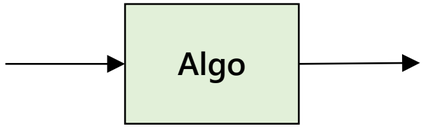
Summary

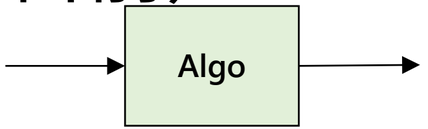
Known graph algorithms (so far)



- Depth first search (DFS)
  - Input: Any graph
  - Output options: DFS order, dead-end order, spanning tree
  - Applicable when: Problem requires visiting all the things (vertices)
- Breadth first search (BFS)
  - Input: Any graph
  - Output options: BFS order, spanning tree
  - Applicable when: Problem requires visiting all the things (vertices)
- Connected components
  - Input: Any graph
  - Output options: Count or Boolean (“is it connected”)
  - Note: Modified DFS/BFS
  - Applicable when: Determining how many “clumps” of vertices there are

总结

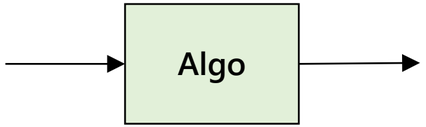
已知图算法（截至目前）



- 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：问题需要访问所有元素（顶点）
- 广度优先搜索（BFS）
  - 输入：任意图
  - 输出选项：BFS顺序、生成树
  - 适用场景：问题需要访问所有元素（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：修改后的DFS/BFS
  - 适用场景：确定有多少个顶点“簇”

Summary

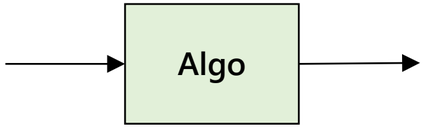
Known graph algorithms (so far)



- Topological sort
  - Input: Directed acyclic graph (DAG)
  - Output: Linear ordering of the vertices
  - Note: We know two algorithms – modified DFS and dec&conq
  - Applicable when: Need to find an order of the vertices
- Minimum spanning tree (MST)
  - Input: Weighted graph
  - Output: Tree
  - Note: We know two algorithms – Prim, Kruskal
  - Applicable when: Want to form the cheapest connected network
- Single-source shortest paths (SSSP)
  - Input: Weighted graph + starting vertex
  - Output options: “Lengths” array, shortest-path tree (aka “prev” array)
  - Note: Dijkstra
  - Applicable when: Looking for shortest path from a particular vertex to all others

总结

已知的图算法（截至目前）



- 拓扑排序
  - 输入：有向无环图（DAG）
  - 输出：顶点的线性排序
  - 注意：我们已知两种算法——改进的深度优先搜索（DFS）和分治法
  - 适用场景：需要找出顶点的一个顺序时
- 最小生成树（MST）
  - 输入：带权图
  - 输出：树
  - 注意：我们知道两种算法——Prim、Kruskal
  - 适用场景：希望构建成本最低的连通网络
- 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（又称“prev”数组）
  - 注意：Dijkstra
  - 适用情况：寻找从某一特定顶点到其他所有顶点的最短路径

Lecture 10

COMP 3760

Dynamic programming

Text chapter 8

第10讲

COMP 3760

动态规划

教材第8章

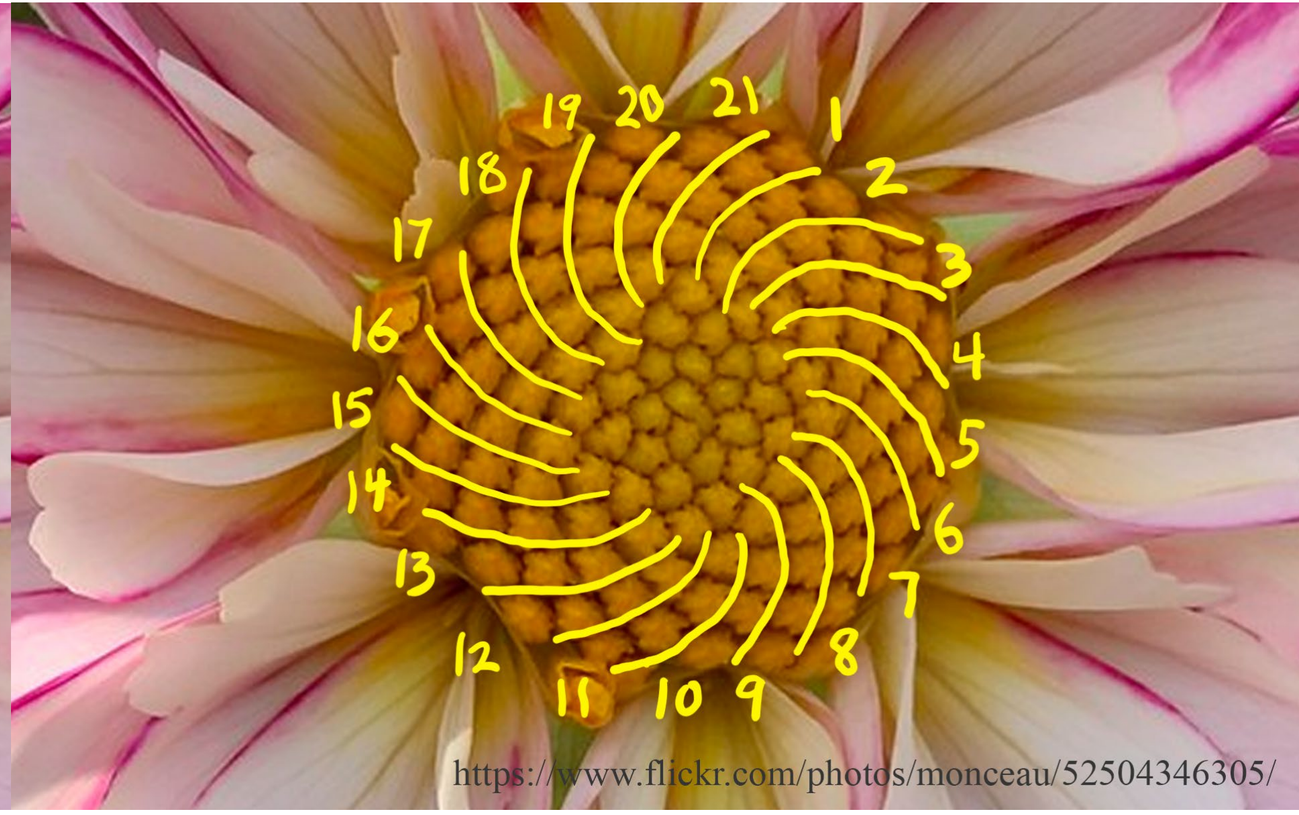

https://www.flickr.com/photos/monceau/52504346305/


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


13
1
12
2
11
3
10
4
9
5
8
6
7
https://www.flickr.com/photos/monceau/52504346305/


13
1
12
2
11
3
10
4
9
5
8
6
7
https://www.flickr.com/photos/monceau/52504346305/


19 20 21 1
18 2
17 3
16 4
15 5
14 6
13 7
12 8
11 10 9
https://www.flickr.com/photos/monceau/52504346305/


