Analysis and Design of Algorithms

Chapter 9: Greedy Algorithms

School of Software Engineering © Yanling Xu

Change Making Problem

- ***Problem***

Given unlimited amounts of coins of denominations d1 > … > dm ,

*give change for amount n with the **least number of coins**.*

- ***Idea*** *---"Greedy" thinking*

- Take the coin with largest denomination without exceeding the remaining amount of cents, → it reduces the remaining amount the most,
- make the locally best choice at each step.
- ***Example:***

How to make 48 cents of change using coins of

d1=25 , d2=10 , d3=5 , d4=1

so that the total number of coins is the smallest?

Solution: 25, 10, 10, 1, 1, 1

- ***Is the solution globally optimal? Yes***

Change Making Problem

- **General Change-Making Problem**

- *Does the greedy algorithm always give an optimal solution for the general change-making problem?*

- ***Example:***

  d1 = 7c, d2 =5c, d3 = 1c, and n = 10c,

  not always produces an optimal solution.

- ***in fact, for some instances, the problem may not have a solution at all! consider instance d1 = 7c, d2 =5c, d3 = 3c, and n = 11c.***

Greedy Algorithms

- **Greedy Strategy**
  - *A greedy algorithm makes a **locally optimal choice** step by step in the hope that a sequence of such choice will **lead to a globally optimal solution**.*
  - *constructing a solution through a sequence of steps*
  - *each step expanding a partially constructed solution obtained so far,*
  - *until a complete solution to the problem is reached*
  - *The choice made at each step must be:*
  - *Feasible -Satisfy the problem's constraints*
  - *locally optimal -Be the best local choice among all feasible choices*
  - *Irrevocable -Once made, choice can't be changed on subsequent steps.*
  - *Do greedy algorithms always yield optimal solutions?*
    - *not the case -approximation algorithms based on the greedy approach*
    - *Example: change making problem with a denomination set of 7, 5 and 1, and n =10?*

Greedy Algorithms

Applications of the Greedy Strategy

- ***For some problems, yields an optimal solution for every instance.***
- ***For most, does not but can be useful for fast approximations.***
- ***Optimal solutions:***
  - some instances of change making
  - Minimum Spanning Tree (MST)
  - Single-source shortest paths
  - Huffman codes
- ***Approximations:***
  - Traveling Salesman Problem (TSP)
  - Knapsack problem
  - other optimization problems

活动安排问题

- **Problem:**
  - 设有n个活动的集合E={1,2,…,n}，其中每个活动都要求使用同一资源，而在同一时间内只有一个活动能使用这一资源。
  - 每个活动 i 都有一个要求使用该资源的起始时间s_i和一个结束时间f_i,且$s_i < f_i$。如果选择了活动 i，则它在半开时间区间$[s_i, f_i)$内占用资源。
  - 若区间$[s_i, f_i)$与区间$[s_j, f_j)$不相交，则称活动 i与活动 j是相容的。也就是说，当$s_i \geq f_j$或$s_j \geq f_i$时，活动 i与活动 j 相容。
  - 活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合

✓ 是可以用贪心算法有效求解的很好例子。
✓ 该问题要求高效地安排一系列争用某一公共资源的活动。
✓ 贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能兼容地使用公共资源。

活动安排问题

- 算法描述

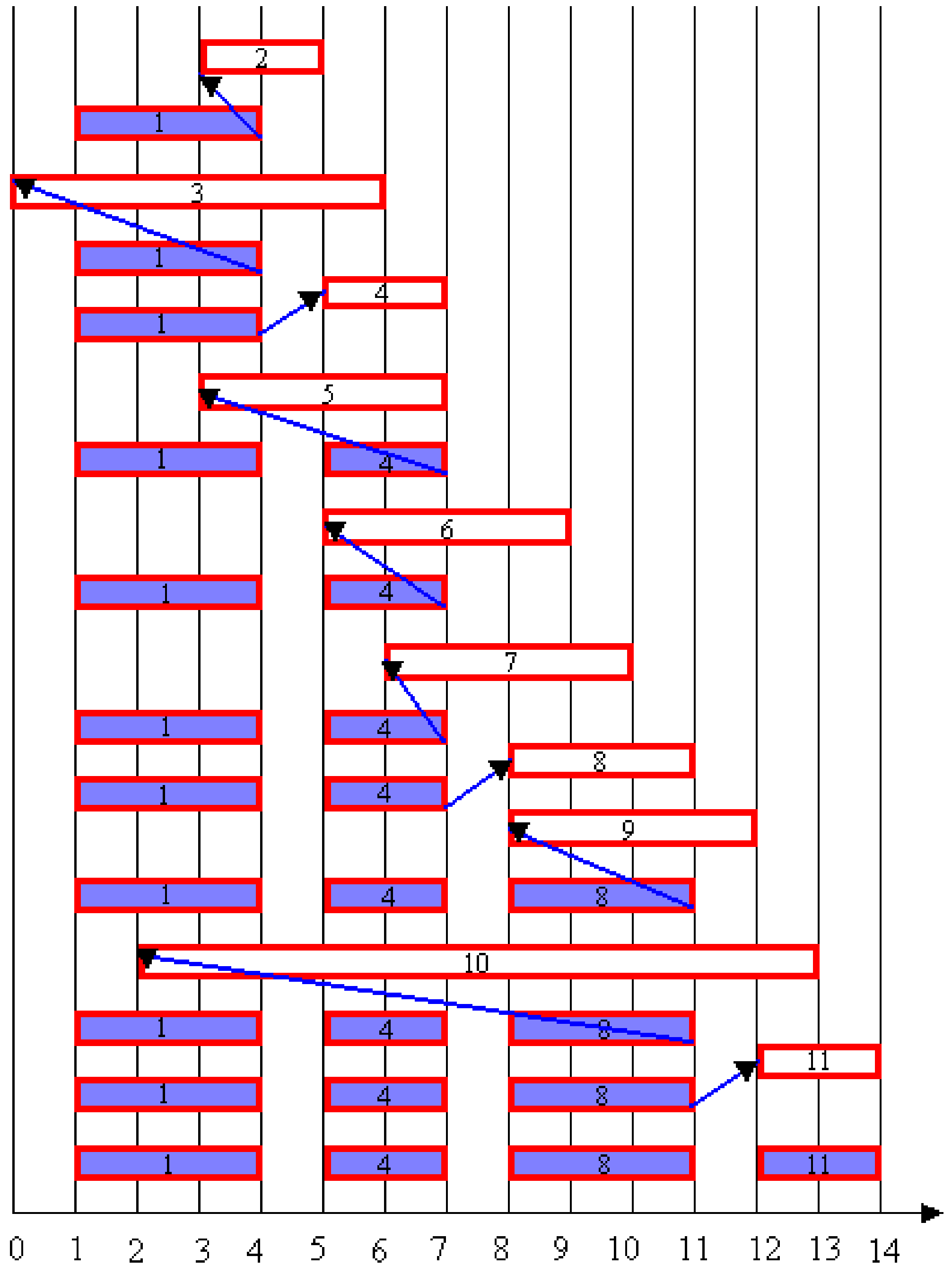
```
template<class Type>
void GreedySelector (int n, Type s[], Type f[], bool A[])
{
      A[1]=true;
      int j=1;
      for (int i=2;i<=n;i++) {
         if (s[i]>=f[j]) { A[i]=true; j=i; }
         else A[i]=false;
         }
}
```

活动安排问题

算法思路

- 各活动的起始时间和结束时间存储于数组*s*和*f* 中且**按结束时间的非减序**排列 $f_1 \le f_2 \le \ldots \le f_n$
- 集合*A*存储所选择的活动，活动 *i* 在集合*A*中，当且仅当A[*i*]的值为true。
- 活动 *i* 与当前已选择的所有活动相容的**充分必要条件是：其开始时间*s*** **s_i不早于最近加入集合A中的活动*j* 的结束时间f_j, 即** $s_i \ge f_j$

活动安排问题



2
1
3
1
4
1
5
1
4
6
1
4
7
1
4
8
1
4
9
1
4
8
10
1
4
8
11
1
4
8
1
4
8
11
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

活动安排问题

算法分析

- 由于输入的活动以其完成时间的非减序排列，所以算法GreedySelector 每次总是选择具有最早完成时间的相容活动加入集合*A*中。
- 按这种方法选择相容活动为未安排活动留下尽可能多的时间。
- 也就是说，该算法的贪心选择的意义是使剩余的可安排时间段极大化，以便安排尽可能多的相容活动。
- 算法GreedySelector的效率极高。当输入的活动已按结束时间的非减序排列，算法只需***O*(*n*)**的时间安排*n*个活动，使最多的活动能相容地使用公共资源。
- 如果所给出的活动未按非减序排列，可以用***O*(*nlogn*)**的时间重排。

活动安排问题

- 数学归纳法证明：贪心算法可以求得活动安排问题的整体最优解
  - 设E={1，2，…，n}为所给的活动集合。由于E中活动按结束时间的非减序排列，故活动1具有最早的完成时间。
  - **S1：** 首先我们要证明活动安排问题有一个最优解以贪心选择开始，即该最优解中包含活动1。
    - 设A⊆ E是所给的活动安排问题的一个最优解，且A中活动也按结束时间非减序排列，A中的第一个活动是活动k。
      - ✓ 若k=1，则A就是一个以贪心选择开始的最优解。
      - ✓ 若k>1，则我们设B=A-{k}∪{1}。由于$f_1 \le f_k$，且A中活动是互为相容的，故B中的活动也是互为相容的。又由于B中活动个数与A中活动个数相同，且A是最优的，故B也是最优的。也就是说B是一个以贪心选择活动1开始的最优活动安排。
    - 因此，我们证明了总存在一个以贪心选择开始的最优活动安排方案。

