0	∞	3	∞
2	0	**5**	∞
∞	7	0	1
6	∞	**9**	0

D^2 =

0	∞	3	∞
2	0	5	∞
9	7	0	1
6	∞	9	0

D^3 =

0	**10**	3	4
2	0	5	**6**
9	7	0	1
6	**16**	9	0

D^4 =

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0