19 20 21 1
18 2
17 3
16 4
15 5
14 6
13 7
12 8
11 10 9
https://www.flickr.com/photos/monceau/52504346305/


8
1
2
7
6
3
5
4
https://www.flickr.com/photos/monceau/52504346305/


8
1
2
7
6
3
5
4
https://www.flickr.com/photos/monceau/52504346305/

Hello, dear old friends:
The Fibonacci numbers

- 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)

- How many can we compute?

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)

- 我们能计算多少个?

DEMO

THE CLASSIC RECURSIVE ALGORITHM:

fib (n):

```
    if n < 2
        return n      //i.e. fib(0) is 0 and fib(1) is 1
    else
        return  fib(n-1) + fib(n-2)
```

DEMO

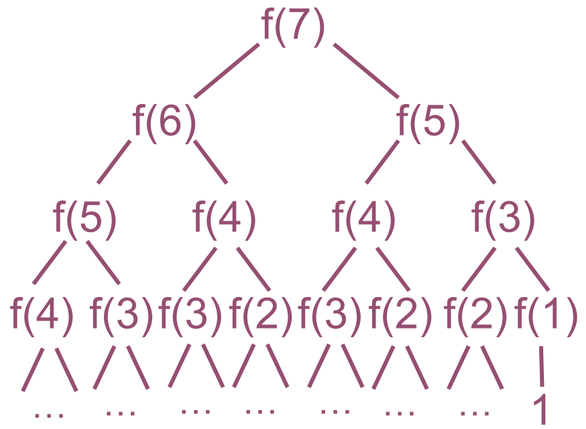
经典递归算法：

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)

Fibonacci numbers: Why you so slow?

```
fib (n):
    if n < 2
        return n
    else
        return  fib(n-1) + fib(n-2)
```

Execution tree:

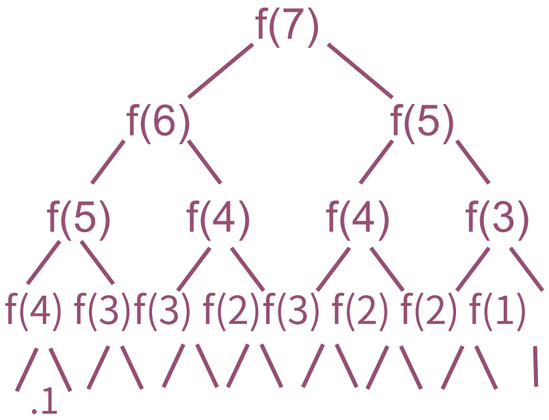


F(n) takes exponential time to compute.

斐波那契数列：为什么你这么慢?

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

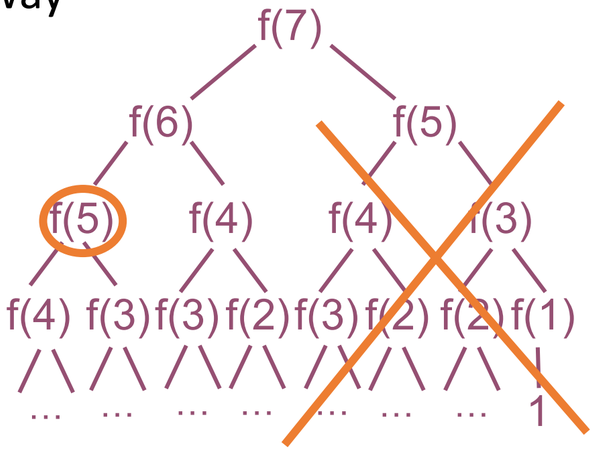
执行树：

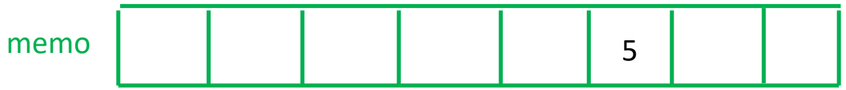


计算 F(n) 需要指数时间。

Space-time trade-off

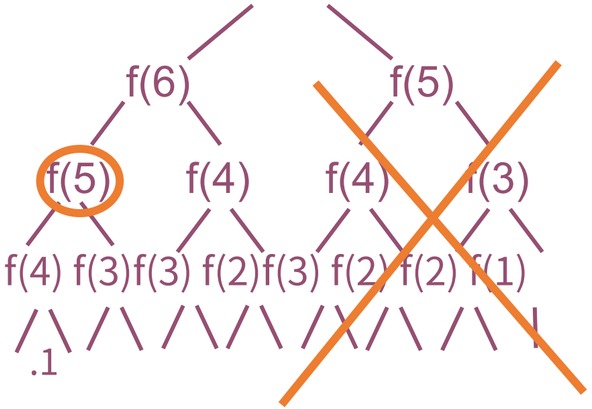
- Augment the algorithm by *remembering* the results you get along the way





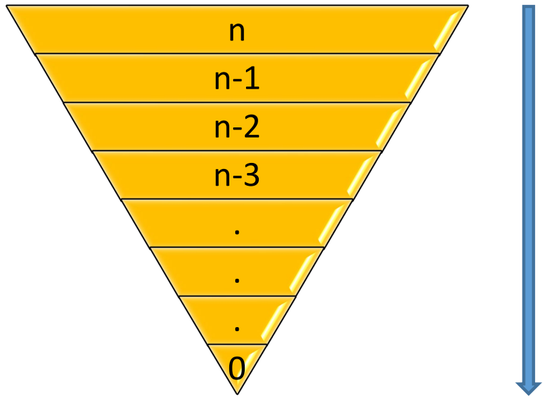
时空权衡

- 通过记住结果来增强算法
  你在路上相处f(7)