活动安排问题

- **S2：** 进一步，在作了贪心选择，即选择了活动1后，原问题就简化为对E中所有与活动1相容的活动进行活动安排的子问题。即，若A是原问题的一个最优解，则A’=A-{1}是活动安排问题E’=$\{i \in E: s_i \geq f_1\}$的一个最优解。
  - 事实上，如果我们能找到E’的一个解B’，它包含比A’更多的活动，则将活动1加入到B’中将产生E的一个解B，它包含比A更多的活动。这与A的最优性矛盾。
  - 因此 ，每一步所作的贪心选择都将问题简化为一个更小的与原问题具有相同形式的子问题。——最优子结构性质
  - 对贪心选择次数用数学归纳法即知，贪心算法GreedySelector最终产生原问题的一个最优解。

贪心算法的基本要素

贪心选择性质

- 所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。
  - 动态规划算法：每步所做的选择往往依赖于子问题的解，只有在解出相关子问题后才能作出选择
  - 贪心算法：仅在当前状态下作出最好选择，即局部最优选择，然后再去作出这个选择后产生的相应的子问题，不依赖于子问题的解，
  - 动态规划算法：通常以自底向上的方式解各子问题，
  - 贪心算法：通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

贪心算法的基本要素

- 对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。
  - 首先考察问题的一个整体最优解，并证明可修改这个最优解，使其以贪心选择开始
  - 利用最优子结构性质，证明：做了贪心选择后，原问题简化为规模更小的类似子问题。
  - 用数学归纳法证明，通过每一步做贪心选择，最终可以得到问题的整体最优解。

Example： 活动安排问题中对贪心解即最优解的证明

贪心算法的基本要素

最优子结构性质

- 当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。
  - **在活动安排问题中，其最优子结构性质表现为：**

    若A是对于E的活动安排问题包含活动1的一个最优解，则相容活动集合A'=A—{1}是对于$E' = \{i \in E : s_i \geq f_1\}$ 的活动安排问题的一个最优解

贪心算法的基本要素

贪心算法与动态规划算法的差异

- 贪心算法和动态规划算法的共同点：要求问题具有最优子结构性质
- 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?

贪心算法的一般策略

- **问题的一般特征：**

  问题的解是由*n*个输入的、满足某些事先给定的条件的子集组成。

- **贪心方法**

  - 根据题意，选取一种**度量标准**。然后按照这种度量标准对*n*个输入**排序**，并按序一次输入一个量。
  - 如果这个输入和当前已构成在这种量度意义下的部分最优解加在一起不能产生一个可行解，则不把此输入加到这部分解中。
  - 否则，将当前输入合并到部分解中从而得到包含当前输入的新的部分解。
  - 这一处理过程一直持续到*n*个输入都被考虑完毕，则记入最优解集合中的输入子集构成这种量度意义下的问题的**最优解**

⇨ 贪心方法: 这种能够得到某种**量度意义下的最优解**的**分级处理**方法称为贪心方法

贪心算法的一般策略

- 贪心方法的抽象化控制描述

```
procedure GREEDY(A,n)
      //A(1:n)包含n个输入//
       solution←Φ     //将解向量solution初始化为空//
       for i←1 to n do
            x←SELECT(A)
      //按照度量标准，从A中选择一个输入，其值赋予x并将之从A中删除//
             if FEASIBLE(solution,x)  then
          //判定x是否可以包含在当前解向量中，即是否能共同构成可行解//
                   solution←UNION(solution,x)
                   //将x和当前的解向量合并成新的解向量，并修改目标函数//
             endif
         repeat
       return
   end GREEDY
```

贪心算法解背包问题

- **0-1背包问题：**给定n 种物品和一背包。物品 i 的重量是w_i，其价值为v_i，背包的容量为C。问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?
    - **对每种物品 *i* 只有两种选择，即装入背包或不装入背包，**
    - **不能将物品 *i* 装入背包多次，不能只装入部分的物品 *i***
    - 其形式化描述是：

      给定$c>0, w_i>0, v_i>0, 1 \le i \le n$, 要找出一个$n$元0-1向量$(x_1,x_2,\ldots,x_n)$，使得

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \le c \qquad x_i \in \{0,1\},\ 1 \le i \le n$$

贪心算法解背包问题

- **背包问题：**与0-1背包问题类似，给定n 种物品和一背包。物品 i 的重量是w_i，其价值为v_i，背包的容量为C。**假定将物品i的某一部分x_i放入背包就会得到v_ix_i的效益($0 \le x_i \le 1$, $v_i > 0$)**。问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?

  - **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，$1 \le i \le n$。**
  - 其形式化描述是：

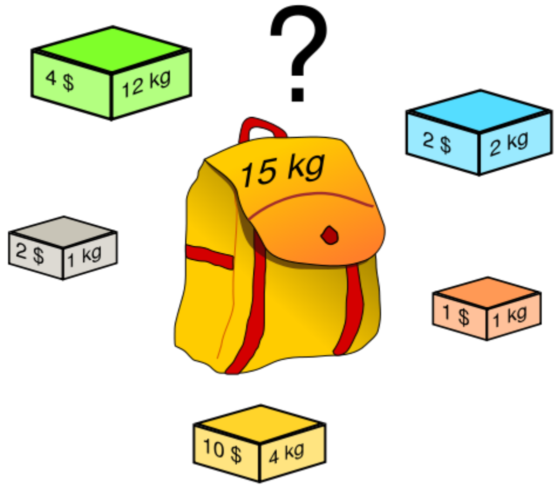
    给定$c>0, w_i>0, v_i>0, 1 \le i \le n$,

    要找出一个n元向量$(x_1,x_2,\ldots,x_n)$,

    使得

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \le c \qquad 0 \le x_i \le 1,\ 1 \le i \le n$$



贪心算法解背包问题

- 两个问题都具有最优子结构性质
  - 0-1背包问题的最优子结构：

    设A是能够装入容量为c的背包的具有最大价值的物品集合，则A_j＝A－{j}是n-1个物品1,2,…,j-1,j+1,…,n可装入容量为c-w_ix_i的背包且具有最大价值的物品集合。

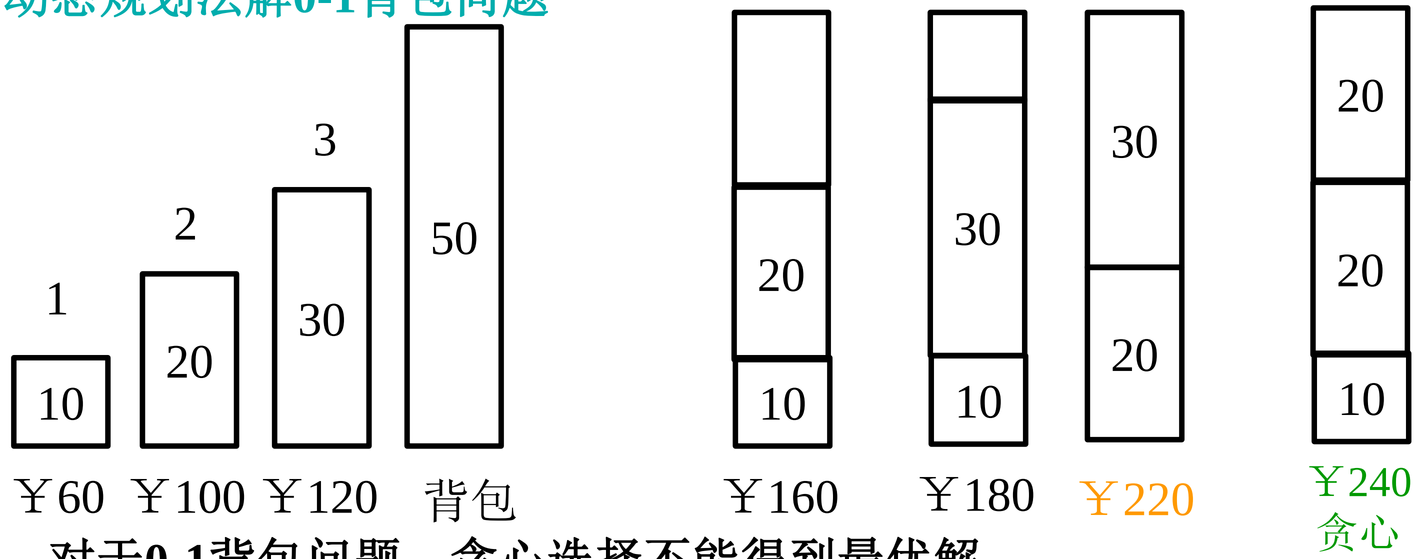
  - 背包问题的最优子结构：

    设问题的最优解包含物品j，则从该最优解中拿出所含的物品j的那部分重量w，剩余的是n-1个原重物品1,2,…, j-1, j+1, …,n及重为w_i-w的物品j可装入容量为c-w的背包且具有最大价值的物品集合。

- **背包问题可以用贪心算法求解， 0-1背包问题不可以用贪心算法求解**

贪心算法解背包问题

- 用动态规划法解0-1背包问题



- **对于0-1背包问题，贪心选择不能得到最优解**

  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。

- **动态规划算法可以有效地解0-1背包问题**
  - 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。

贪心算法解背包问题

- **背包问题：** 给定n种物品和一背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。**假定将物品i的某一部分x_i放入背包就会得到v_ix_i的效益($0\le x_i\le 1$, $v_i>0$)**。问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?

  - **目标：**使装入背包的物品的总效益达到最大。
  - **约束条件：**装入背包的总重量不能超过C
  - 如果所有物品的总重量不超过C，即 $\sum_{i=1}^{n} w_i x_i \le C$，则把所有的物品都装入背包中将获得最大可能的效益值
  - 如果物品的总重量超过了C，则将有物品**不能（部分/全部）**装入背包中。由于$\mathbf{0\le x_i\le 1}$，所以可以把物品的一部分装入背包，故最终背包中可**刚好装入重量为C的若干物品（整体或一部分）**。这种情况下，如果背包没有被装满，则显然不能获得最大的效益值。

贪心算法解背包问题

- 背包问题形式化描述：

  给定$c>0, w_i>0, v_i>0, 1 \le i \le n$,

  要找出一个**n元向量**$(x_1,x_2,\ldots,x_n)$,

  - 目标函数

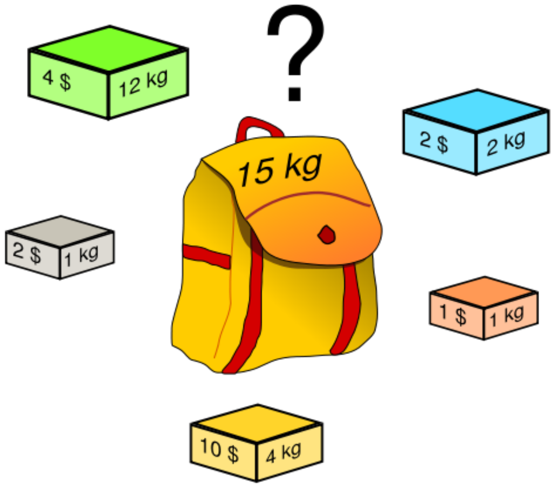
$$\max \sum_{i=1}^{n} v_i x_i$$

  - 约束条件

$$\sum_{i=1}^{n} w_i x_i \le c \qquad 0 \le x_i \le 1,\ 1 \le i \le n$$

  - 可行解：满足上述约束条件的任一$(x_1,x_2,\ldots,x_n)$ 都是问题的一个可行解 ——可行解可能为多个。
  - 最优解：能够使目标函数取最大值的可行解是问题的最优解

  —— 最优解也可能为多个。



贪心算法解背包问题

- ***Example:***

设，n=3，C=20，$(v_1,v_2,v_3) = (25,24,15)$， $(w_1,w_2,w_3) = (18,15,10)$。

可能的可行解有：

	(x_1,x_2,x_3)	$\sum w_i x_i$	$\sum v_i x_i$	
①	(1/2,1/3,1/4)	16.5	24.25	//没有装满背包//
②	(1, 2/15, 0)	20	28.2	
③	(0, 2/3, 1)	20	31	
④	(0, 1, 1/2)	20	31.5	

贪心算法解背包问题

贪心策略求解背包问题

度量标准的选择：三种不同的选择

- 以目标函数作为度量
  - 每装入一件物品，就使背包获得最大可能的价值增量。
  - 该度量标准下的处理规则是：
    - 按利益值的非增次序将物品一件件地放入到背包；
    - 如果正在考虑的物品放不进去，则只取其一部分装满背包：

贪心算法解背包问题

- ***Example:***
  - n=3，C=20，(v_1,v_2,v_3) = (25,24,15)， (w_1,w_2,w_3) = (18,15,10)。

    ∵ $v_1>v_2>v_3$

    - ∴ 首先将物品1放入背包，此时$x_1=1$，背包获得$v_1=25$的利益增量，同时背包容量减少$w_1=18$个单位，剩余空间ΔC=2。
    - 其次考虑物品2和3。就ΔC=2而言有，只能选择物品2或3的一部分装入背包。
      - ✓ 物品2：若 $x_2=2/15$， 则 $v_2\,x_2=16/5=3.1$
      - ✓ 物品3：若 $x_3=2/10$， 则 $v_3\,x_3=3$
      - ✓ 为使背包的效益有最大的增量，应选择物品2的2/15装包，即$x_2=2/15$
    - 最后，背包装满， ΔC=0，物品3不装包，即$x_3=0$ 。
    - 背包最终可以获得利益值＝ $x_1\,v_1+x_2\,v_2+x_3\,v_3$ ＝ 28.2 (**次优解**,非问题的最优解)

贪心算法解背包问题

- 以容量作为度量标准

Problem: 以目标函数作为度量标准所存在的问题：尽管背包的利益值每次得到了最大的增加，但背包容量也过快地被消耗掉了，从而不能装入“更多”的物品。

Solution: 让背包容量尽可能慢地消耗，从而可以尽量装入“较多”的物品。

即，以容量作为度量

- 该度量标准下的处理规则：
  - 按物品重量的非降次序将物品装入到背包；
  - 如果正在考虑的物品放不进去，则只取其一部分装满背包；

贪心算法解背包问题

- ***Example:***
  - n=3，C=20，(v_1,v_2,v_3) = (25,24,15)， (w_1,w_2,w_3) = (18,15,10)。

    ∵ $w_3<w_2< w_1$

    ➢ ∴ 首先将物品3放入背包，此时x_3＝1，背包获得v_3＝15的利益增量，同时背包容量减少w_3＝10个单位，剩余空间ΔC=10。

    ➢ 其次考虑物品2和1。就ΔC=10而言有，也只能选择物品2或1的一部分装入背包。

    ✓ 物品2： 若 x_2＝10/15， 则 $v_2\,x_2$＝16

    ✓ 物品1： 若 x_1＝10/18， 则 $v_1\,x_1$＝13.9

    ✓ 为使背包的效益有最大的增量，应选择物品2的10/15装包，即x_2=10/15

    ➢ 最后，背包装满ΔC=0，物品1将不能装入背包，故 x_1＝0 。

    ➢ 背包最终可以获得利益值＝ $x_1\,v_1$ ＋$x_2\,v_2$＋$x_3\,v_3$ ＝ 31 (次优解,非问题的最优解)

存在的问题：效益值没有得到“最大程度”的增加

贪心算法解背包问题

- 最优的度量标准
  - 影响背包利益值的因素：
    - 背包的容量C
    - 放入背包中的物品的重量及其可能带来的利益值
  - 可能的策略是：在背包利益值的增长速率和背包容量消耗速率之间取得平衡，即每次装入的物品应使它所占用的每一单位容量能获得当前最大的单位利益。
  - 这种策略下的量度：已装入的物品的累计利益值与所用容量之比。
  - ***Solution:*** 新的量度标准是：每次装入要使累计利益值与所用容量的比值有最多的增加（首次装入）和最小的减小（其后的装入）。

    此时，将按照物品的**单位利益值：v_i/w_i 比值的非增次序**考虑。

贪心算法解背包问题

- ***Example:***
  - n=3，C=20，(v_1,v_2,v_3) = (25,24,15)， (w_1,w_2,w_3) = (18,15,10)。
    - ∵ $v1/w1 < v3/w3 < v2/w2$ ∴ 首先将物品2放入背包，此时x_2＝1，背包获得v_2＝24的利益增量，同时背包容量减少w_2＝15个单位，剩余空间ΔC=5。
    - 其次，在剩下的物品1和3中，应选择物品3,且就ΔC=5而言有，只能放入物品3的一部分到背包中 。即 x_3＝5/10＝1/2
    - 最后，背包装满ΔC=0，物品1将不能装入背包，故x_1＝0 。
    - 最终可以获得的背包利益值＝ $x_1 v_1 + x_2 v_2 + x_3 v_3$ ＝ 31.5 (最优解)

贪心算法解背包问题

- **GreedyKnapsack算法描述**
  - 用贪心算法解背包问题的基本步骤
    - 计算每种物品单位重量的价值v_i/w_i，
    - 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。
  - 用贪心算法解背包问题的效率
    - 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$O(n\log n)$。
    - 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。

贪心算法解背包问题

- 贪心算法求解背包问题

```
void GreedyKnapsack(int n, float M, float v[],float w[],float x[])
{
      Sort(n,v,w); //v(1:n)和w(1:n)分别含有按v(i)/w(i)≥v(i＋1)/w(i＋1)排序的n 件物品的效益值和重量。M是背包的容量大小，而x(1:n)是解向量//
      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];
}
```

贪心算法解背包问题

最优解的证明

- 要证明什么？
  - 即证明：由第三种策略所得到的贪心解是问题的最优解。
  - 最优解：在满足约束条件的情况下，使目标函数取极值的可行解。
  - 贪心解是可行解，故只需证明：贪心解可使目标函数取得极值

贪心算法解背包问题

- **证明的基本思想：**
  - 将此贪心解与（假设中的）任一最优解相比较。
  - 如果这两个解相同，则显然贪心解就是最优解。
  - 如果这两个解不同，就设法去找两者开始不同的第一个分量位置*i*，
  - 然后设法用贪心解的这个x_i去替换最优解对应的分量，并证明最优解在分量代换前后总的利益值没有任何变化(且不违反约束条件)。
  - 然后比较二者。若还不同，则反复进行代换，直到代换后产生的“最优解”与贪心解完全一样。
  - 在上述代换中，最优解的利益值没有任何损失，从而证明贪心解的利益值与代换前后最优解的效益值相同。即，贪心解如同最优解一样可取得目标函数的最大/最小值。
  - 从而得证：该贪心解即是问题的最优解。

贪心算法解背包问题

- **定理 如果$p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$, 则算法GreedyKnapsack对于给定的背包问题实例生成一个最优解。**
- **证明：**
  - 设$X=(x_1, x_2, \ldots, x_n)$是GreedyKnapsack所生成的贪心解。
  - 如果所有的x_i都等于1，则显然X 就是问题的最优解。
  - 否则， 设 j 是使$x_i \neq 1$的最小下标。由算法的执行过程可知，