Fibs, top-down

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



top-down (Recursive)

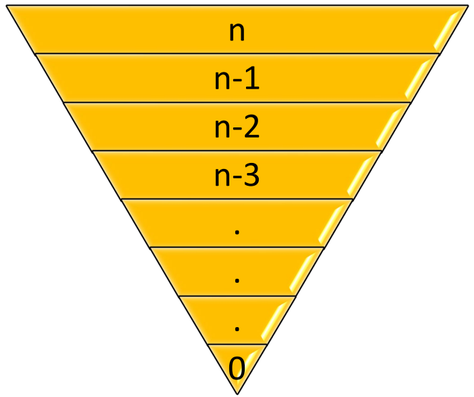
memo	0	1	1	. . .	*fib*(*n*-2)	*fib*(*n*-1)	*fib*(*n*)

Efficiency:

- time: O(n)
- space: Needs an array size O(n)

斐波那契数列，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）

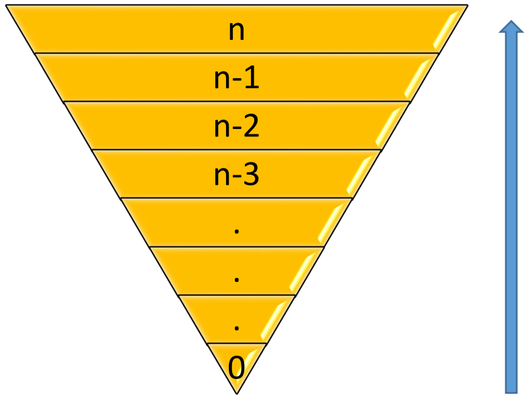
memo	0	1	1	. . .	*fib*(*n*-2)	*fib*(*n*-1)	*fib*(*n*)

效率：

- 时间：O(n)
- 空间：需要一个大小为 O(n) 的数组

Fibs, bottom-up

```
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]
}
```



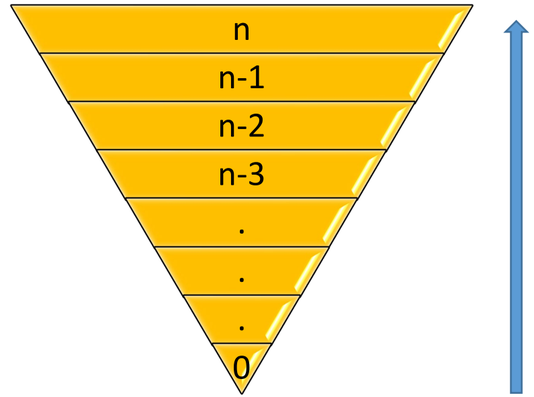
bottom-up

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

Efficiency:
- time: O(n)
- space: Needs an array size O(n)