    $x_i = 1$, $1 \leq i < j$,

    $x_j = [0, 1]$

    $x_i = 0$, $j < i \leq n$

贪心算法解背包问题

- 设$Y = (y_1, y_2, \dots, y_n)$是问题的最优解，且有$\sum w_i y_i = C$
- 若$X = Y$，则X 就是最优解。
- 否则， X 和 Y 至少在1个分量上存在不同。设k是使得$y_k \neq x_k$的最小下标，则有$y_k < x_k$。可分以下情况说明：

  a) 若$k < j$，则x_k=1。因为$y_k \neq x_k$，从而有$y_k < x_k$

  b) 若$k = j$，由于 $\sum w_i x_i = C$ ，且对$1 \le i < j$，有$y_i = x_i = 1$, 而对$j < i \le n$，有$x_i = 0$；故此时若$y_k > x_k$，则将有$\sum w_i y_i > C$ ，与Y是可行解相矛盾。而$y_k \neq x_k$, 所以$y_k < x_k$

  c) 若$k > j$，则 $\sum w_i y_i > C$ ，不能成立

- 若在Y中作以下调整：将y_k增加到x_k，为保持解的可行性，必须从$(y_{k+1}, \dots, y_n)$中减去同样多的量。设调整后的解为$Z=(z_1, z_2, \dots, z_n)$，其中$z_i = x_i$，$1 \le i \le k$，且有： $\sum_{k<i\le n} w_i(y_i - z_i) = w_k(z_k - y_k)$

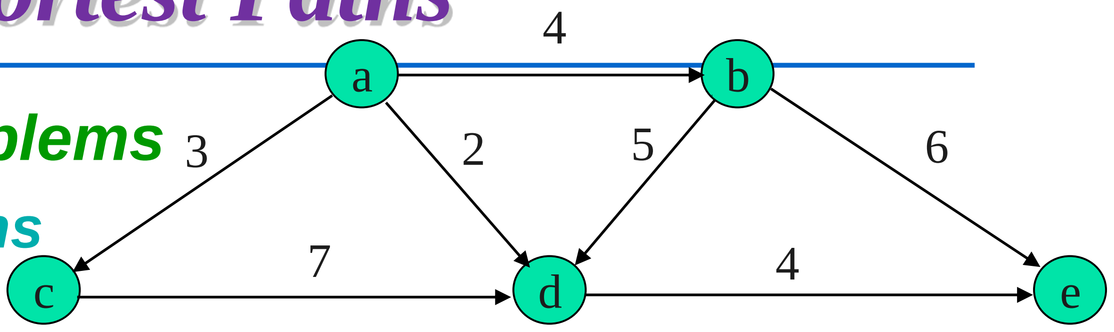
贪心算法解背包问题

Z的利益值有：
$$\sum_{1\le i\le n} v_i z_i = \sum_{1\le i\le n} v_i y_i + (z_k - y_k) w_k v_k / w_k - \sum_{k<i\le n} (y_i - z_i) w_i v_i / w_i$$
$$\ge \sum_{1\le i\le n} v_i y_i + [(z_k - y_k) w_k - \sum_{k<i\le n} (y_i - z_i) w_i] v_k / w_k$$
$$= \sum_{1\le i\le n} v_i y_i$$

由以上分析得，

- 若 $\sum v_i z_i > \sum v_i y_i$ ，则Y将不是最优解；
- 若 $\sum v_i z_i = \sum v_i y_i$ ，且$Z=X$，则X就是最优解；
- 或者$Z\neq X$，则重复以上替代过程，或者证明Y不是最优解，或者把Y转换成X，从而证明X是最优解

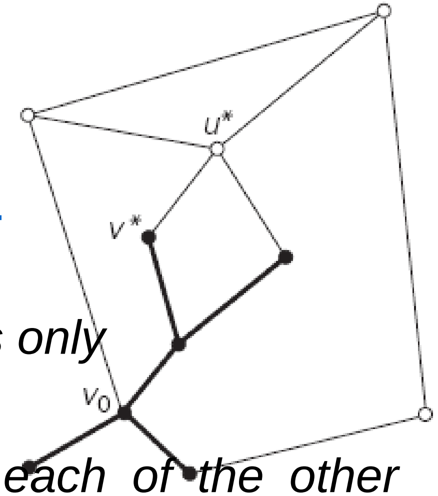
Single-source Shortest Paths



- **Shortest Path Problems**
  - ***All pair shortest paths***
    - *Floy's algorithm*
  - ***Single Source Shortest Paths***
    - *Dijkstra's algorithm*
    - *Given a weighted graph G, find the shortest paths from a source vertex s to each of the other vertices*
      - *not a single shortest path that starts at the source and visits all the other vertices. e.g. traveling salesman*
  - ***Applications***

    - transportation planning
    - *packet routing in communication networks*
    - *finding shortest paths in social networks*
    - *speech recognition, document formatting, robotics, compilers*
    - *airline crew scheduling.*
    - *path-finding in video games*

Single-source Shortest Paths



Dijkstra's alg.

- *undirected and directed graphs with nonnegative weights only*
- *one source vertex s*
- *to find the shortest paths from a source vertex s to each of the other vertices*

Idea of Dijkstra's

- *Start with a subtree consisting of a single source vertex*
- *a sequence of expanding subtrees, T_1, T_2, … --- one vertex/edge at a time*
  - *In general, before its i^{th} iteration, the alg. has already identified the shortest paths to i -1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_{i-1} of the given graph.*
  - *Keep track of shortest path from source to each of the vertices in T_i*

Single-source Shortest Paths

- ***Dijkstra's alg.***
  - ***Idea of Dijkstra's***
    - *On each iteration, $T_i \rightarrow T_{i+1}$ connecting a vertex in tree (T_i) to one not yet in tree*
      - *expands the current tree in the greedy*
    - *the next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i .*
      - *referred to as "fringe vertices"; candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.*
      - *Actually, all the other vertices can be treated as fringe vertices connected to tree vertices by edges of infinitely large weights*

Single-source Shortest Paths

- *Idea of Dijkstra’s-- greedy*

- 设*S*是已经对其生成了最短路径的结点集合（包括源v_0)。
- 如果下一条最短路径是到结点*u*，则这条路径是从结点v_0出发，在*u*处终止，且只经过那些在*S*中的结点，即由v_0至*u*的这条最短路径上的所有中间结点都是*S*中的结点，证明如下：

- 设*w*是这条路径上的任意中间结点，则从v_0到*u*的路径也包含了一条从v_0到*w*的路径，且其长度小于从v_0到*u*的路径长度。
- 根据生成规则：最短路径是按照路径长度的非降次序生成的，因此从v_0到*w*的最短路径应该已经生成。从而*w*也应该在*S*中。
- 故：从结点v_0到*u*的最短路径上，

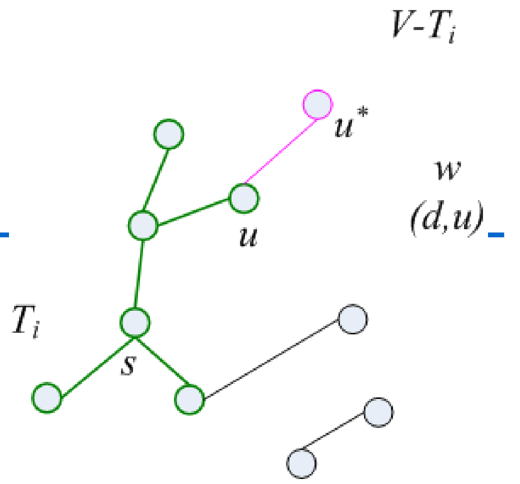
$$\underset{\substack{\uparrow \\ \text{均在S中}}}{\underline{v_0,\ s_1,\ s_2,\ \cdots,\ w,\ \cdots,\ s_{m-1},}}\ u$$

不存在不在*S*中的中间结点。

Single-source Shortest Paths

Dijkstra's alg.

Idea of Dijkstra's ('cont)



- *label each vertex w with two labels.*
  - *The numeric label d indicates the length of the shortest path from the source to this vertex found by the algorithm so far*
  - *The other label u indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed.*
- *finding the next nearest vertex u∗ becomes a simple task of finding a fringe vertex with the smallest d value.*

$$u^* = \min_{w \in V - T_i} d[w] = \min_{w \in V - T_i} (d[u] + c(w,u))$$

  the sum of
  - *the distance to the nearest tree vertex u , i.e. weight of the edge **c(w, u)***
  - *and, the length of the shortest path from the source to u (previously determined by the algorithm), i.e. **d[u]***

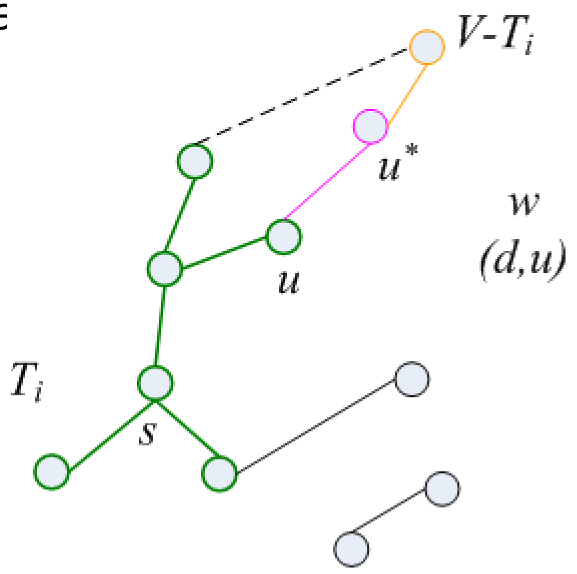
  → *edge (u*,u) with lowest d[u] + c(u*, u)*
- *stops when all vertices are included.*

Single-source Shortest Paths

Dijkstra's alg.

Idea of Dijkstra's ('cont)

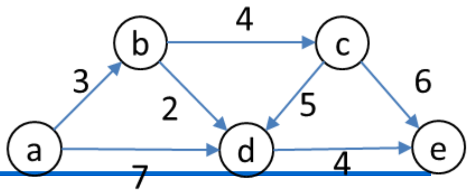
- *After we have identified a vertex u∗ to be added to the tree, make some modification to other fringe vertices*
  - *Move u∗ from the fringe to the set of tree vertices.*
  - *For each remaining fringe vertex w that is connected to u∗ by an edge of weight c(u∗, w) such that d[u∗] + c(u∗, w) < d[w], update the labels of w by u∗ and d[u∗] + c(u∗, w), respective*



Single-source Shortest Paths: Dijkstra's

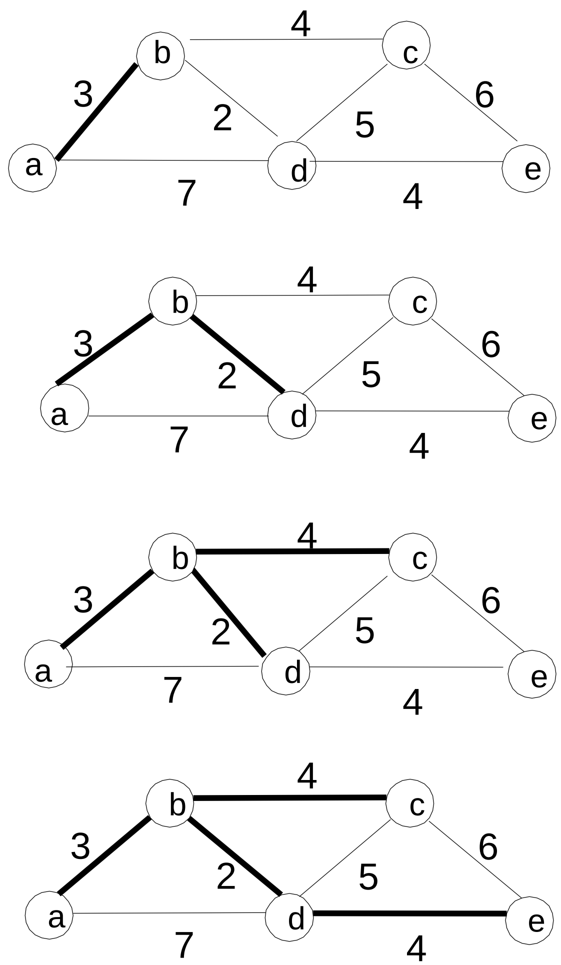
```
ALGORITHM Dijkstra(G, s)
//Input: A weighted connected graph $G = <V, E>$ and a source vertex $s$
//Output: The length $d_v$ of a shortest path from s to v and its penultimate vertex $p_v$
for every vertex $v$ in $V$

Initialize (Q)                  //initialize vertex priority in the priority queue
for every vertex v in V do
          $d_v \leftarrow \infty$ ; $P_v \leftarrow$ null  // $P_v$ , the parent of $v$
          insert(Q, v, $d_v$)      //initialize vertex priority in the priority queue

$d_s \leftarrow 0$; Decrease(Q, s, $d_s$)      //update priority of $s$ with $d_{s,}$ making $d_{s,}$ the minimum
$V_T \leftarrow \varnothing$

for $i \leftarrow 0$ to $|V| - 1$ do         //produce $|V| - 1$ edges for the tree
          $u^* \leftarrow$ DeleteMin(Q)           //delete the minimum priority element
          $V_T \leftarrow V_T$ U {$u^*$}      //expanding the tree, choosing the locally best vertex
          for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
                    if $d_{u*} + w(u^*, u) < d_u$
                              $d_u \leftarrow d_u + w(u^*, u)$; $p_u \leftarrow u^*$
                              Decrease(Q, u, $d_u$)
```

Single-source Shortest Paths



- *Example : Dijkstra's*

Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) d(b,3+2) e(-,∞)
d(b,5)	c(b,7) e(d,5+4)
c(b,7)	e(d,9)
e(d,9)	



The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:
from a to b : a − b of length 3
from a to d : a − b − d of length 5
from a to c : a − b − c of length 7
from a to e : a − b − d − e of length 9

Single-source Shortest Paths

- ***Correctness of Dijkstra's***

- Dijkstra's algorithm, run on a weighted, directed graph $G=(V, E)$, with non-negative weight function ***c*** and source v_0, terminates with d[u]= $\delta(s,u)$ for all vertices $u \in V$. Here $\delta(s,u)$ means the shortest distance from s to u.
- ***Proof (by contradiction)***

- Since $S = V$ in the end and for each vertex v, after it was put into S, the value of d[u] would never be changed. we only need to prove :

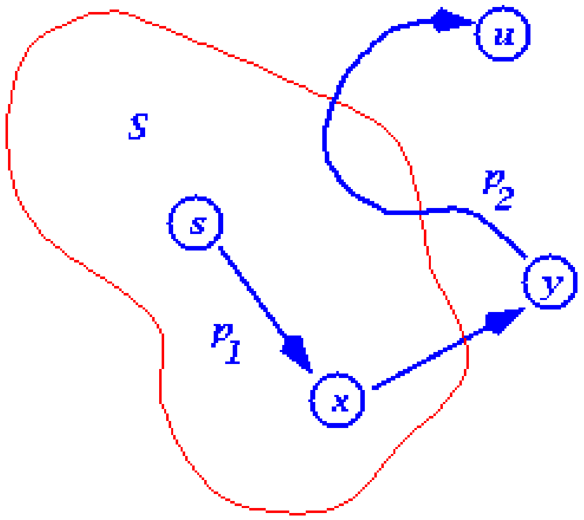
☞ for each vertex v added to S, there **holds d[*v*]= δ(*s*, *v*) when *v* is added to *S*.**

- Suppose that added to S u is the first vertex for which d[u] $\neq \delta(s, u)$ when it was
- Note
- u is not s because d[s] = 0= d(s, s)
- There must be a path $s \rightarrow \ldots \rightarrow u$, since otherwise d[$u$]= $\delta(s, u) = \infty$.
- Since there's a path, there must be a shortest path (note there is no negative cycle).

Single-source Shortest Paths

Correctness of Dijkstra's (‘cont)

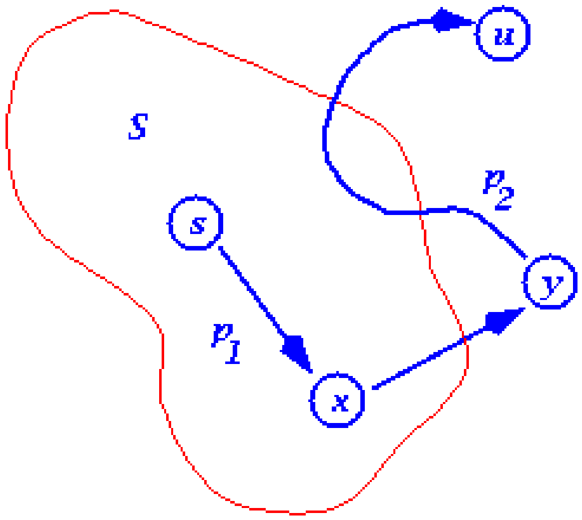
- Let $s \rightarrow x \rightarrow y \rightarrow u$ be the shortest path from s to u, where at the moment u is chosen to S,
- x is in S and y is the first outside S (y may not exist)
- When x was added to S, d[x] = $\delta(s, x)$
- Edge $x \rightarrow y$ will be considered at that time, d[y] <= d(s, x)+c(x, y) = $\delta(s, y)$



Single-source Shortest Paths

Correctness of Dijkstra's ('cont)

- so $d[y] = \delta(s, y) \le \delta\ (s, u) \le d[u]$
- But, when we chose u, both u and y are in Q,
  so $d[u] \le d[y]$
  (otherwise we would have chosen y)
- Thus the inequalities must be equalities
- $d[y] = d(s, y) = d(s, u) = d[u]$
- And our hypothesis ($d[u] \ne d(s, u)$) is contradicted!



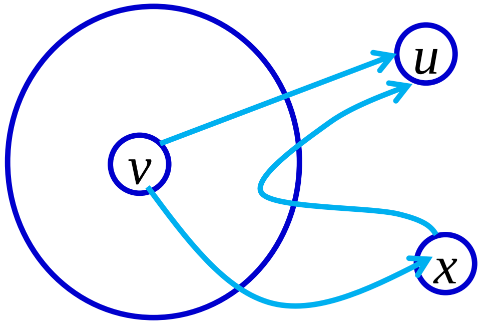
Single-source Shortest Paths

Dijkstra 算法的贪心选择性质

- Dijkstra 算法的贪心选择是从*V-S*中选择具有最短特殊路径的顶点*u*，从而确定从源v_0到*u*的最短路径长度dist[*u*] 。
- 是最优解吗？有无从源v_0到*u*的更短的其它路径？
  - 如果存在一条从源v_0到*u*且长度比dist[*u*]更短的路，设这条路初次走出*S*外到达的顶点为$x \in V\text{-}S$，然后徘徊于*S*内外若干次，最后离开*S*到达*u*。