斐波那契数列，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]
}
```



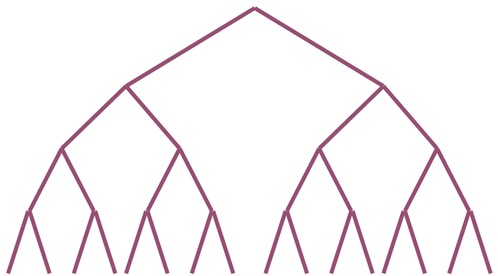
自底向上

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

Dynamic programming

- Key point: *remembering* recursively-defined solutions to sub-problems and using them to solve the problem
- Store solutions to sub-problems for possible reuse.
- A good idea if many of the sub-problems are repeats
- DP is ***not*** divide-and-conquer
  - Both d&c and dp use recursion (at least conceptually)



动态规划

- 关键点：记住子问题的递归定义解，并利用它们来解决原问题
- 存储子问题的解，以便可能重复使用。
- 如果许多子问题是重复的，这是一个好主意。
- 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）。

Dynamic programming overview

- **Step 1:**
  - Decompose problem into smaller, equivalent sub-problems
- **Step 2:**
  - Express solution in terms of sub-problems
- **Step 3:**
  - Use table to compute optimal value bottom-up
- **Step 4:**
  - Find optimal solution based on steps 1-3

动态规划概述

- **步骤 1：**
  - 将问题分解为更小的、等价的子问题
- **步骤 2：**
  - 用子问题表达解决方案
- **步骤 3：**
  - 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

Dynamic programming examples

- Fibonacci numbers
- Robot Coin Collecting
- Transitive Closure (Warshall)
- All Pairs Shortest Paths (Floyd)

动态规划示例

- 斐波那契数
- 机器人硬币收集
- 传递闭包（Warshall）
- 所有结对最短路径（Floyd）

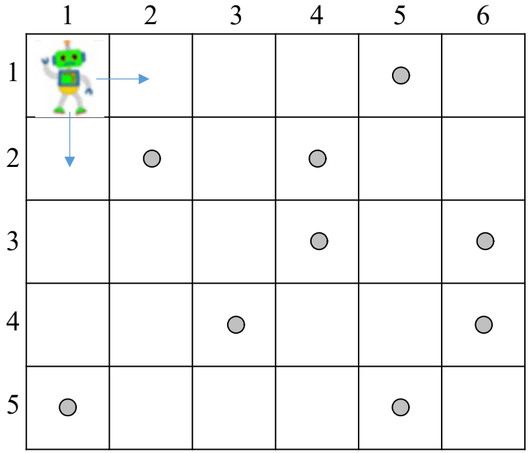
Dynamic Programming: Coin-collecting Robot

(Chapter 8)

动态规划：收集硬币的机器人(第8章)

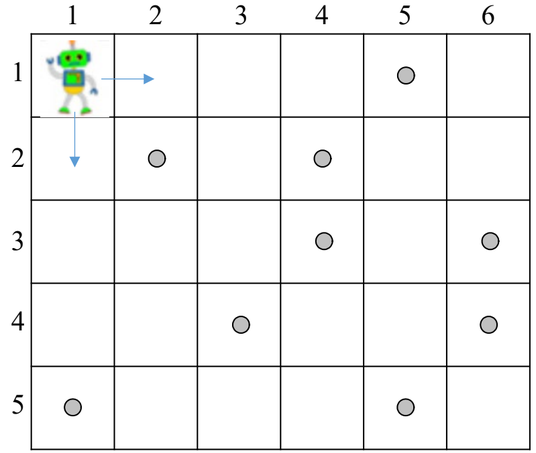
Coin-collecting robot

Several coins are placed in cells of an *n*×*m* board. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. The robot can only move *right* or *down*.



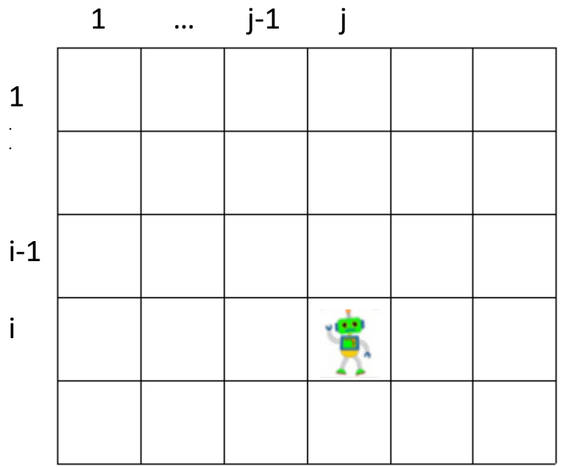
收集硬币机器人

若干枚硬币放置在一个*n*×*m*棋盘的格子中。位于棋盘左上角的一个机器人需要收集尽可能多的硬币，并将它们带到右下角的格子。机器人只能向右或向下移动。



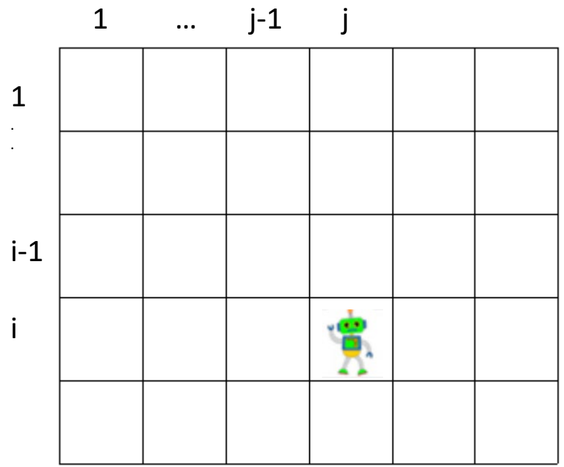
Solution

- Let F(i,j) be the largest number of coins the robot can collect and bring to cell (i,j) in the *ith* row and *j*th column.



解法

- 设 F(i,j) 表示机器人能够收集并带到第 i行第*j*列单元格的最大硬币数。

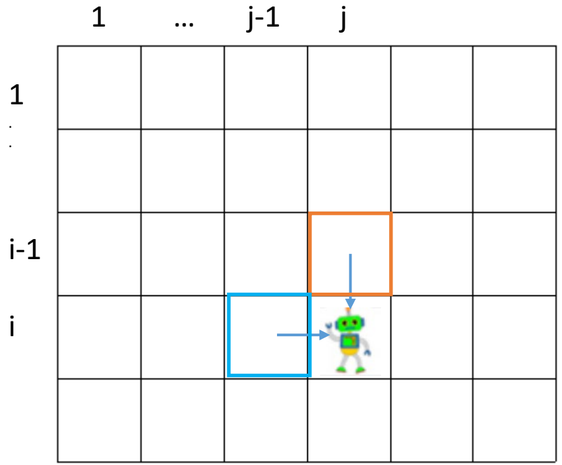


Solution

How many coins could the robot bring to cell (i,j)?

If it comes from the left → F(i, j-1)

If it comes from above → F(i-1, j)

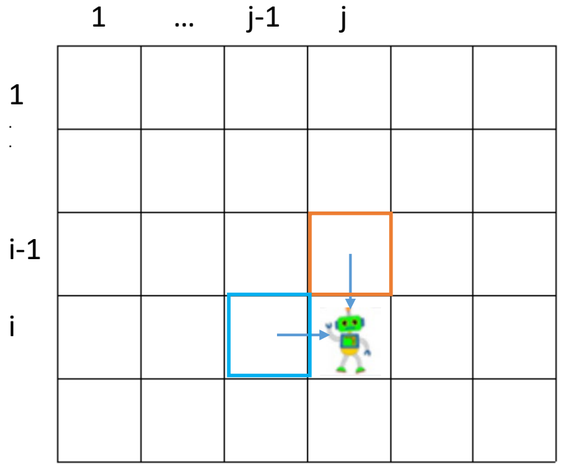


解决方案

机器人最多能将多少枚硬币带到单元格 (i,j)?

如果它从左侧来 → F(i, j-1)

如果它来自上方 → F(i-1, j)

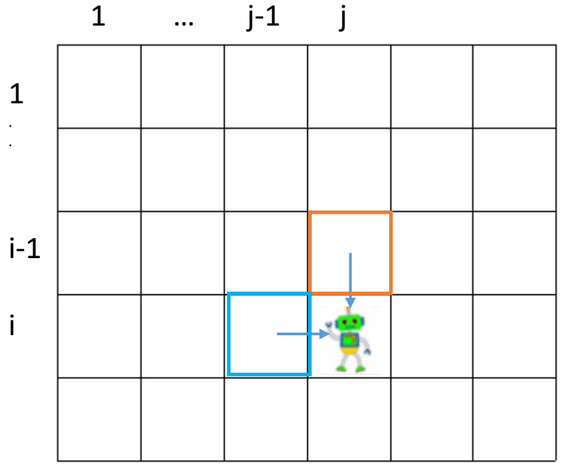


Solution

Recursive definition:

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

where $c_{ij} = 1$ if there is a coin in cell (i,j), and $c_{ij} = 0$ otherwise

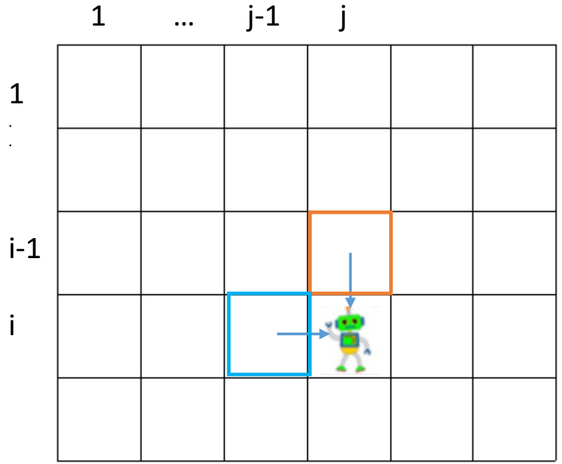


解决方案

递归定义：

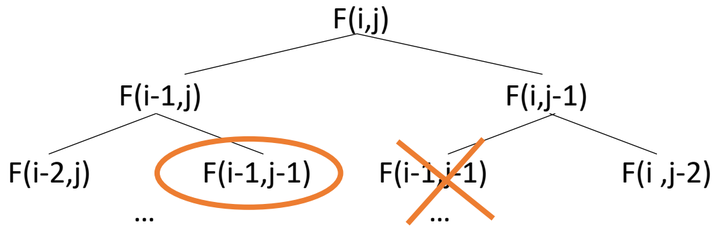
$$F(i, j) = \max\{F(i\text{-}1, j),\ F(i, j\text{-}1)\} + c_{ij} \quad \text{for } 1 \le i \le n,\ 1 \le j \le m$$

其中 $c_{ij} = 1$ 如果格子（i,j）中有硬币，否则为 $c_{ij} = 0$



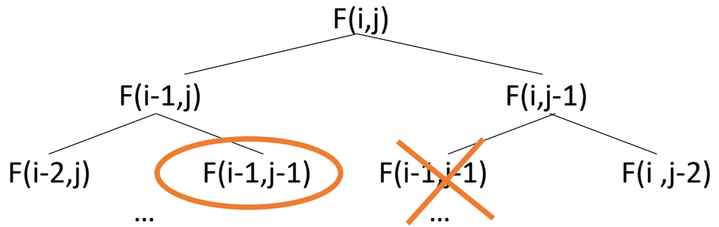
Solution (cont.)

- F(*i*, *j*) = max{F(*i*-1, *j*), F(*i*, *j*-1)} + c_{ij}
- F(0, *j*) = 0 for 1 ≤ *j* ≤ *m* and F(*i*, 0) = 0 for 1 ≤ *i* ≤ *n*.



解法（续）

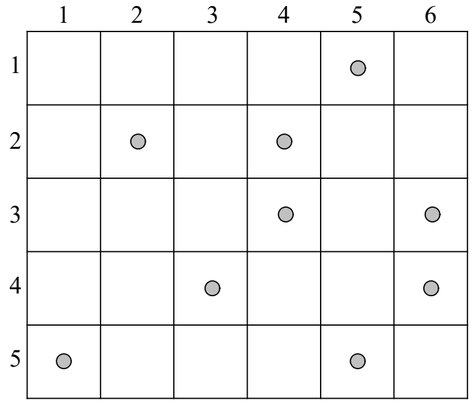
- F(*i*, *j*) = max{F(*i*-1, *j*), F(*i*, *j*-1)} + cij
- F(0, *j*) = 0 对于 1 ≤ *j* ≤ *m* 且 F(*i*, 0) = 0 对于 1 ≤ *i* ≤ n。



Solution (cont.)

Bottom-up calculation

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

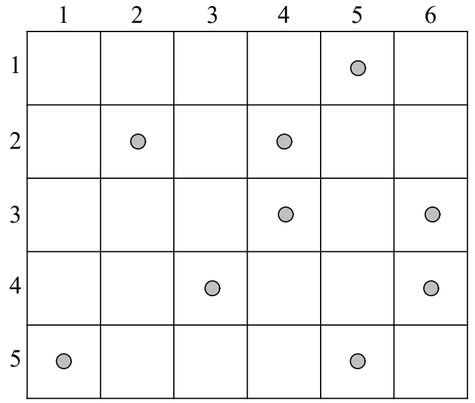


	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

解答（续）

自底向上计算

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



	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**

Robot Coin Collection