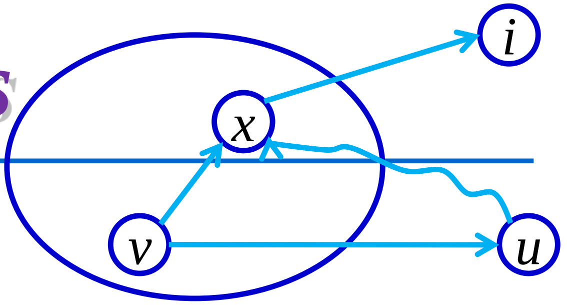
    $\text{dist}[x] \le d(v,x)$

    $d(v,x) + d(x,u) = d(v,u) < \text{dist}(u)$



  - 利用边的非负性可知$d(x,u) \ge 0$， ⇨ $\text{dist}[x] < \text{dist}[u]$。
  - 矛盾！

Single-source Shortest Paths



Dijkstra 算法的最优子结构性质

- 要证明算法中确定的dist[*u*]确实是当前从源到顶点*u*的最短特殊路径长度。
- 为此，我们只要考虑算法在添加*u*到*S*中后，dist[*u*]的值所起的变化。
- 我们将添加*u*之前的*S*称为老的*S*。当添加了*u*之后，可能出现一条到顶点*i*的新的特殊路。
  - 如果这条新特殊路是先经过老的*S*到达顶点*u*，然后从*u*经一条边直接到达顶点*i*，则这种路的最短的长度是dist[*u*] +c[*u*][*i*]。
    - 如果dist[*u*] +c[*u*][*i*]<dist[*i*]，则算法中用dist[*u*] +c[*u*][*i*]作为dist[*i*]的新值。
  - 如果这条新特殊路径经过老的*S*到达*u*后，不是从*u*经一条边直接到达*i*，而是像图4-8那样，回到老的*S*中某个顶点*x*,最后才到达顶点*i*，那么由于*x*在老的*S*中，因此*x*比*u*先加入*S*，故图中从源到*x*的路的长度比从源到*u*，再从*u*到*x*的路的长度小。
    - 于是当前dist[*i*]的值小于图中从源经*x*到*i*的路的长度，也小于图中从源经*u*和*x*，最后到达*i*的路的长度。因此，我们在算法中不必考虑这种路。

⇨ 由此即知，不论算法中dist[*u*]的值是否有变化，它总是关于当前顶点集*S*到顶点*u*的最短特殊路径长度。

Single-source Shortest Paths : Dijkstra's

- ***Efficiency of Dijkstra's***
  - *Use unordered array to store the priority queue: Θ(n^2)*
  - *Use min-heap to store the priority queue: O(m log n)*

Single-source Shortest Paths : Dijkstra's

Efficiency of Dijkstra's

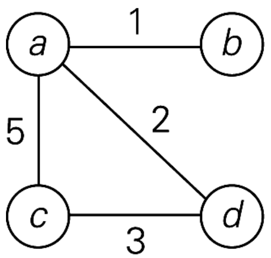
- 上述算法的计算时间：$O(n^2)$

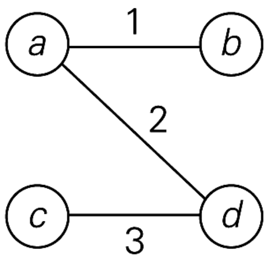
```
(1)  for i←1 to n do                                    $\Theta(n)$
         S(i) ←0;DIST(i) ←COST(v,i)
     repeat
(2)  for num←2 to n-1 do                                $O(n-2)$
         选取结点u,它使得DIST(u)=  $\min_{S(w)=0}\{DIST(w)\}$
         S(u) ←1                                        $O(n)$
         for 所有S(w)＝0的结点w do                        $O(n)$
             DIST(w) = min(DIST(w), DIST(u) + COST(u,w))
         repeat
     repeat
```

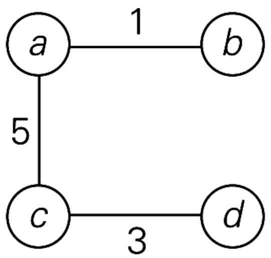
Minimum Spanning Tree

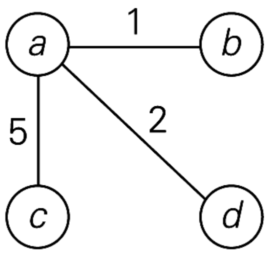
- ***Problem:***
  - *G =(V,E): undirected connected graph。*

    edge weights for (v,w)∈E
  - *Spanning Tree: a connected acyclic subgraph (tree) of G that includes all of G's vertices*
    - *Note: a spanning tree with n vertices has exactly n-1 edges.*
  - *weight of a tree: the sum of the weights on all its edges*
  - *minimum spanning tree: spanning tree of the smallest weight*


graph


$w(T_1) = 6$


$w(T_2) = 9$


$w(T_3) = 8$

Minimum Spanning Tree

- ***Applications***
  - *problem: given n points, connect them in the cheapest possible way so that there will be a path between every pair of points.*
  - *design of all kinds of networks by providing the cheapest way to achieve connectivity*
    - *communication, computer, transportation, electrical*
  - *identifies clusters of points in data sets*
  - *classification purposes in archeology, biology, sociology,*

→ minimum spanning tree

the points ----- vertices of a graph

connections ---- the graph's edges

connection costs ---- the edge weights

Minimum Spanning Tree

- **Problem:**
  - **MST problem:** *Given a connected, undirected, weighted graph G= (V, E), find a minimum spanning tree for it.*
    - *Compute MST through Brute Force?*
      - *Brute force*

        - generate all possible spanning trees for the given graph.

        - find the one with minimum total weight.
      - *Feasibility of Brute force*

        - Possible too many trees (exponential for dense graphs)
    - *Kruskal: 1956, Prim: 1957*

Minimum Spanning Tree

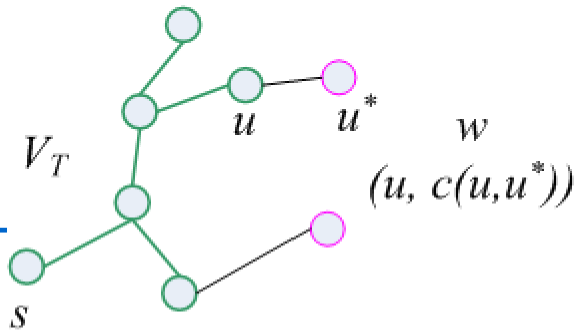
- **Prim**
  - **Idea of Prim**
    - *initial subtree T_0 consists of a single vertex selected arbitrarily from set V.*
    - *a sequence of expanding subtrees, T_1, T_2 … --- one vertex/edge at a time*
      - *$T_{i-1} \rightarrow T_i$ each stage*
      - *expands the current tree in the greedy*
    - *On each iteration, attaching to tree the nearest vertex not in that tree.*
      - *the nearest vertex: a vertex not in the tree, and connected to a vertex in the tree by an edge of the smallest weight.*
    - *stops after all the graph's vertices have been included in the spanning tree*
    - *the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n − 1, where n is the number of vertices in the graph.*
    - *The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.*

Minimum Spanning Tree

Prim -I

```
ALGORITHM Prim(G)
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = (V, E)$
//Output: $E_T$, the set of edges composing a minimum spanning tree of
    $G$
$V_T \leftarrow \{v_0\}$    //$v_0$ can be arbitrarily selected
$E_T \leftarrow \phi$
for $i \leftarrow 1$ to $|V|-1$ do
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v,$
    $u)$ such that $v$ is in $V_T$ and $u$ is in $V-V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return $E_T$
```

Minimum Spanning Tree



- ***Idea of Prim ('cont)***
  - *How to find the minimum weight edge that connecting a vertex in tree $T_{i\text{-}1}$ to a vertex not yet in the tree ?*

    → attaching two labels to a vertex:

    - the name of the nearest tree vertex

    - the length (the weight) of the corresponding edge
  - *finding the next vertex to be added to the current tree $T_i = (V_T, E_T)$*

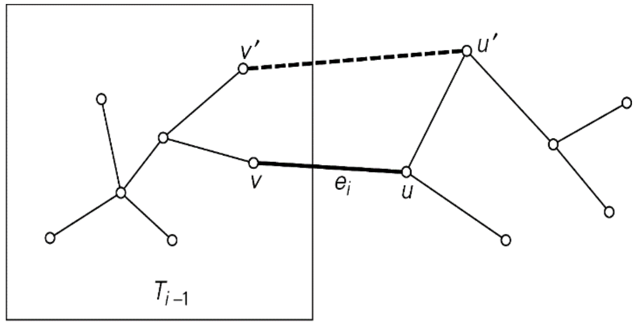
    ↔ finding a vertex with the smallest distance label in the set $V - V_T$

    $u \in V_T$, $u* \in V\text{-}V_T$, $\min\{ c[u, u*] \}$
  - *After we have identified a vertex u^* , make some modification to u's neighbors*
    - *Move u^* from the set $V - V_T$ to the set of tree vertices V_T .*
    - *For each remaining vertex w in $V - V_T$ that is connected to $u*$ by a shorter edge than the u's current distance label, update its labels by $u*$ and the weight of the edge between $u*$ and u, respectively.*

Minimum Spanning Tree

- *Prim is greedy*
  - *The choice of edges added to current subtree satisfying the three properties of greedy algorithms.*
    - *Feasible, each edge added to the tree does not result in a cycle, guarantee that the final ET is a spanning tree*
    - *Local optimal, each edge selected to the tree is always the one with minimum weight among all the edges crossing V_T and $V\text{-}V_T$*
    - *Irrevocable, once an edge is added to the tree, it is not removed in subsequent steps.*

Minimum Spanning Tree



Correctness of Prim

- *Prove by induction that this construction process actually yields MST.*
- *T_0 consists of a single vertex and must be a subset of any MST*
- *Assume that T_{i-1} is a subset of some MST T, we should prove that T_i which is generated from T_{i-1} is also a subset of some MST.*

  → By contradiction,

  - assume that T_i does not belong to any MST.

  - Let $e_i = (u, v)$ be the minimum weight edge from a vertex in T_{i-1} to a vertex not in T_{i-1} used by Prim's algorithm to expanding T_{i-1} to T_i ,

  - according to our assumption, e_i can not belong to any MST T.

  - So, adding e_i to T results in a cycle, which must containing another edge e' = (u', v') connecting a vertex v' in T_{i-1} to a vertex u' not in it, and $w(e') \geq w(e_i)$ according to the greedy Prim's algorithm.

  - Removing e' from the circle and adding e_i to T results in another spanning tree T' with weight $w(T') \leq w(T)$, indicating that T' is a minimum spanning tree including T_i which contradict to assumption that T_i does not belong to any MST.

Minimum Spanning Tree

Implementation of Prim

- *Method I.*
- *label each vertex with either 0 or 1, 1 represents the vertex in V_T, and 0 otherwise.*
- *Traverse the edge set to find an minimum weight edge whose endpoints have different labels.*
- *Time complexity: O(VE) if adjacency linked list and O(V^3) for adjacency matrix*

- For sparse graphs, use adjacency linked list

- *Method II*

At each stage, the key point of expanding the current subtree T_i is to

- *Determine which vertex in V-V_T is the nearest vertex to T.*

- V-V_T can be thought of as a priority queue:

- The key (priority) of each vertex, key[u], means the minimum weight edge from u to a vertex in T. Key[u] is ∞ if u is not linked to any vertex in T.

- The major operation is to find and delete the nearest vertex u^*, for which key[u^*] is the smallest among all the vertices

- *Remove the nearest vertex u^* from V-V_T and add it to T.*

- With the occurrence of that action, the key of u's neighbors will be changed.

Minimum Spanning Tree

Prim -II

```
ALGORITHM MST-PRIM( G, $w$, $r$ ) //w: weight; r: root, the starting vertex
1.      for each $u \in V[G]$
2.          do $key[u] \leftarrow \infty$
3.              $\pi[u] \leftarrow$ NIL            // $\pi[u]$ : the parent of u
4.      $key[r] \leftarrow 0$
5.      $Q \leftarrow V[G]$      //Now the priority queue, Q, has been built.
6.      while $Q \neq \emptyset$
7.          do $u \leftarrow$ Extract-Min$(Q)$ //remove the nearest vertex from Q
8.              for each $v \in Adj[u]$  // update the key for each of v's adjacent
        nodes.
9.                  do if $v \in Q$  and $w(u,v) < key[v]$
10.                     then $\pi[v] \leftarrow u$
11.                         $key[v] \leftarrow w(u,v)$
```

an array π[] is introduced to record the parent of each vertex.
That is π[*u*] is the vertex in the expanding subtree *T* that is closest to *u* not in *T*.

Minimum Spanning Tree

- *Efficiency*
  - *Need priority queue for locating the nearest vertex*
  - *Use unordered array to store the priority queue:*

    Efficiency: $\Theta(n^2)$
  - *Use binary min-heap to store the priority queue*

    Efficiency: For graph with n vertices and m edges:

    O(m log n)
  - *Use Fibonacci-heap to store the priority queue:*

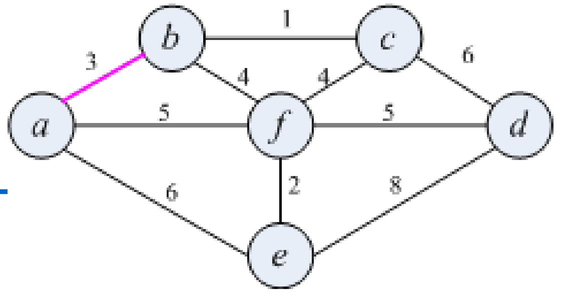
Efficiency: For graph with n vertices and m edges:

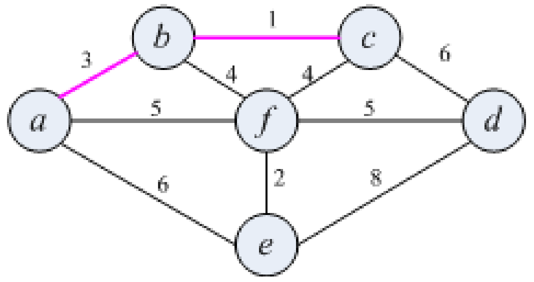
O(n log n + m)

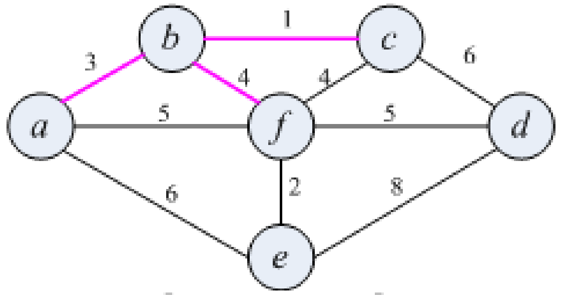
Minimum Spanning Tree

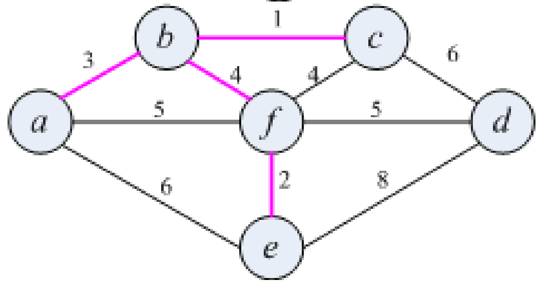
- *Example : Prim*

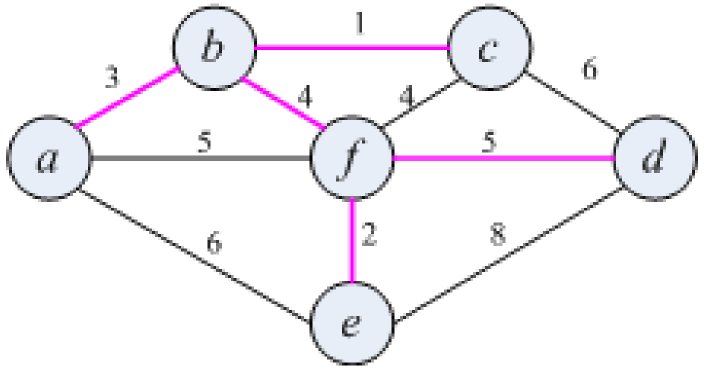
Tree vertices	Remaining vertices
a(-,0)	<u>b(a,3)</u> c(-,∞) d(-, ∞) e(a,6) f(a,5)
b(a,3)	<u>c(b,1)</u> d(-, ∞) e(a,6) f(b,4)
c(b,1)	d(c, 6) e(a,6) <u>f(b,4)</u>
f(b,4)	d(f, 5) <u>e(f,2)</u>
e(f,2)	<u>d(f, 5)</u>
d(f, 5)	











Minimum Spanning Tree

Kruskal's algorithm

Idea of Kruskal's

- *looks at a minimum spanning tree of a weighted connected graph G =(V,E) as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest.*
- *sorting the graph's edges in nondecreasing order of their weights.*
- *starting with the empty subgraph, it scans this sorted list,*
- *adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.*

  - need efficient way of detecting/avoiding cycles
- *algorithm stops when all vertices are included*

Minimum Spanning Tree

Kruskal's

```
ALGORITHM Kruscal(G)
//Input: A weighted connected graph $G = <V, E>$
//Output: $E_T$, the set of edges composing a minimum spanning tree of G.

Sort E in nondecreasing order of the edge weights $w(e_{i_1}) <= ... <= w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; ecounter $\leftarrow 0$          //initialize the set of tree edges and its size
$k \leftarrow 0$
while encounter < $|V| - 1$ do
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$ ; ecounter $\leftarrow$ ecounter + 1
return $E_T$
```

Minimum Spanning Tree

Kruskal's Algorithm (Advanced Part)

- *Some reviews:*
- *Kruskal's algorithm has to check whether the addition of the next edge to the edges already selected would create a cycle.*
- *a new cycle is created ↔ the new edge connects two vertices already connected by a path, ↔ the two vertices belong to the same connected component (Figure 9.6).*
- *Note also that each connected component of a subgraph generated by Kruskal's algorithm is a tree because it has no cycles.*

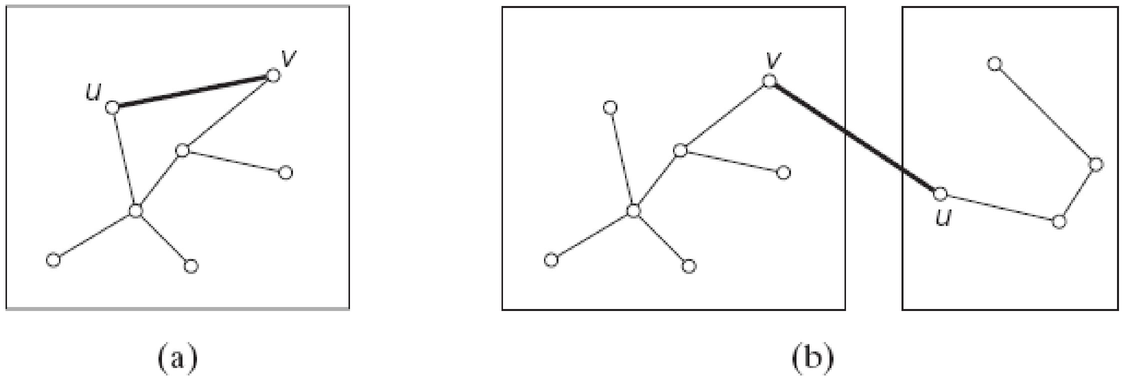


FIGURE 9.6 New edge connecting two vertices may (a) or may not (b) create a cycle.