```
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]
```

Complexity? Θ(*nm*) time, Θ(*nm*) space

机器人收集硬币

```
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？Θ(*nm*) 时间，Θ(*nm*) 空间

Dynamic programming TL/DR

- Understand the problem
- Make a recursive definition of the problem
  - What are the *subproblems*?
  - How are the subproblems *related*?
- Decide how to store the results of subproblems
- Algorithm to calculate/fill in the data structure

动态规划极简概述

- 理解问题
- 为问题建立一个递归定义
  - 什么是子问题?
  - 子问题之间如何相互关联?
- 确定如何存储子问题的结果
- 用于计算/填充数据结构的算法

Dynamic Programming: Transitive Closure

(Chapter 8)

动态规划：传递闭包

（第8章）

Transitive Closure

- What nodes are reachable from other nodes?
- Problem:
  - given a directed unweighted graph G with n vertices, find all paths that exist from vertices v_i to v_j, for all $1 \leq (i, j) \leq n$

- Note: this problem is always solved with an adjacency matrix graph representation

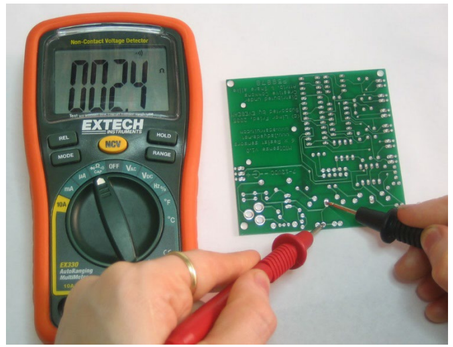
传递闭包

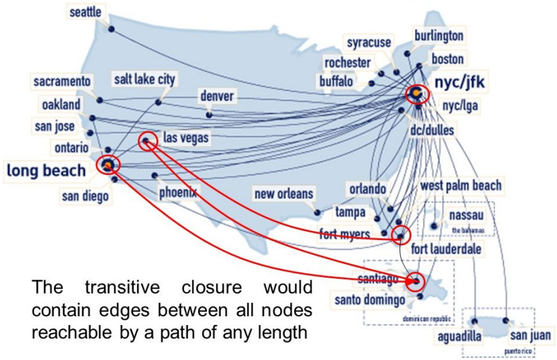
- 从一个节点可以到达哪些其他节点?
- 问题：
  - 给定一个具有 n 个顶点的有向无权图 G，找出从顶点 v_i 到 v_j 的所有可能路径，
    对于所有 $1 \leq (i, j) \leq n$