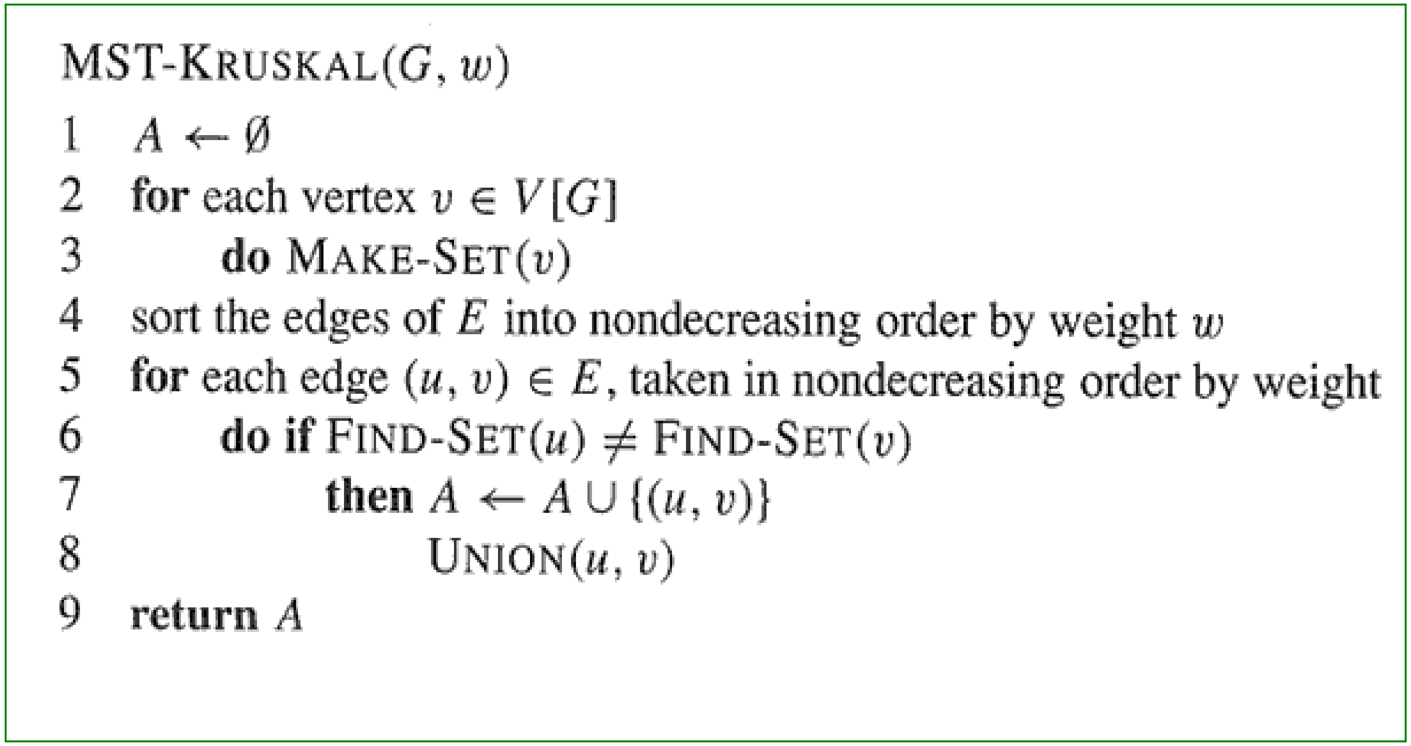
Minimum Spanning Tree

Kruskal's Algorithm (Advanced Part)

- *Kruskal's alg. with a slightly different interpretation.*
- *consider the algorithm's operations as a progression through a series of forests containing all the vertices of a given graph and some of its edges.*
- *The initial forest consists of |V | trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, MST*
- *On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges,*
- *finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).*
- *union-find algorithms..*

Minimum Spanning Tree

Kruskal's (Advanced Part)

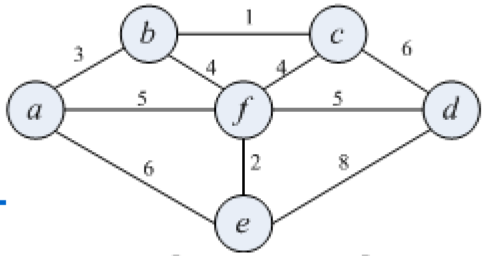


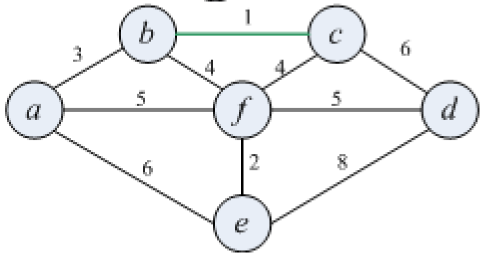
```
MST-KRUSKAL($G, w$)
$A \leftarrow \emptyset$
for each vertex $v \in V[G]$
        do MAKE-SET($v$)
sort the edges of $E$ into nondecreasing order by weight $w$
for each edge $(u, v) \in E$, taken in nondecreasing order by weight
        do if FIND-SET($u$) $\neq$ FIND-SET($v$)
                then $A \leftarrow A \cup \{(u, v)\}$
                        UNION($u, v$)
return $A$
```

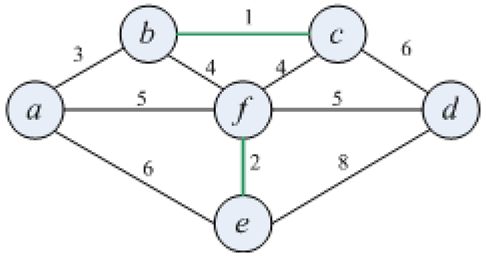
Minimum Spanning Tree

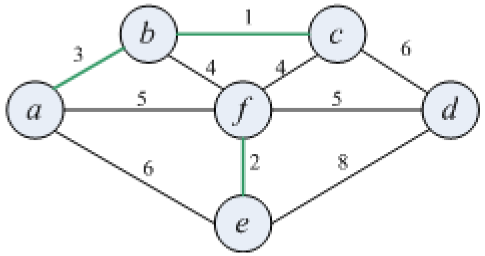
Example: Kruskal's

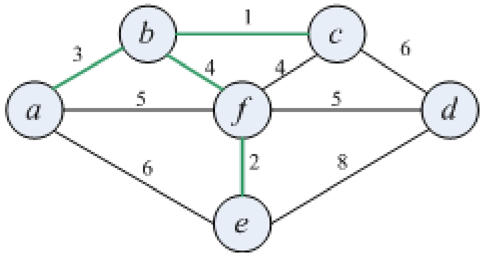
Tree edges	Sorted list of edges									
	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
bc										
1										
ef										
2										
ab										
3										
bf										
4										
df										
5										

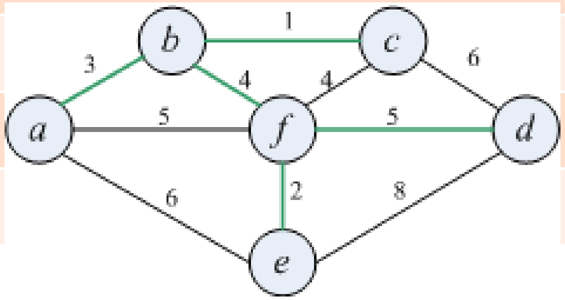












Minimum Spanning Tree

- ***Kruskal's Efficiency***
  - *O(EV) when disjoint-set data structure are not used.*
  - *When use disjoint-set data structure with union-by-rank and path-compression heuristics:*

    a) Initializing the set A in line 1 takes O(1) time.

    b) O(V) MAKE-SET operations in lines 2-3

    c) Sort the edges in line 4 takes O(ElogE) time.

    d) The for loop of lines 5-8 performs O(E) FIND-SET and UNION operations on the disjoint-set forest

    – *b) and d) take a total of $O((V+E)\alpha(V))$*

    – *Line 9: O(V+E)*

    Note that: $E \geq V-1$ and $\alpha(V) = O(\log V) = O(\log E)$ and $E \leq V^2$

    SO total time is: **O(ElogV)**

Minimum Spanning Tree

Properties of MST

Property 1

- *Let (u, v) be a minimum-weight edge in a graph G = (V, E), then (u, v) belongs to some minimum spanning tree of G.*

Property 2

- *A graph has a unique minimum spanning tree if all the edge weights are pairwise distinct.*
- *The converse does not hold.*

Minimum Spanning Tree

- **Properties of MST**
  - **Property 3**
    - *Let T be a minimum spanning tree of a graph G, and let T' be an arbitrary spanning tree of G, suppose the edges of each tree are sorted in non-decreasing order, that is,* $w(e_1) \le w(e_2) \le \ldots \le w(e_{n-1})$ *and* $w(e_1') \le w(e_2') \le \ldots \le w(e_{n-1}')$, *then for* $1 \le i \le n-1$, $w(e_i) \le w(e_i')$.
  - **Property 4**
    - *Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T, then for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.*

Minimum Spanning Tree

- **Properties of MST**
  - **Property 5**
    - *Let e=(u, v) be a maximum-weight edge on some cycle of G. Prove that there is a minimum spanning tree that does not include e.*
    - *Proof. Arbitrarily choose a MST T. If T does not contain e, it is proved. Otherwise, T\e is disconnected, and suppose X, Y are the two connected components of T\e. Let e is on cycle C in G. Let P=C\e. Then there is an edge (x,y) on P such that $x \in X$, and $y \in Y$. And $w(x, y) \le w(e)$.*

      T'=T\e+(x,y) is a spanning tree and $w(T') \le w(T)$.

      Also we have $w(T) \le w(T')$, so $w(T') = w(T)$.

      T' is a MST not including e.

Single-source Shortest Paths vs. Minimum Spanning Tree

- ***Prim's vs. Dijkstra's Algorithsm***
  - ***Generate different kinds of spanning trees***
    - Prim's: a minimum spanning tree.
    - Dijkstra's : a spanning tree rooted at a given source s, such that the distance from s to every other vertex is the shortest.
  - ***Different greedy strategies***
    - Prims': Always choose the closest (to the tree) vertex in the priority queue Q to add to the expanding tree V_T.
    - Dijkstra's : Always choose the closest (to the source) vertex in the priority queue Q to add to the expanding tree V_T.
  - ***Different labels for each vertex***
    - Prims': parent vertex and the distance from the tree to the vertex..
    - Dijkstra's : parent vertex and the distance from the source to the vertex.