- 注意：此问题始终使用邻接矩阵图表示法来解决

Transitive Closure

- Applications:
  - Testing digital circuits, reachability testing

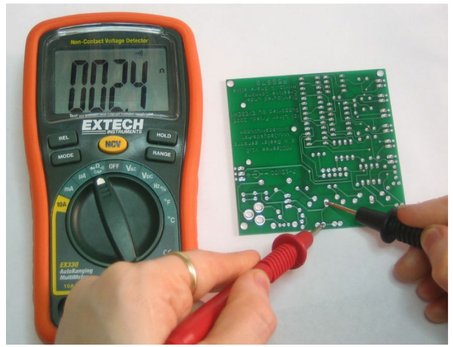


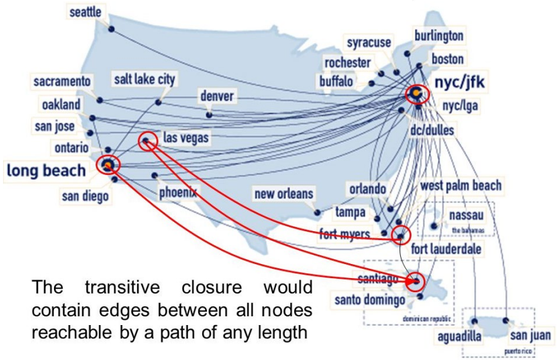


The transitive closure would contain edges between all nodes reachable by a path of any length

传递闭包

- 应用：
  - 测试数字电路，可达性测试

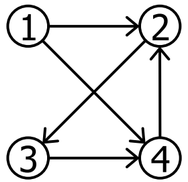




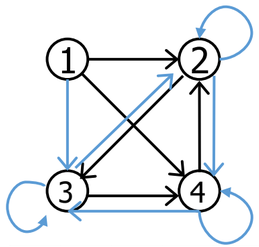
The transitive closure would contain edges between all nodes reachable by a path of any length

Transitive Closure

- Idea of algorithm:
  - Create a new graph where every *edge* represents a *path* in the original



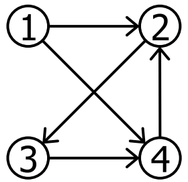
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0



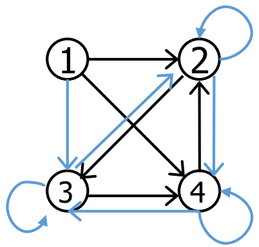
	1	2	3	4
1	0	1	1	1
2	0	1	1	1
3	0	1	1	1
4	0	1	1	1

传递闭包

- 算法思想：
  - 创建一个新图，其中每条边代表原图中的一条路径在原始图中



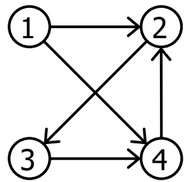
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0



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

Transitive Closure example

- Consider the graph below, and its corresponding adjacency matrix ...

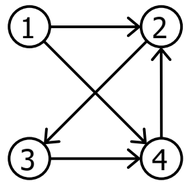


	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

- We call this initial matrix R^0.
  - For convenience here we are using a 1-based array: A[1..n][1..n]

传递闭包示例

- 考虑下面的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

- 我们将这个初始矩阵称为 R^0。
  - 为了方便起见，这里我们使用基于1的数组：A[1..n][1..n]

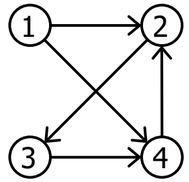
Transitive Closure

Step 1:

- select row 1 and column 1
- for all i,j

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

In this case there are no changes.



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

At the end of this step this matrix is known as R^1.

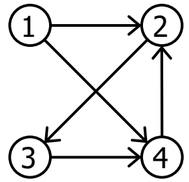
传递闭包

步骤 1：

- 选择第 1 行和第 1 列
- 对于所有 i,j

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

本例中没有变化。



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

此步骤结束时，该矩阵称为 R^1。

Transitive Closure

Step 2:

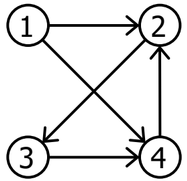
- select row 2 and column 2
- for all i,j

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

Notice:

(1,2) == (2,3) == 1 → set (1,3) ← 1

(4,2) == (2,3) == 1 → set (4,3) ← 1



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

At the end of this step this matrix is known as R^2.

传递闭包

步骤 2：

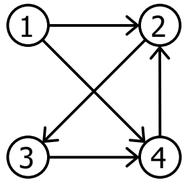
- 选择第 2 行和第 2 列
  - 对于所有 i,j

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



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

此步骤结束时，该矩阵称为 R^2。

Transitive Closure

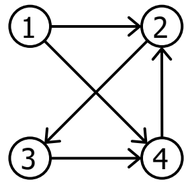
Step 3:

- select row 3 and column 3
- for all i,j

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

Notice:

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



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

At the end of this step this matrix is known as R^3.

传递闭包

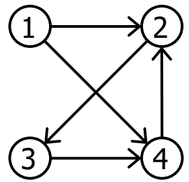
步骤 3：

- 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



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

此步骤结束时，该矩阵称为 R^3。

Transitive Closure

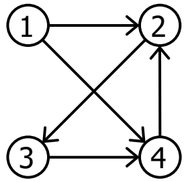
Step 4:

- select row 4 and column 4
- for all i,j

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

Notice:

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

At the end of this step this matrix is known as R^4. It is the "Transitive Closure on G". The existence of a one in cell (i,j) tells us that there exists a path from i to j in G.

传递闭包

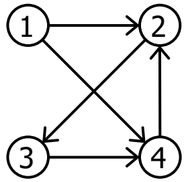
步骤 4：

- 选择第 4 行和第 4 列
  - 对于所有 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

此步骤结束时，该矩阵称为 R^4。它是“G 上的传递闭包”。单元格 (i,j) 中存在一个 1 表明在 G 中存在一条从 i 到 j 的路径。

Warshall's algorithm

- Maybe the best thing about this algorithm is its simplicity

```
Warshall(R[1..n, 1..n])
    for k ← 1 to n {
        for i ← 1 to n {
            for j ← 1 to n {
                if ( R[i,k] == R[k,j] == 1 ) {
                    set R[i,j] ← 1
                }
            }
        }
    }
```

Efficiency: ?

沃舍尔算法

- 也许这个算法最出色的地方在于它的简洁性

```
Warshall(R[1..n, 1..n]) for k ← 1 到 n { for i ← 1 到
n { for j ← 1 到 n { if ( R[i,k] == R[k,j] == 1 ) { 设置
R                                    [    ] ← 1i,j } } } }
```

效率：？

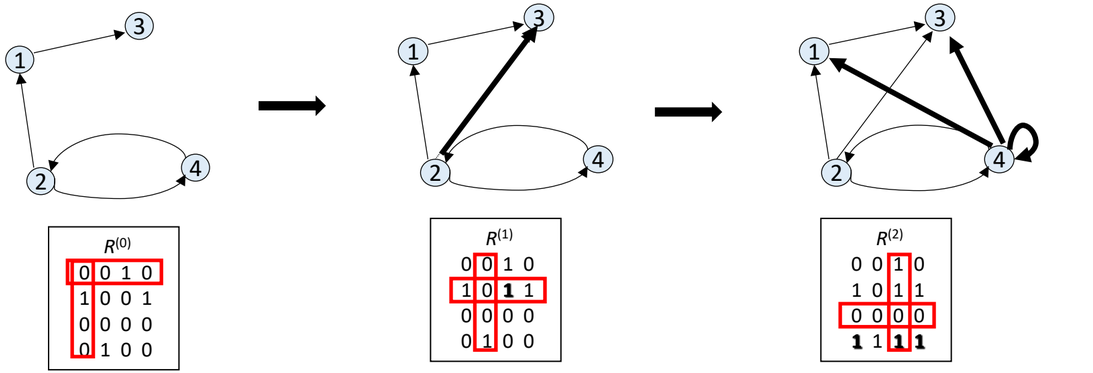
Why is this Dynamic Prog?

- On the *k*-th iteration:
  - The algorithm determines for every pair of vertices *i, j* if a path exists from *i* and *j* with just vertices 1,…,*k* allowed as intermediate
- So: It finds the paths from simpler subproblems
- Also produces the result bottom-up from a matrix recording as you go

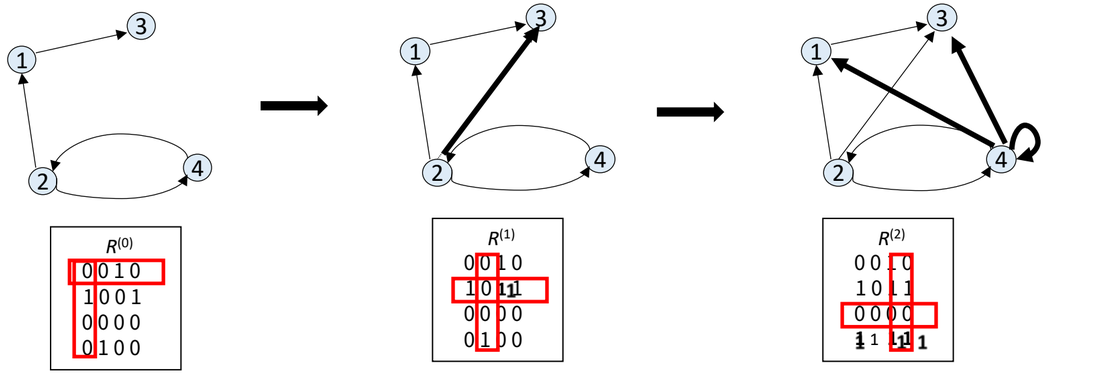
为什么这是动态规划?

- 在第 *k*-次迭代中：
  - 算法会判断每一对顶点 *i, j* 是否存在一条从 *i* 到 *j* 的路径，且该路径仅允许使用顶点 1,…,*k* 作为 中间顶点
- 因此：它通过更简单的子问题来寻找路径
- 同时通过记录过程中的矩阵自底向上生成结果

Another Example



另一个示例



Dynamic Programming: All-pairs shortest paths

(Chapter 8)

动态规划：所有节点对的最短路径(第8章)

All-pairs shortest paths

- Problem:
  - Given a directed weighted graph G with n vertices, find the shortest path from any vertex v_i to any other vertex v_j, for all $1 \leq (i,j) \leq n$
- Note: this problem is always solved with an adjacency matrix graph representation
- Applications: This problem occurs in lots of applications – notably in computer games, where it is useful to find shortest paths before planning movement.

所有点对最短路径

- 问题：
  - 给定一个具有 n 个顶点的有向加权图 G，对所有 $1 \leq (i,j) \leq n$，求从任意顶点 v_i 到任意其他顶点 v_j 的最短路径
- 注意：此问题通常使用邻接矩阵图表示法来解决
- 应用：此问题在许多实际应用中都会出现
  ——尤其是在计算机游戏中，在规划移动之前找到最短路径非常有用。

Floyd's algorithm

- Like Warshall's algorithm, but different:
  - Add weight (or cost) to each edge in the initial graph
  - When no edge exists the weight is ∞
    - "You can't get there from here" (yet)
  - Set the weights on the diagonal to be 0
    - The shortest path from a vertex to itself should be 0

Floyd算法

- 与Warshall算法类似，但有所不同：
  - 在初始图的每条边上添加权重（或代价）
  - 当边不存在时，其权重为∞
    - "你无法从此处到达那里"（还不能）
  - 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's algorithm

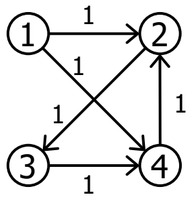
- And the real key change:
  - Warshall's algorithm says this:
    - if (i,k) == (k,j) == 1 then set (i,j) ← 1
    - i.e. If you can get from i to k and from k to j, then now you can get from i to j
  - …but for Floyd's we will say this:
    - if (i,k) + (k,j) < (i,j) then set (i,j) ← (i,k) + (k,j)
    - i.e. If i-k-j costs less than the (so far) best known path from i to j, then update the best known path"

Floyd算法

- 而真正的关键变化是：
  - Warshall算法指出：
    - 如果(i,k) == (k,j) == 1 ，则设置(i,j)← 1
    - 即：如果可以从i到达k，并且从k到达j，那么现在就可以从i到达j
  - ……但对于Floyd算法，我们将这样说：
    - 如果 (i,k) + (k,j) < (i,j) ，则设置 (i,j) ← (i,k) + (k,j)
    - 即：如果 i-k-j 的代价小于从 i 到 j 的（目前）已知最优路径，则更新该已知最优路径"

Floyd's algorithm

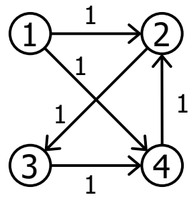
- Initial representation of the graph



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

Floyd算法

- 图的初始表示



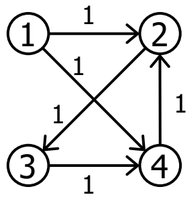
i \ j	1	2	3	4
1	0	1	∞	1
2	∞	0	1	∞
3	∞	∞	0	1
4	∞	1	∞	0

Floyd's Algorithm

Step 1:

- select row 1 and column 1
- for all i,j

  if (i,1) + (1,j) < (i,j) then set (i,j) ← (i,1) + (1,j)



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

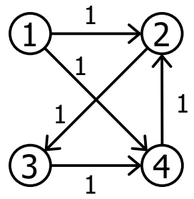
In this case there are no changes.

弗洛伊德算法

步骤 1:

- 选择第 1 行和第 1 列
- 对于所有 i,j

  如果 (i,1) + (1,j) < (i,j)，则设置 (i,j) ← (i,1) + (1,j)



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

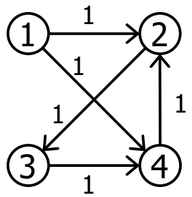
本例中无任何变化。

Floyd's Algorithm

Step 2:

- select row 2 and column 2
- for all i,j

  if (i,2) + (2,j) < (i,j) then set (i,j) ← (i,2) + (2,j)



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

Notice:

(1,2) + (2,3) < ∞ → set (1,3) ← 2

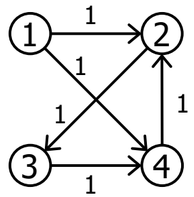
(4,2) + (2,3) < ∞ → set (4,3) ← 2

弗洛伊德算法

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)，则设置 (i,j) ← (i,2) + (2,j)



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

注意:

(1,2) + (2,3) < ∞ → set (1,3) ← 2

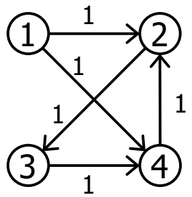
(4,2) + (2,3) < ∞ → set (4,3) ← 2

Floyd's Algorithm

Step 3:

- select row 3 and column 3
- for all i,j

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



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

There is only one change this time ...

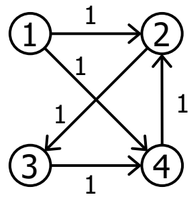
(2,3) + (3,4) < ∞ → set (2,4) ← 2

Floyd算法

步骤3:

- 选择第3行和第3列
- 对所有i,j

  如果(i,3) + (3,j) < (i,j)，则设置(i,j) ← (i,3) + (3,j)



i \ j	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……

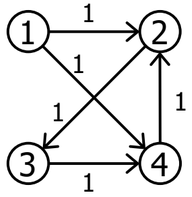
(2,3) + (3,4) < ∞ → 设置 (2,4) ← 2

Floyd's Algorithm

Step 4:

- select row 4 and column 4
- for all i,j

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



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

Again, only one change ...

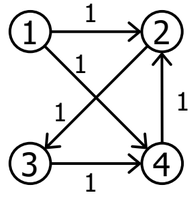
(3,4) + (4,2) < ∞ → set (3,2) ← 2

弗洛伊德算法

步骤 4:

- 选择第 4 行和第 4 列
- 对所有 i,j

  如果 (i,4) + (4,j) < (i,j)，则设置 (i,j) ← (i,4) + (4,j)



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

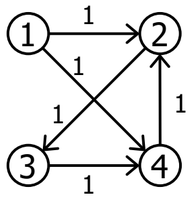
同样，仅有一处更改……

(3,4) + (4,2) < ∞ → 设置 (3,2) ← 2

Floyd's Algorithm

This time our solution gives the shortest paths from any i to any j.

We can see that the none of 2,3, or 4 have paths to 1, and the algorithm has discovered two hop paths for 1→3, 2→4, 3→2, and 4→3,



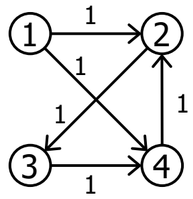
j

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

这一次我们的解法给出了从任意 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，而算法已经找到了 1→3、 2→4、3→2 和 4→3 的两跳路径，

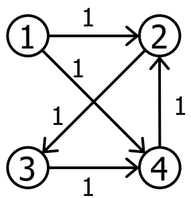


j

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

Floyd's Algorithm

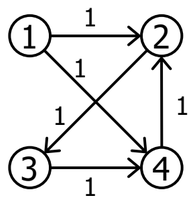
- The final matrix gives the shortest paths from any i to any j.
- Observations:
  - You can't get from anywhere to 1
  - The algorithm discovered two-hop paths for $1 \rightarrow 3$, $2 \rightarrow 4$, $3 \rightarrow 2$, and $4 \rightarrow 3$



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

- 最终的矩阵给出了从任意 i 到任意 j 的最短路径。
- 观察结果：
  - 无法从任何地方到达 1
  - 该算法发现了两跳路径，适用于 $1 \rightarrow 3$、 $2 \rightarrow 4$、 $3 \rightarrow 2$以及$4 \rightarrow 3$



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

Floyd's Algorithm (pseudocode)

```
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
               }
           }
       }
   }
```

This middle section is referred to as the "Warshall Parameter". We can change it around to solve a variety of problems.

Efficiency: ?

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
               }
           }
       }
   }
```

此中间部分被称为“Warshall参数”。我们可以对其进行修改，以解决各种问题。

效率：？

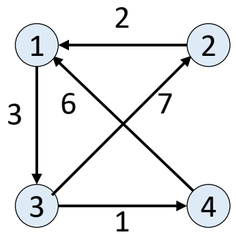
How is this DP?

- (Like Warshall's) the "sub-problem" is that it is finding shortest paths that use vertices 1..k as hopping points
- One new vertex (k) is added into the picture at each step
- After each step, you have a matrix D_k that gives the best (yet) distance through those vertices

这个DP怎么样?

- （类似于Warshall算法）所谓的"子问题"是指寻找使用顶点1..k作为中转点的最短路径
- 每一步都会新增一个顶点（k）进入考虑范围
- 每一步之后，你都会得到一个矩阵D_k，它给出了经过这些顶点的当前最优距离

Another Example



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

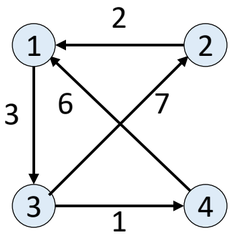
D^3 =

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

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